Windows Store Apps Succinctly

by John Garland

Syncfusion | Technology Resource Portal

Windows Store Apps Succinctly

By
John Garland

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by

This publication was edited by Jay Natarajan, senior product manager, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

John Garland is a senior consultant at Wintellect and has been developing software professionally since the 1990s. Prior to consulting, he spent much of his career working on high-performance video and statistical analysis tools for premier sports teams, with an emphasis on the NFL, the NBA, and Division 1 NCAA football and basketball. His consulting clients range from small businesses to Fortune 500 companies and his work has been featured at Microsoft conference keynotes and sessions. John lives in New Hampshire with his wife and daughter, where he is an active participant in the New England development community. When he isn't finding cause for yet another upgrade to some piece of home technology, he occasionally turns to motorcycling and Florida Gator football to unplug. He is a graduate of the University of Florida with a bachelor's degree in computer engineering.

Code Samples

The syntax highlighting used for the code samples in this book is based on Visual Studio 2012.

The code samples found in this book can be downloaded at https://bitbucket.org/syncfusion/windowsapps.

Chapter 1 Core Concepts

Introducing Windows Store Apps

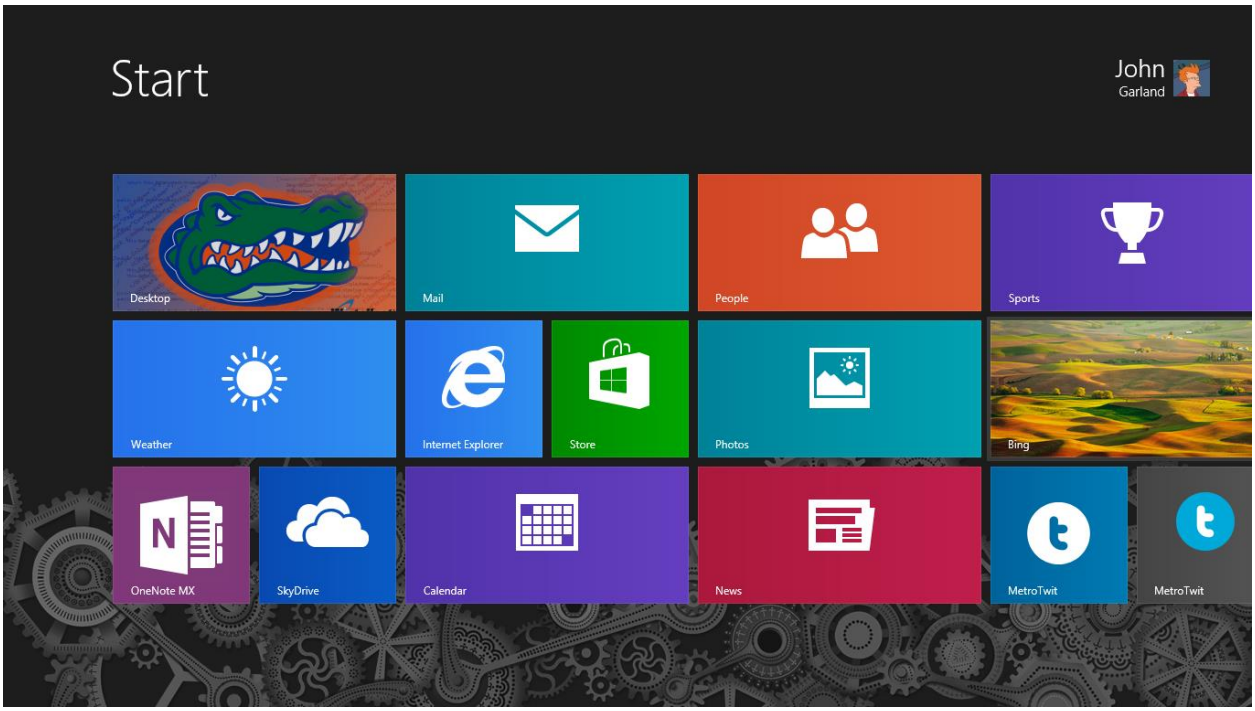


Figure 1: The Windows 8 Start Screen

Windows Store apps are a new kind of application that run on Microsoft's most recent generation of operating systems. Currently, this includes Windows 8, Windows RT, and Windows Server 2012. When installed, an app can have one or more tiles pinned in user-selected positions on the Windows Start screen. Users can launch the app by simply tapping or clicking one of its tiles. Additionally, some applications can be launched by Windows itself as a result of user interaction with common Windows interface elements, including the charms bar, which provides a focal point for accessing common functions such as in-app searching (Search charm), app-to-app data exchange (Share charm), hardware interaction (Device charm), and configuring settings and preferences (Settings charm). Apps can even be launched as a result of scenarios where they have elected to participate in Windows-brokered interactions that are actually initiated from within other applications.

Windows Store apps and the Windows environment they run in feature a new user experience. Apps occupy a single window, and run either full-screen or in a secondary, fixed-size reduced screen known as the "snapped" view. This user experience follows a set of published Microsoft design principles. A summary of these principles includes:

- Content before chrome: Elimination of any frivolous elements that take away from the display of the information or controls being presented to users.

- Fast and fluid: Response to user interactions should be quick and intuitive, and the UI should not “lock up” to support data processing or other background activities.
- Support for multiple view states: The app should handle being displayed in different screen modes, whether it is running as the primary landscape app, in the afore-mentioned snapped landscape view, or full-screen in portrait orientation.
- Support for the right contracts: The app can interact with other Windows components or other apps via the provided contracts and extensions, and should do so to foster and reinforce a powerful and familiar user experience across all apps.
- Live tiles: Even when the app isn’t running, both its primary launching tile and any secondary tiles used to launch the app can come alive and be used to provide app-related information to users.
- Settings and user context roam via the cloud: Apps now have the option to tap into support for moving settings beyond just the local machine, potentially providing users with continuity within the app regardless of what machine they run it from.

Windows provides a controlled environment for Windows Store apps to run in—sometimes known as a “sandbox”—which allows Windows to protect system resources and state from defective or malicious programs. Apps submitted to the Windows Store are qualified against a published set of requirements as part of a certification process that helps to ensure that customers’ systems are not adversely affected by defective or malicious apps. Windows Store apps are digitally signed to provide verification of their authenticity and integrity. Apps published to the store can be offered free of charge; include time-based trials, feature-based trials, or both; or be sold for a fee. In-app purchases and subscriptions are also available for Windows Store apps.

As the name implies, most Windows Store apps are made available for purchase from the centralized Windows Store. This provides development efforts of all sizes, from single hobbyist developers to large corporate concerns, the opportunity to reach a global marketplace of customers with their apps to realize revenue or recognition—or both! However, Windows Store app distribution isn’t limited to the Windows Store—several line-of-business deployment scenarios exist, including deployment via enterprise management systems. This process of deploying an app through a means other than the Windows Store is known as “sideloading.”

A thorough overview of Windows Store apps has been published by Microsoft and can be found at http://msdn.microsoft.com/en-us/windows/apps/hh852650.aspx.

The Windows Runtime

Windows Store apps run on top of a new runtime API called the Windows Runtime, or WinRT. This is a fundamental shift in Windows development, where for quite some time development has occurred on top of one version or another of the Win32 APIs (albeit the presence of Win32 has sometimes been abstracted away by development tools and runtimes, such as MFC, Visual Basic 6, or even .NET). In contrast to the C-style approach afforded by Win32, WinRT provides a more modern object-based surface for application development.

In addition to providing a new API surface for developers to build on, the Windows Runtime also contributes to a development approach that strives to put different development technology stacks on similar footing. Developers can choose to write Windows Store apps using UI/code combinations that (at present) include XAML/.NET, HTML/JavaScript, or XAML/C++, depending on their own backgrounds and preferences. While these tools are positioned to be on somewhat equal footing in terms of their ability to deliver Windows Store apps, they each have their strengths (XAML/C++ has access to DirectX for advanced gaming, for example) which also play into the decisions as to which technology should be used.

Note: The only .NET languages that are presently supported include C# and Visual Basic. To use other languages such as F#, Portable Class Library projects can be used. A discussion of Portable Class Libraries is beyond the scope of this book.

Among several innovations related to these multiple development environments is the concept of "language projections." Windows Runtime components are exposed to each of these environments in a way that is familiar and appropriate to each one. This greatly simplifies the P/Invoke or COM interop process that .NET developers had to go through to access OS APIs. It is important to note that this isn't limited to the Windows Runtime Components provided by the OS—developers can create their own Windows Runtime components in .NET or C++ that expose their functionality through interfaces built from combinations of the standard Windows Runtime types. These components can then be consumed by any Windows Store app development environment, and are also "projected" into language-specific idioms.

Note: Custom Windows Runtime components can be created in C++, C#, or Visual Basic, but not JavaScript.

Another key feature of the WinRT APIs is a fundamental shift to emphasize the use of asynchronous calls for long-running tasks. API methods that might take longer than 50 ms to execute have been implemented as asynchronous methods, with no corresponding synchronous call (as was sometimes the case in other APIs where asynchronous calls were made available). This focus on asynchrony for long-running operations helps to ensure that the UI is not locked while waiting for these operations to complete.

The following code illustrates using the asynchronous API call for creating a file from both C# and JavaScript. The C# code takes advantage of the new **`await`** keyword to support the asynchronous operation, whereas the JavaScript code makes use of JavaScript "promises."

```
// Using CreateFileAsync from C#.
var folder = ApplicationData.Current.LocalFolder;
var file = await folder.CreateFileAsync("NewFileName.txt",
                                        CreationCollisionOption.GenerateUniqueName);
// Work with the file that was created.
```

```
// Using createFileAsync from JavaScript.
var folder = applicationData.current.localFolder;
folder.createFileAsync("NewFileName.txt",
                       storage.CreationCollisionOption.generateUniqueName)
    .then(function (file) {
        // Work with the file that was created.
    });
```

Note: The `async` and `await` keywords are new additions to the C# language. When the compiler encounters these keywords, it will actually generate IL code that sets up the necessary state tracking, allowing the method to return control to its caller and handling the logic necessary to wait on the asynchronous operation to continue. When the asynchronous operation completes, the code that was produced by the compiler will resume execution at what was originally the next line of code in the method. This framework greatly simplifies the writing and readability of asynchronous code, allowing the compiler to manage all of the complex details.

Developing Windows Store Apps with XAML and .NET

Windows Store apps are commonly developed using the development tools that have become familiar to most Windows application developers over the past several years. This includes the Visual Studio 2012 IDE and Expression Blend for Visual Studio 2012, the latter having been enhanced from its roots as a design-centric tool for XAML development to also provide functionality for HTML-based Windows Store app development. As of this writing, Windows Store apps can only be developed on the Windows 8 operating system—when installed on other operating systems, Visual Studio 2012 will not include the options for creating Windows 8 projects.

Tip: You can download Visual Studio Express 2012 for Windows 8 free of charge from Microsoft at http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-8.

Note: While C# and Visual Basic can be considered "sibling languages" for .NET development, and much has been done to create functional parity between these two languages, the rest of this book will be focusing exclusively on C#.

In order to develop Windows Store apps, developers must first obtain a (free) developer license. A developer license is installed on the machine being used to write Windows Store apps. These licenses do expire periodically, so they must be renewed. An active Internet connection is required to obtain a developer license. Visual Studio Express 2012 for Windows 8 automates the process of obtaining or renewing a developer license. When Visual Studio Express is launched and a developer license is not available on the machine, users will be prompted to "Get a developer license for Windows 8." Clicking **I Agree** will ask users for Microsoft Account credentials to obtain the license. Once the credentials are provided, the license will be installed on the machine.

Note: The prompt-on-launch behavior is specific to the Visual Studio Express for Windows 8 SKU. In the other Visual Studio SKUs, the prompt will appear when a Windows Store project is first opened on a machine without a valid developer license.

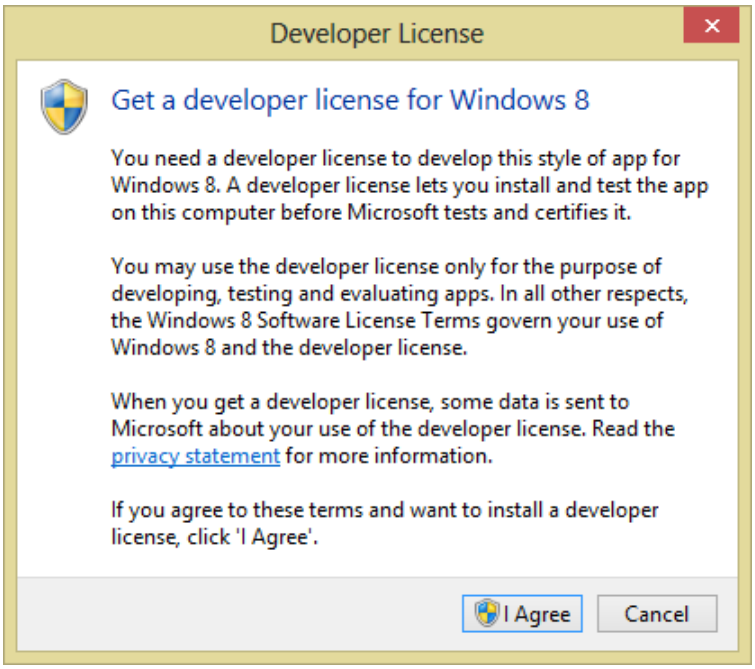


Figure 2: Obtaining a Developer License within Visual Studio 2012

Visual Studio Project Types for .NET Development

There are six default Visual Studio templates available for creating .NET projects for Windows Store apps in Visual Studio 2012 and Expression Blend:

- Blank App (XAML): Creates the simplest executable project, containing a single, empty page. This page is usually not very useful as an application page, as it doesn't include any layout elements typically used in a Windows Store app.
- Grid App (XAML): Creates an executable project featuring three pages for navigating through a grouped hierarchy of elements, including a page for showing the groups, showing an individual group, and showing an individual item.

- Split App (XAML): Creates an executable project featuring two pages for navigating through a grouped hierarchy of elements, including a page for showing the groups, and another for showing a master-detail layout of the items within the selected group.
- Class Library (Windows Store apps): Creates a .NET class library project that can be referenced from other .NET Windows Store app projects.
- Windows Runtime Component: Creates a Windows Runtime component library project that can be referenced by other Windows Store app projects, regardless of the programming language selected.
- Unit Test Library (Windows Store apps): Creates a project that can be used to unit test Windows Store apps, components, or class libraries.

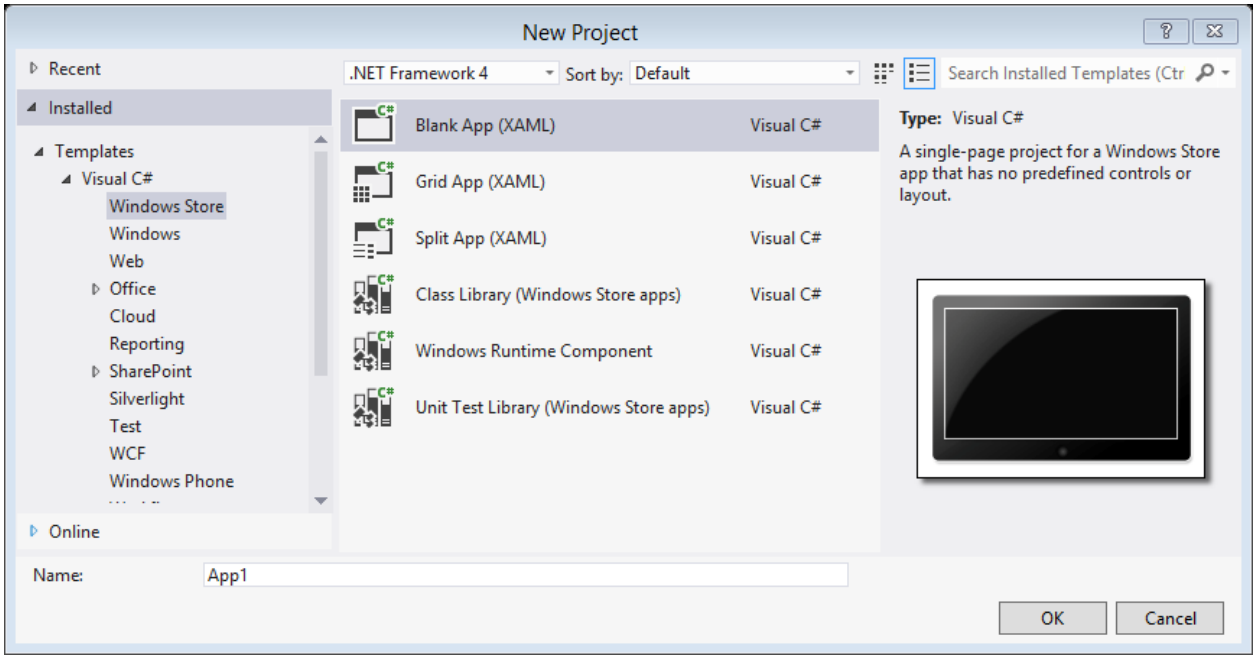


Figure 3: The Visual Studio Project Selection Dialog

There is a subtle but important difference between the class library and Windows Runtime component projects. The class library project can only be consumed by other .NET Windows Store app projects, including other class library projects. The elements exposed by these libraries can expose WinRT types as well as .NET types included within the .NET subset exposed to Windows Store apps (this distinction will be covered later). Windows Runtime component projects can be consumed by any Windows Store app project, including XAML/C++ and HTML/JavaScript projects. As a result, the elements they can expose are restricted to valid WinRT types and conventions.

There is technically another project type that can be developed for and consumed by XAML/.NET Windows Store app projects—the Portable Class Library. This project allows creating class libraries that can be consumed by multiple .NET framework variants, including .NET 4 and 4.5, Silverlight 4 and 5, Windows Phone 7 and 7.5, and Xbox 360. Details of this library type are outside the scope of this book, but more information can be found at http://msdn.microsoft.com/en-us/library/windows/apps/gg597391.aspx.

Tip: Microsoft has published a vast set of code samples for Windows Store apps (along with various other code samples) in the MSDN Code Gallery, available at http://code.msdn.microsoft.com. Furthermore, support for searching, downloading, and installing these samples has been built directly into Visual Studio's New Project dialog. In this dialog, under the Online node, there is now a Samples entry. The collection of available samples can be browsed by selecting values under this node, or the contents can be searched by using the Search Installed Samples text box in the upper right-hand corner of the dialog. Selecting a sample template will download the template and open a new Visual Studio project that includes the contents of that sample. Additional information about obtaining online samples can be found at http://msdn.microsoft.com/en-us/library/jj157272.aspx.

The .NET for Windows Store Apps Framework Profile

The .NET Framework is exposed via a profile that is specific to Windows Store apps, known as ".NET for Windows Store apps." Much like Silverlight applications have access to a reduced set of available .NET types and members, a similar paring of functionality occurs with the .NET for Windows Store apps profile. This reduction serves two purposes: First, .NET types that overlap WinRT types have been removed to prevent duplication. Second, types and members that would otherwise provide functionality outside of the controlled runtime environment provided by Windows for Windows Store apps have also been removed so as to maintain this sandboxed environment and minimize any potential confusion about their availability.

Tip: The provided Windows Runtime types can often be distinguished from the provided .NET types based on their namespaces. WinRT types are usually located in namespaces that start with "Windows," such as `Windows.Storage`, whereas .NET types are often located in namespaces that start with "System," like `System.IO`.

A list of supported .NET APIs supported in the .NET for Windows Store apps profile can be found at http://msdn.microsoft.com/en-us/library/windows/apps/br230232.aspx.

.NET Framework Tools for Asynchronous Programming

An important new addition to the .NET Framework that is included in the .NET for Windows Store apps profile is support for the new **`async`** and **`await`** keywords, which work with the compiler to greatly simplify writing asynchronous code. In previous incarnations of .NET, it could be tedious to write asynchronous code. The latest popular technique made use of lambda functions and closures to simplify things, but it was still an awkward process that involved hooking the completion callback, providing the completion code in a lambda, and then calling the method that invokes the callback. In rough English, your code would read, "This is what to do when you've done your next task. Now do your next task." This technique could be complex and hard to understand when multiple asynchronous steps were being chained together. The following code illustrates this technique when retrieving the markup for the Syncfusion website:

```
private void FetchSomething()
{
    var client = new WebClient();
    client.DownloadStringCompleted += (sender, args) =>
    {
        var result = args.Result;
        // Run your code.
    };
    client.DownloadStringAsync(new Uri("http://www.syncfusion.com"));
}
```

With the **async** and **await** keywords, the natural flow of your code is preserved, and the compiler takes on the task of handling the convoluted details of sequencing the code. The following much simpler code makes a similar web call:

```
private async void FetchSomething()
{
    var httpClient = new HttpClient();
    var result =
            await httpClient.GetStringAsync(new Uri("http://www.syncfusion.com"));
    // Run your code.
}
```

The key differences start with the use of the **await** keyword preceding the **GetStringAsync** call, which instructs the compiler to set things up so that the code following the “Run your code” comment does not execute until the **GetStringAsync** call returns, while at the same time allowing the calling thread to continue executing. The second difference is that when a method contains an **await** call, that method’s declaration must specify the **async** keyword.

The emphasis on asynchronous calls in the WinRT API will lead to frequent use of **async** and **await** in most Windows Store app code, including the samples throughout this book, so additional context can be found throughout later chapters. Additional information on asynchronous programming with **async** and **await** can be found at http://msdn.microsoft.com/en-us/library/hh191443.aspx.

Lining Up the WinRT API and the .NET Framework

While the WinRT members generally play nicely with their .NET framework counterparts, there are a few places where the alignment isn’t quite perfect. In these cases, new objects, new methods, or helpers are sometimes provided to bridge some of the gaps. A comprehensive review of all of these is beyond the scope of this book, but there is an important set of extension methods that is worth calling out here, and some others will be mentioned in context within the chapters that follow. For more information, visit http://msdn.microsoft.com/en-us/library/windows/apps/hh694558.aspx.

There are three main sets of extension methods that facilitate conversion between .NET Framework types and WinRT types. These are:

- Streams—**System.IO.WindowsRuntimeStreamExtensions**: Extension methods in this class provide ways for converting between streams in WinRT and manages streams in the .NET Framework.
- Files and folders—**System.IO.WindowsRuntimeStorageExtensions**: Extension methods in this class provide ways for accessing the WinRT file and folder interfaces—**IStorageFile** and **IStorageFolder**, respectively—through .NET streams.
- Buffers/byte arrays—**System.Runtime.InteropServices.WindowsRuntime.WindowsRuntimeBufferExtensions**: Extension methods in this class provide ways for moving between .NET byte arrays and the contents of WinRT buffers, exposed as **IBuffer** implementations.

Creating WinRT Components with .NET

There was a special project type mentioned in the list of available Visual Studio project types: the Windows Runtime component project. As mentioned, this project allows components to be created using .NET that can be referenced from other Windows Store app development languages, including C++ and JavaScript. Because the component that is created is actually a WinRT component, there are some restrictions that need to be followed—these restrictions are similar in concept to the idea of creating CLS-compliant .NET assemblies.

The guidelines that custom WinRT components must follow apply only to outward-facing (public) members, and include:

- Fields, parameters, and return values must be WinRT types, except for certain types that projections are provided for:
    - **System.Int32**, **System.Int64**, **System.Single**, **System.Double**, **System.Boolean**, **System.String**, **System.Enum**, **System.UInt32**, **System.UInt64**, **System.Guid**, **System.Byte**, **System.Charm,** **System.Object**.
    - **System.Uri**.
    - **System.DateTimeOffset**.
    - Collection interfaces such as **IEnumerable<T>**, **IList<T>**, **IReadOnlyList<T>**, **IDictionary<TKey, TValue>**, **IReadOnlyDictionary<TKey, TValue>**, **KeyValuePair<TKey, TValue>**, **IEnumerable**, **IList**.
    - Property change-related types, including **INotifyPropertyChanged**, **PropertyChangedEventHandler**, **and PropertyChangedEventArgs**.
- Classes and interfaces:
    - Cannot be generic.
    - Cannot implement a non-WinRT interface
    - Cannot override object methods other than **ToString()**.
    - Cannot derive from types not in WinRT (like **SystemException** and **System.EventArgs**).
- Types must have a root namespace that matches the assembly name, and the assembly name may not begin with “Windows.”

- Structs cannot only have public fields, which must be value types or strings.
- Classes must be sealed.

Note: WinRT types only support absolute URIs, whereas .NET types can support both relative and absolute URIs. When the System.Uri is used to provide values to WinRT components, care must be taken to ensure that the System.Uri object only contains absolute URIs.

Additionally, while WinRT types support method overloading, JavaScript's weakly typed and type coercion nature will cause it to struggle to determine which method instance to call in situations where a method has overloads with the same number of parameters. In such a case, one of the overloads must be decorated with the **`Windows.Foundation.Metadata.DefaultOverloadAttribute`**, and JavaScript code will only be able to access that particular overload. This is illustrated in the following code sample, where JavaScript code will only be able to call the version of **`DoSomething`** that takes an **`Int32`** as a parameter:

```
[DefaultOverload]
public void DoSomething(Int32 value)
{
}

public void DoSomething(String value)
{
}
```

Whereas .NET handles asynchronous operations using the **`Task`** or **`Task<T>`** type, WinRT uses several interfaces: **`IAsyncAction`**, **`IAsyncActionWithProgress<TProgress>`**, **`IAsyncOperation<TResult>`**, or **`IAsyncOperationWithProgress<TResult, TProgress>`**. There are two extension methods, **`WindowsRuntimeSystemExtensions.AsAsyncAction`** and **`WindowsRuntimeSystemExtensions.AsAsyncOperation<T>`**, that allow a **`Task`** to be returned as the equivalent **`IAsyncXXX`** interfaces that do not involve reporting progress. To report progress, additional steps need to be taken involving the WinRT **`AsyncInfo`** class.

Additional information about the restrictions for WinRT components and the mappings that occur between the .NET types and their WinRT counterparts can be found at http://msdn.microsoft.com/en-us/library/windows/apps/br230301.aspx.

Creating a Simple Windows Store App

Up to this point, a lot of important concepts for developing Windows Store apps with .NET have been presented. It seems to be a good opportunity to close out this introductory chapter by creating an application in the spirit of the ubiquitous "Hello World" application. This will allow an opportunity to touch on some final concepts and show some of the already-presented concepts in action before moving on to content that is more specifically targeted.

To get started, launch Visual Studio 2012 (hereafter just “Visual Studio”) and select **New Project** either from the start page or from the **File** menu. In the **New Project** dialog, select the **Visual C# template** group and the **Windows Store** group under it. Select the **Blank App (XAML)** project template, and enter a name for the project or simply accept the default AppX name that is provided. Congratulations! You have just created a Windows Store app.

Project Anatomy

Once Visual Studio finishes spinning up the project, the **Solution Explorer** should contain the following:

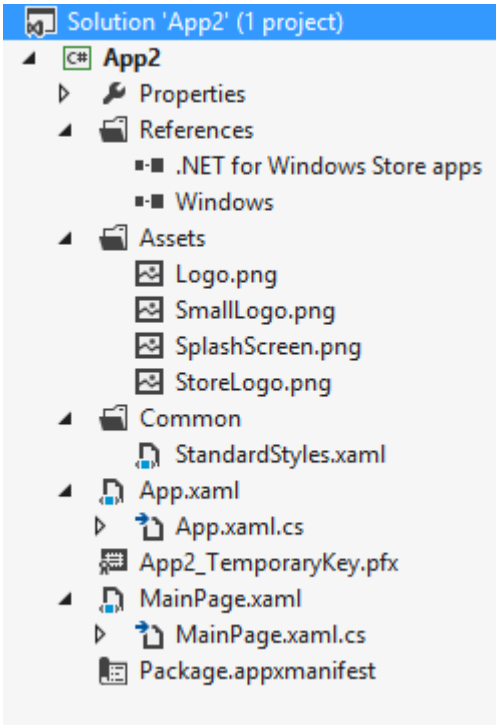


Figure 4: A Windows Store Application Project

The contents of the project shown in Figure 4 include:

- **App.xaml** and **App.xaml.cs**: These files define the class that represents the application object which governs the lifetime of a Windows Store app. Among other things, the implementation of this class contains any resources to be made available throughout the entire application (the XAML resource system will be discussed in the next chapter), as well as any application lifetime event handlers.
- **MainPage.xaml** and **MainPage.xaml.cs**: This is a user interface page for the application. It is currently blank.
- **StandardStyles.xaml**: Provides a standard set of preconfigured styles and other resources that can be used to apply the Windows Store user interface look and feel to your user interface elements.
- **App2_TemporaryKey.pfx**: A preconfigured test certificate that is used to digitally sign your app.

- **Package.appmanifest**: This is known as the application manifest configuration file. It contains configuration settings that govern how the Windows Store app integrates with Windows.

Other items in the project include assembly references to the .NET for Windows Store apps Framework profile and the Windows Runtime, and several image files that are used for application configuration.

The Application Manifest Configuration File

As indicated previously, the Application Manifest Configuration File contains the settings that govern how your application will integrate with Windows. The file itself is simply an XML settings file, and Visual Studio provides an interactive user interface for making changes to this file. During the build process, this configuration file is copied in order to produce the Application Manifest file.

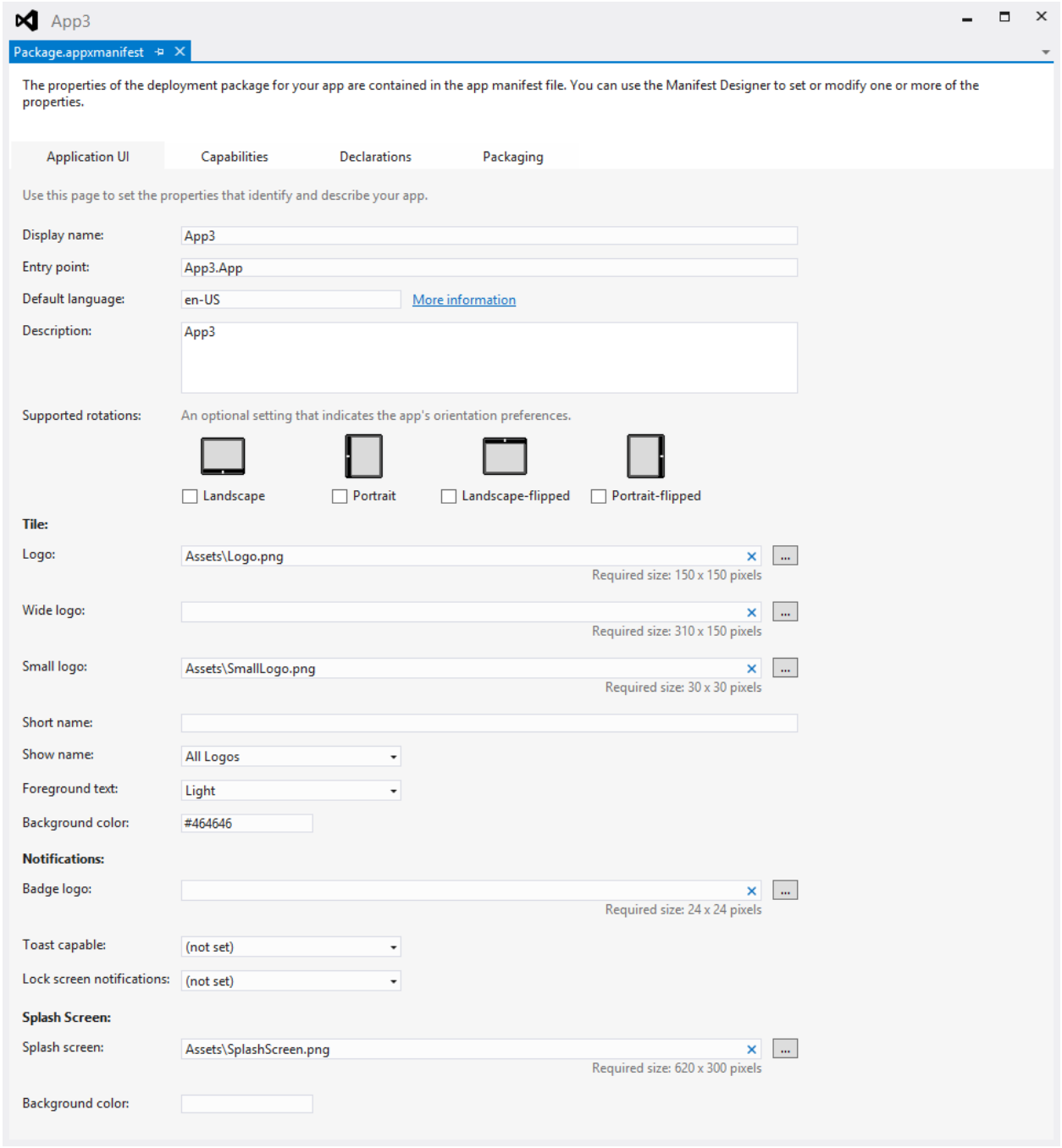


Figure 5: App Manifest Configuration

Visual Studio divides the manifest into four sections: Application UI, Capabilities, Declarations, and Packaging. The Application UI page contains settings that describe the app when it is deployed, including the text and images to use for the application's tiles and other icon displays, the splash screen content to display when launched, and the screen orientations the application will support.

The Capabilities page indicates system resources that your app intends to access, such as removable storage, webcam, or Internet, to name a few. Failing to declare a capability in the manifest and then trying to access that resource from code will result in an exception. Declaring more capabilities than an app actually needs can result in the app failing the marketplace certification step. When customers download an app from the store, they are first informed of the capabilities that the app has declared. Certain capabilities—specifically enterprise authentication, shared user certificates, and documents library—can only be set for applications that are published by a company account, as opposed to an individual developer account. Capabilities are described in detail at http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx.

The Declarations page configures settings for the system extensibility points—known as contracts and extensions—which the application elects to interact with. The available declarations are described at http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx, and the subjects of contracts and extensions will be discussed more fully later in this book.

The Packaging page allows setting properties that affect the deployment of the project package. This includes the package name which identifies the package on the system. This value will be replaced when the package is uploaded to the Windows Store. The **Package Display Name** property provides a user-friendly name that is displayed in the Store. This value is also replaced when the app package is uploaded to the Store. **Logo** specifies the image to use in the Windows Store description page. **Version** indicates the version of the app that will be used and displayed. The **Publisher** field allows configuring the digital certificate that is used to authenticate the package. This is also replaced when the app is uploaded to the store. The **Publisher Name** specifies the value used for that display in the store, and is another field that is updated when the app is uploaded to the store. Finally, **Package Family Name** is read-only, calculated from the package name and publisher, and is used to identify the package on the system.

Saying “Hello”

It is time to add some content to the simple “Hello” application. This application simply collects some information from users about the characteristics of a random “word” to be requested from the API provided at http://www.random.org. Because of the HTTP calls being made, this simple scenario provides an opportunity to use asynchronous calls within a Windows Store app. The first step will be to add some user interface elements. Adding the following markup between the **Page** elements of the MainPage.xaml file will provide a title and the relevant controls:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <!-- Page title. -->
    <Grid Grid.Row="0" Margin="120,0,0,0">
        <TextBlock Grid.Column="1"
                Text="Hello Windows Store apps Succinctly"
                Style="{StaticResource PageHeaderTextStyle}"/>
    </Grid>
```

```
    <!-- Page content. -->
    <Grid Grid.Row="1" Margin="120,0,120,50">
        <Grid.Resources>
            <Style TargetType="TextBlock" BasedOn="{StaticResource BasicTextStyle}">
                <Setter Property="VerticalAlignment" Value="Center"/>
                <Setter Property="Margin" Value="0,0,10,0"/>
            </Style>
        </Grid.Resources>
        <StackPanel>
            <!-- Specify the length of the word to return. -->
            <TextBlock Text="Word Length"/>
            <ComboBox x:Name="LengthSelection" SelectedIndex="0"
                      Width="150" HorizontalAlignment="Left">
                <ComboBox.Items>
                    <x:String>5</x:String>
                    <x:String>6</x:String>
                    <x:String>7</x:String>
                    <x:String>8</x:String>
                </ComboBox.Items>
            </ComboBox>

            <!-- Indicate if the word should contain digits. -->
            <ToggleSwitch x:Name="IncludeDigits" Header="Include Digits"
                          OnContent="Yes" OffContent="No" IsOn="False"/>

            <!-- Indicate the allowable letter cases for the word. -->
            <TextBlock Text="Letter Case"/>
            <StackPanel Margin="20,10,0,0">
                <RadioButton x:Name="MixedCase" Content="Mixed Case Letters"
                             IsChecked="True"/>
                <RadioButton x:Name="UpperCaseOnly" Content="Upper Case Letters Only"/>
                <RadioButton x:Name="LowerCaseOnly" Content="Lower Case Letters Only"/>
            </StackPanel>

            <!-- Button to trigger the fetch. -->
            <Button Content="Go" Click="GoButton_Click" Width="250" Margin="0,10,0,0"/>

            <!-- Display the results.-->
            <TextBlock x:Name="RandomWordText"
                       Style="{StaticResource SubheaderTextStyle}" />
        </StackPanel>
    </Grid>
</Grid>
```

Next, a handler is provided for when users click the **Go** button. In this case, the request URL is built based on the user input and the UI is updated with the results from the call. It is important to note that the call to make the request is asynchronous, requiring the use of the previously mentioned **`async`** and **`await`** keywords. This code is placed in the MainPage.xaml.cs file.

```
private async void GoButton_Click(object sender, RoutedEventArgs e)
{
    // Request a random set of letters from http://www.random.org.
    // For details, see http://www.random.org/clients/http/.

    // 1 string result.
    // Variable character length based on UI input.
```

```
    // Variable includes digits based on UI input.
    // Variable upper and lower alpha based on UI input.
    // Request unique results.
    // Request results as plain text.
    // Request should initiate a new randomization.

    // Build the Uri from the inputs in the UI.
    var builder = new UriBuilder("http://www.random.org/strings/");
    var lengthArg = String.Format("len={0}", LengthSelection.SelectedItem);
    var digitsArg = String.Format("digits={0}", IncludeDigits.IsOn ? "on" : "off");
    var includeUpper = UpperCaseOnly.IsChecked.Value || MixedCase.IsChecked.Value;
    var upperAlphaArg = String.Format("upperalpha={0}", includeUpper ? "on" : "off");
    var includeLower = LowerCaseOnly.IsChecked.Value || MixedCase.IsChecked.Value;
    var lowerAlphaArg = String.Format("loweralpha={0}", includeLower ? "on" : "off");
    var queryString =
        String.Format("num=1&{0}&{1}&{2}&{3}&unique=on&format=plain&rnd=new",
        lengthArg,
        digitsArg,
        upperAlphaArg,
        lowerAlphaArg);
    builder.Query = queryString;

    // Make the Http request.
    var httpClient = new HttpClient();
    var results = await httpClient.GetStringAsync(builder.Uri);

    // Split the first result off from the (potential) list of results.
    var resultWord = results
        .Split(new[]{'\n'}, StringSplitOptions.RemoveEmptyEntries)
        .First();

    // Report the word in the UI.
    RandomWordText.Text = String.Format("Your random word is: {0}", resultWord);
}
```

This code will require a **`using`** statement for the **`System.Net.Http`** namespace.

Note: This is a very simple, contrived example, so much of the error checking and UI niceties that would be part of even the simplest real-world browser apps are being omitted for the sake of brevity.

Running the Code

At this point, the app can be built, deployed, and launched using the facilities provided by Visual Studio. Selecting **Start Debugging** from the **Debug** menu will launch the app with the debugger attached. Visual Studio also provides a special simulator environment that enables the app to run within a special window that simulates a tablet. The simulator provides options for:

- Using a mouse to simulate touch interactions.
- Simulating device rotation.

- Selecting different device resolutions.
- Providing simulated location information to the app's GPS sensor.
- Taking screenshots.

The simulator is an especially valuable debugging tool since it runs in a window, whereas a Windows Store app would otherwise run full-screen, forcing you to toggle between Visual Studio and the running app in single-monitor environments. Furthermore, it enables enhanced testing of machine facilities—such as touch or orientation—that may not be readily available or may otherwise be inconvenient on a traditional development machine. To run an app within the simulator, the simulator option can be selected from the **Target Device** drop-down list in the standard Visual Studio toolbar.

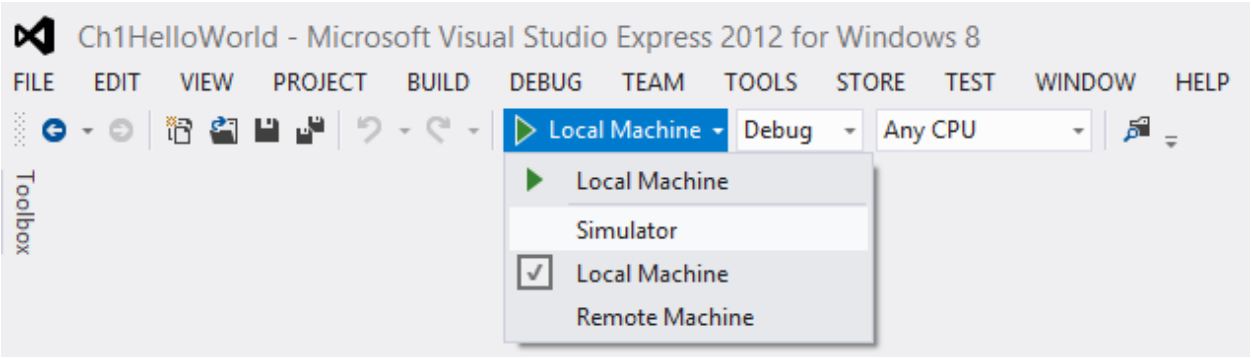


Figure 6: Selecting "Simulator" from the Visual Studio Debug Target Options

Note: The simulator is not actually running in a virtual machine or another isolated environment like other systems' simulators, including the emulator provided in the Windows Phone 7 SDKs. The Visual Studio simulator actually runs as a remote desktop connection back to the current machine.

Debugging a Windows Store app remotely on another Windows 8 machine is also an available option, though it will not be covered in this book. For more information on the remote machine option, see http://msdn.microsoft.com/en-us/library/windows/apps/hh441469.aspx.

Running the "Hello World" app within the simulator should resemble the following:

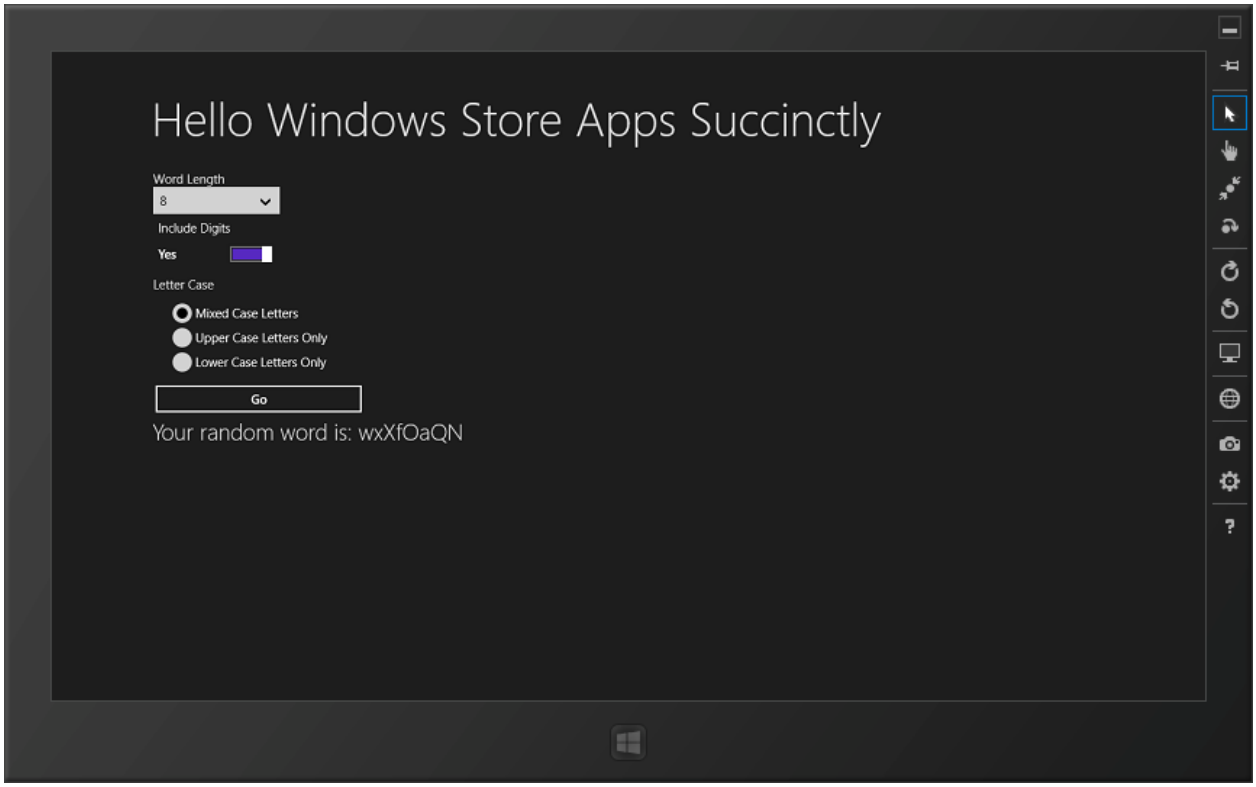


Figure 7: The Hello World App Running in the Simulator

Recap

This chapter presented several of the core concepts underlying development for Windows Store apps. This included an overview of Windows Store apps and a discussion of the new Windows Runtime. The fundamentals of using Visual Studio to develop a Windows Store app with .NET were discussed, and a sample “Hello World” app was created and run within the Visual Studio simulator.

The rest of this book will focus on several key areas that are important to understand for developing a Windows Store app. These include:

- The use of XAML for user interface development, including several new controls specific to Windows Store app development, as well as the model for navigating between pages in Windows Store apps.
- Concepts related to the special techniques and requirements related to managing the lifetime of a Windows Store app along with options for storing state and other application information.
- Contracts and extensions, which are mechanisms that allow an app to integrate with Windows facilities that provide additional ways the app can be used.
- Facilities available for providing user feedback through live tile updates, toast notifications, and push notifications.

- Several ways Windows Store apps can incorporate integration with system hardware such as sensors, geographic positioning information, and interaction with cameras and microphones.
- Concepts related to the deployment of a Windows Store app to the Windows Store itself—including trial modes, in-app purchases, using advertising frameworks, and also mechanisms for deploying apps without using the Windows Store, which is particularly valuable for line-of-business enterprise apps.

Chapter 2 XAML, Controls, and Pages

When WPF was first released, one of the technologies it featured included a new XML-based language for declaratively specifying user interfaces, called the Extensible Application Markup Language (XAML, pronounced *zammel)*. At its core, XAML is a mechanism for declaratively defining and setting properties inside hierarchical object graphs. Although its main use to date has been for user interface layout—with several "dialects" for WPF, Silverlight, and Windows Phone 7—it has also seen other uses, including being at the core of XML Paper Specification (XPS) for documents and being used for the design of object graphs used in Windows Workflow Foundation.

For Windows Store apps created with .NET, XAML continues to be the primary mechanism for user interface layout and design. As with WPF and Silverlight, there are specific elements that are different with the XAML dialect used for Windows Store apps. However, understanding XAML for user interface design in any of the previously mentioned platforms will provide a solid foundation for how to go about using XAML to build a Windows Store app.

This chapter will initially provide a high-level look at some foundational XAML concepts and their application to laying out Windows Store apps. Along the way, it will introduce several of the new user interface concepts and elements that have been introduced for Windows Store apps. Finally, it will conclude with a discussion of the Page control, which will contain the other XAML controls within a Windows Store app, and related mechanisms that support navigation and layout orientations.

Declaring User Interfaces with XAML

As a platform for user interface design, XAML and several related technologies combine to enable rapid development of sophisticated user experiences. One of the primary features of XAML-based user interfaces is a separation between an application's user interface layout and its behavior, with the layout of the UI elements declared in the XAML markup and the behavior exhibited by those UI elements defined in .NET code, tied together through code-behind files and other mechanisms. As mentioned in the previous chapter, the main tools used for building XAML user interfaces include Visual Studio and Expression Blend. Visual Studio will be familiar to most Windows application developers, whereas Expression Blend is targeted more toward visual and graphic designers; however, many developers use both Visual Studio and Expression Blend in tandem to design and structure their user interfaces, taking advantage of the strengths of both IDEs. Although initially the layout engines used by these tools were distinct, several of the visual design tools used for XAML editing that are in Visual Studio 2012 have actually been brought over from Expression Blend.

Tip: Experienced developers who have access to multi-monitor environments often have a project open on one monitor in Visual Studio and the same project also opened in an adjacent monitor within Expression Blend, simultaneously taking advantage of the strengths of both IDEs. In fact, right-clicking on a XAML file in Visual Studio displays a context menu that includes a command to open the file directly in Expression Blend, and

Blend includes a context menu option to open files in Visual Studio. It is important when doing this to remember to save content when moving between applications, since the unsaved edits are not automatically kept in sync, though both IDEs detect changes made to any open files and will prompt to load in a new version of the file when the other application has made and saved some modifications.

The following markup shows a bare-bones page that is created when a blank **Page** element is added to a Visual Studio project:

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

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>
```

Just this small snippet of XAML provides a starting point for discussing several of the fundamental concepts underlying XAML-based UI development.

Class and Namespace Specifications

At its most basic, the XAML sample instructs Windows to create a top-level **Page** control. The **x:Class** identifier indicates that the specific **Page** subtype that is being created should be an instance of the **DemoBlankPage** class, defined in the **WindowsStoreAppsSuccintly** .NET namespace (this class is referred to as the "code-behind class"). At design time, a "semi-hidden" partial class file named DemoBlankPage.g.i.cs is created from the XAML that includes corresponding fields for any XAML elements that are identified with the **x:Name** property. An implementation of the **InitializeComponent** method is also created. It is called in the class' constructor and is responsible for loading and parsing the XAML markup file at run time, creating instances of the desired objects, and setting the values of the fields mentioned previously to the actual corresponding UI elements.

Note: While the previous description of what happens with the `x:Class` attribute and the code-behind class file may sound complex, it is usually a process that is fairly invisible to developers. Having a high-level awareness of what is going on here is helpful for the occasional circumstance when something goes wrong in this connection, which is usually caused by either the code-behind class being renamed or moved into a new namespace without the `x:Class` declaration also being updated (resulting in a compile-time error), or some bad markup failing to be properly parsed at run time during the call to `InitializeComponent`.

Along with the call to **`InitializeComponent`**, the code-behind class will contain the .NET code that defines the XAML control's overall behavior. In addition to being able to access any elements identified with the **`x:Name`** property in the XAML via the fields that are automatically created, any event handlers that are established declaratively within the markup will refer to corresponding methods within this type.

Following the **`x:Class`** property, the markup also includes the declaration of several namespaces. Namespace declarations are specified with **`xmlns`** identifiers and are used by XML to help provide scope for the content contained within a document. In the case of XAML, they provide information about where the UI elements defined in markup originate, and sometimes disambiguate similar classes that are defined in different .NET namespaces, similar to how alias declarations are done with the **`using`** keyword in C# code. The markup sample includes the root namespace—the one that has the **`xmlns`** declaration not followed by a "**`:alias`**" term—that applies to the core XAML controls, and the **`x`** namespace which identifies various XAML utility features. These namespaces will be in every XAML document. Additionally, it includes the **`d`** and **`mc`** utility namespaces primarily used in Windows Store apps to identify items that are only interpreted at design time, most often to provide access to design-time data. This data can be used to visualize XAML elements in the IDEs with simulated or actual application data. The final namespace to mention is the **`local`** namespace, which is an instance of a custom namespace declaration. XAML files can use multiple namespace declarations to provide scope for internal and third-party controls that originate in various .NET namespaces. Declaring a custom namespace allows the object in question to be included in the XAML document by qualifying it with its namespace alias. The following markup example shows how a custom namespace declaration is used to reference a third-party control—in this case, the **`TileView`** control from Syncfusion's Essential Studio for WinRT control suite.

```
<!-- Custom namespace declaration/alias. -->
xmlns:syncfusion="using:Syncfusion.UI.Xaml.Controls.Layout"

<!-- XAML element qualified using a custom namespace alias. -->
<syncfusion:TileView>
    <!-- Content omitted for brevity. -->
</syncfusion:TileView>
```

Note: It is important for WPF, Silverlight, and Windows Phone developers to note that the syntax used for custom namespace declarations has been changed and simplified in the XAML used for Windows Store apps. For Windows Store apps, the syntax follows the convention *`xmlns:alias="using:.NET-namespace"`* ***, as opposed to the older*** *`xmlns:alias="clr-namespace:namespace;assembly=assembly"`* ***syntax (e.g.,*** *`xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"`* ***). This difference in syntax is the first of many reasons why sharing XAML markup without modification between Windows Store apps and other application types is nearly impossible.***

Resource Dictionaries and Resource References

Following the initial **Page** element declaration, the markup then specifies the addition of a **Grid** element. The **Grid** is a powerful control used in XAML-based UIs to provide row-based and column-based layout, and will be discussed in more detail shortly along with several other related controls. Within the **Grid** declaration, its **Background** property is specified using a specialized syntax known as a "markup extension." Markup extensions provide extensions to XAML and can be spotted by their use of braces within quotes. In this case, the markup extension element refers to a **StaticResource** element and uses the resource system included in XAML-based UIs to set the grid's background to use the **ApplicationPageBackgroundThemeBrush**—a system-defined resource to set a standard color for the background of a page based on the currently selected desktop theme.

One of the core base classes inherited by items that are to be included in XAML-based UI layouts is the **FrameworkElement** class. Any element that inherits from the **FrameworkElement** class exposes a **Resources** property which returns a **ResourceDictionary** reference, as does the app's root **Application** object. A resource entry in a resource dictionary is simply an object instance along with the key that designates the resource's name. While most often the key is specified using the **x:Key** attribute, for the cases of implicit styles and control templates, there is a **TargetType** specification that serves as a surrogate key (implicit styles and control templates will be discussed shortly). Resources can be added through XAML property syntax, or they can also be added programmatically. The following example shows several different kinds of resources added to the previously shown grid's resource collection. These include a color definition, a **Brush** that can be used to draw user interface elements with the previously defined color, an implicit style that applies to text elements and sets their foreground color to use that brush, and a **String** definition.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.Resources>
        <!-- Define a color called “ForegroundColor”. -->
        <Color x:Key="ForegroundColor">#1BA1E2</Color>

        <!-- Define a brush element called “ForegroundBrush”. -->
        <SolidColorBrush x:Key="ForegroundBrush"
                                    Color="{StaticResource ForegroundColor}"/>

        <!-- Define an implicit style resource to be applied to TextBlock elements. -->
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="32"/>
            <Setter Property="Foreground" Value="{StaticResource ForegroundBrush}"/>
        </Style>

        <!-- Define a String resource. -->
        <x:String x:Key="Sample">Hello World</x:String>
     </Grid.Resources>
    <TextBlock Text="{StaticResource Sample}"/>
</Grid>
```

Once resources have been defined, the process that XAML uses for looking up their values is recursive, so when a resource is referenced from within a XAML element, the resource management system searches through the item's parent elements until the first match is found within an element's resource collection. This traversal will also include the **`Application`** object's resources, as well as a special collection of platform-defined resources. The use of the phrase "first match" is deliberate—resources defined at a higher level in the hierarchy can be overridden at lower levels, allowing for customization. As can be seen in the previous example, resource lookup in XAML occurs through the **`StaticResource`** markup extension. Additionally, resources can be retrieved programmatically using the indexer on any given **`ResourceDictionary`** property; however, this programmatic lookup only includes the current item. It does not use the same parent traversal that the **`StaticResource`** markup extension does.

In addition to the resources defined in the locations described, XAML also allows for the definition and inclusion of stand-alone resource dictionary files which contain collections of defined resources. These dictionaries can be merged with an existing **`ResourceDictionary`** via **`MergedDictionary`** elements. Windows Store app projects created using Visual Studio templates other than the Blank App template include a StandardStyles.xaml resource dictionary file that defines dozens of layout-related resources for use in Windows Store apps. This file is brought in as a **`Merged Dictionary`** in the **App.xaml** file:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <!--
                Styles that define common aspects of the platform look and feel
                required by Visual Studio project and item templates.
             -->
            <ResourceDictionary Source="Common/StandardStyles.xaml"/>
        </ResourceDictionary.MergedDictionaries>
        <!-- Application-specific resources. -->
        <x:String x:Key="AppName">Windows Store apps Succinctly</x:String>
    </ResourceDictionary>
</Application.Resources>
```

Tip: Expression Blend includes a Resources panel (typically a tab on the right side of the IDE, adjacent to the Properties tab) that shows visual representations for the resources defined in a XAML project. Both Visual Studio and Blend also include the ability to assign a resource value to a property through the GUI by selecting the small square adjacent to values in the property panels that support resource values and selecting Local Resource for a list of applicable locally defined resources or System Resource to see the applicable platform resources. Furthermore, locally selected properties can be "promoted" to resources by selecting the Convert to New Resource menu option.

Properties and Events

Within XAML elements, attributes are used in the XML to set the properties of the declared objects. The value that is set in the XML is simply assigned to the target property. In the likely event that the target property is not actually a **`String`** type (for example, a number value to specify a size or a member of an enumeration), the XAML parser works with some helper objects called type converters to convert the text value declared in the markup to the appropriate type. For properties that cannot be converted or that otherwise need to be set to more complex object values, XAML provides a property-element syntax that allows a nested XML element to be used as the value for a property. This syntax can be used by nesting the value to be assigned within an additional XML element with the form **`ParentType.PropertyName`**. The following code shows the **`Background`** property of a **`Grid`** element both with simple and complex values. In the first case, the **`"Blue"`** value is implicitly converted into a **`SolidColorBrush`** with its **`Color`** value set to the **`Blue`** color member of the **`Windows.UI.Colors`** enumeration. In the second case, the property is set to a **`LinearGradientBrush`** which defines a gradual color shift between the specified child values, and within that brush, the **`GradientStops`** collection is set to a set of discrete **`GradientStop`** values:

```
<!-- Grid with a simple property setter. -->
<Grid Background="Blue"/>

<!-- Grid with a complex property setter using property-element syntax.-->
<Grid>
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Offset="0" Color="Orange"/>
                <GradientStop Offset="1" Color="Blue"/>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

In addition to setting property values, XAML can be used to attach event handlers to the objects that are declared in the markup. To create such a connection, a handler method with the correct parameter structure needs to be defined in the related code-behind class. That method's name is assigned to the desired event name in the markup in the same way simple property assignments are made. The following code sets the **`Button_Click_1`** method to handle the **`Click`** event on a **`Button`** control:

```
<!-- Button with a simple property setter and a listener for the Click event. -->
<Button Content="Click Me" Click="Button_Click_1"/>

// The related event handler in the code-behind file.
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    // Code omitted for brevity.
}
```

Note: In this example, the target function was automatically created by Visual Studio as a result of typing the event name followed by an equals sign and selecting the <New Event Handler> context menu entry that appeared in the Visual Studio XAML editor. Alternatively, double-clicking the text box next to an event name in the event listing of the properties panel can be used to also automatically generate an event handler method that is connected to the event within the markup.

Dependency Properties and Attached Properties

Almost every element that can be used to include UI elements in the XAML markup ultimately inherits from the **`DependencyObject`** class. This class is at the core of a special property framework that supports several of the advanced layout and interactivity features available in XAML-based UIs. Properties defined using this framework are known as dependency properties. Dependency properties can be read and set like ordinary properties, but they also bring several important features along with their implementation, including:

- Built-in property change notification, which allows these properties to participate in data binding, which will be discussed shortly.
- Hierarchical value resolution, which allows these properties to internally hold a hierarchy of values at any given time, with a set of precedence rules used to determine which one is to be returned.
- The ability to set default values and change callback functions to be used by the property.

The first step involved in declaring a dependency property is the creation of a static **`DependencyProperty`** object that includes the configuration information for the property and registers the new property with the dependency property system. This declaration can also provide an optional default value for the property and an optional callback function to be called when the property's value is changed. Once the static **`DependencyProperty`** object is defined, a regular property can be declared that uses the **`DependencyProperty`** as its backing store by using the **`DependencyObject GetValue`** and **`SetValue`** methods. The following code shows a dependency property called **`SampleDependencyProperty`** being registered and exposed as an instance property:

```
// Using a DependencyProperty as the backing store for SampleDependencyProperty.
// This enables animation, styling, binding, etc...
public static readonly DependencyProperty SampleDependencyPropertyProperty =
    DependencyProperty.Register(
    "SampleDependencyProperty",
    typeof(Int32),
    typeof(DemoBlankPage),
    new PropertyMetadata(0, changeCallback));

// The public property backed by a value registered in the dependency property system.
public Int32 SampleDependencyProperty
{
```

```
    get { return (Int32)GetValue(SampleDependencyPropertyProperty); }
    set { SetValue(SampleDependencyPropertyProperty, value); }
}
```

Note: It is important to avoid including any additional code in the public property getter and setter. In several circumstances, .NET bypasses this particular property declaration and works directly with the dependency property that was registered, so any special logic included will be skipped. If special logic needs to be included in the property set calculation, the property change callback value should be provided when the dependency property is defined and registered.

There is a specialization of the standard dependency properties that can be defined, known as an attached property. Attached properties allow one class to set property values on a property that is actually defined in a different class, effectively "attaching" an externally-defined property to the class. Examples of an attached property are the **`Grid.Row`** and **`Grid.Column`** properties that can be set on an object contained within a grid to indicate where it should be situated within the grid, as shown in the following code:

```
<Grid>
    <!-- Element with the grid row and column attached properties set.-->
    <TextBlock Grid.Row="0" Grid.Column="0" Text="Hello World"/>
</Grid>
```

Note that in this case the **`TextBlock`** element has some values set as to where it should be positioned within its parent grid, but this has been accomplished without explicitly adding grid-specific properties to the **`TextBlock`** type.

Attached properties are defined in a manner very similar to how dependency properties are defined, except that the **`RegisterAttached`** method is used instead of the **`Register`** method. Also, it is customary to include static methods to facilitate setting and retrieving attached property values from a supplied **`DependencyObject`**.

```
// Using a DependencyProperty as the backing store for MyAttachedProperty.
// This enables animation, styling, binding, etc...
public static readonly DependencyProperty MyAttachedPropertyProperty =
    DependencyProperty.RegisterAttached(
    "MyAttachedProperty",
    typeof(Int32),
    typeof(DemoClass),
    new PropertyMetadata(0, changeCallback));

public static Int32 GetMyAttachedProperty(DependencyObject obj)
{
    return (Int32)obj.GetValue(MyAttachedPropertyProperty);
}

public static void SetMyAttachedProperty(DependencyObject obj, Int32 value)
```

```
{
    obj.SetValue(MyAttachedPropertyProperty, value);
}
```

Animations

XAML user interfaces also feature first-class support for animations. Animations apply changes to dependency property values over time, and are coordinated in container objects called storyboards. Animations take advantage of the dependency property hierarchical value resolution mechanism mentioned previously. When an animation is applied to an element in the user interface, the end-state value it sets on the targeted dependency property is only applied as long as the animation is running. The property internally retains its original value plus a few other possible values, and when an applied animation is either stopped or removed, the property value hierarchy reverts to returning the pre-animation value. This behavior is especially valuable for updating the application layout when the display view changes, as will be discussed later in this chapter.

There are two basic types of animations: interpolation-based animations and key-frame based animations. Interpolated animations gradually apply changes linearly to the values which the animations affect over the duration of the animation. Key-frame animations identify values at discrete intervals of time within the animation timespan. When the animation arrives at the key-frame target time, the new value is applied without any use of intermediate values. Interpolated animations include the **ColorAnimation**, **DoubleAnimation**, and **PointAnimation** types, which can be applied to colors, double values, and point values respectively. The key-frame animation types include **ColorAnimationUsingKeyFrames**, **DoubleAnimationsUsingKeyFrames**, **PointAnimationUsingKeyFrames**, and the added **ObjectAnimationUsingKeyFrames**. While the first three apply to the same types as their counterpart interpolated animations, the **ObjectAnimationUsingKeyFrames** can be used to apply key-frame animations to objects for which a specialized animation class is not already available.

The main properties set on animations include the property being affected, the duration for the animation, and the target value that the affected property should have when the animation time has elapsed, or in the case of key frames, when the target key-frame time is reached. The following storyboard scales a rectangle to 50% of both its original height and width and gradually turns its contents red:

```
<Storyboard x:Name="DemoStoryboard">
       <DoubleAnimation Duration="0:0:2" To="0.5"
              Storyboard.TargetName="Rectangle"
              Storyboard.TargetProperty=
                     "(UIElement.RenderTransform).(CompositeTransform.ScaleX)"/>
       <DoubleAnimation Duration="0:0:2" To="0.5"
              Storyboard.TargetName="Rectangle"
              Storyboard.TargetProperty=
                     "(UIElement.RenderTransform).(CompositeTransform.ScaleY)"/>
       <ColorAnimation Duration="0:0:2" To="Red"
              Storyboard.TargetName="Rectangle"
              Storyboard.TargetProperty=
                     "(Shape.Fill).(SolidColorBrush.Color)"/>
```

```
</Storyboard>
```

To help provide more natural or interesting transitions during animations, a set of “easing functions” have been provided that can be appended to animations to provide additional effects. Additional information about easing functions as well as other content on applying animations in Windows Store apps can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh452701.aspx.

Note: Manually creating the markup required for most animations can be tedious and error-prone. Fortunately, Expression Blend provides a powerful set of tools for visually exploring, implementing, and managing animations and storyboards.

Once an animation is defined, it can be triggered programmatically by calling the **Storyboard Begin** method. Animations in Windows Store apps are “independent animations.” They run independently of the main UI thread, and therefore should not be negatively affected by work the app is doing on that thread while the animation is running. When the animation reaches its end, it will either hold its value (the default) or stop depending on the value of its **FillBehavior** property. As previously mentioned, when the animation is stopped or removed, the property being animated will revert to the next value in the hierarchy that is maintained by the affected dependency property. An animation can be programmatically stopped by calling the **Storyboard Stop** method. Finally, the **Storyboard** object also exposes a **Completed** event that is raised when the animation is completed.

The following code shows an animation being either started or stopped programmatically in response to a button click event as well as a subscription to the **Storyboard Completed** event:

```
// Locate the storyboard as a resource on the page object.
var storyboard = (Storyboard)this.Resources["DemoStoryboard"];

// Subscribe to the storyboard completed event.
storyboard.Completed += OnStoryboardCompleted;
// Note: Code to only subscribe to the event once omitted for brevity.

// If the storyboard is stopped, start it. Otherwise, stop it.
if (storyboard.GetCurrentState() == ClockState.Stopped)
{
    storyboard.Begin();
}
else
{
    storyboard.Stop();
}
```

Theme Transitions and Animations

In addition to creating and triggering custom-defined animations, Window Store apps can access a built-in animation library that includes a variety of prepackaged animations already in use throughout the Windows user interface. Access to these pre-built animations is provided through either theme transitions or theme animations.

Theme Transitions

When they are defined on controls in a Windows Store app, theme transitions are automatically triggered in response to some UI change, including items being added or removed from the element where the transition is applied, or when child item locations and sizes are updated within that element. The key aspect of theme transitions to remember is that once defined on an item, the animations they incorporate are tied to a specific set of triggers and are automatically applied when these triggering events occur.

There are several elements that work with theme transitions by supplying properties where the transitions can be defined. These include:

- The **`UIElement Transitions`** property. The transitions defined apply to the current control.
- The **`Panel ChildrenTransitions`** property. The transitions defined apply to all of the content items (children) of the panel.
- The **`ItemsControl ItemContainerTransitions`** property. The transitions defined apply to items generated in the Items control.
- The **`ContentControl ContentTransitions`** property. The transitions defined apply to the content contained in the control.
- The **`Popup ChildTransitions`** property. The transitions defined apply to the child element of a Popup control.

These properties accept an instance of the **`TransitionCollection`** class, which itself can contain one or more of the following transition items, depending on the context:

- **`AddDeleteThemeTransition`**: Used in panels when content is added or removed.
- **`ContentThemeTransition`**: Applied when a content control's content changes.
- **`EdgeUIThemeTransition`**: Applied to animate an item's appearance from the edge of the UI.
- **`EntranceThemeTransition`**: Applied when the applicable controls first appear.
- **`PaneThemeTransition`**: Applied to animate a panel's appearance from the edge of the UI.
- **`PopupThemeTransition`**: When items are added to a collection, they pop into view.
- **`ReorderThemeTransition`**: Applied when the items in an Items control change order.
- **`RepositionThemeTransition`**: Applied when items' positions are changed.

In the following XAML markup, a **`StackPanel`** is configured such that when new items are added to its **`Children`** collection, an animation is applied that moves items in from the bottom of the panel.

```
<!-- Animate new content being added by scrolling content from the bottom. -->
<StackPanel x:Name="ThemeTransitionPanel">
    <StackPanel.ChildrenTransitions>
        <TransitionCollection>
            <PopupThemeTransition/>
        </TransitionCollection>
```

```
    </StackPanel.ChildrenTransitions>
</StackPanel>
```

Theme Animations

Theme animations are similar to theme transitions in that they define standard animations, but they do not attach them to any particular triggers. Theme animations need to be included within a **`Storyboard`** where one of the custom animations described previously would otherwise be. Theme animations must also define a **`TargetName`** property to identify the element to which they will be applied, and perhaps additional properties that set the characteristics of the animation. It is up to the app to invoke the animation, either programmatically as shown in the previous sample, or through the visual state manager, which will be discussed next. The theme animations included in the animation library include:

- **`DragItemThemeAnimation`**, **`DragOverThemeAnimation`**, **`DropTargetItemThemeAnimation`**: Used to provide animations for drag and drop events.
- **`FadeInThemeAnimation`**, **`FadeOutThemeAnimation`**: Used to fade items in and out.
- **`PointerDownThemeAnimation`**, **`PointerUpThemeAnimation`**: Used to react to touch and mouse-based pointer interaction with elements
- **`PopInThemeAnimation`**, **`PopOutThemeAnimation`**: Used to make items pop in and pop out of the UI.
- **`RepositionThemeAnimation`**: Used to reposition elements on the screen
- **`SplitOpenThemeAnimation`**, **`SplitCloseThemeAnimation`**: Used to apply a split effect.
- **`SwipeBackThemeAnimation`**: Used to slide items back into position after a swipe interaction.
- **`SwipeHintThemeAnimation`**: Used to indicate that an item can respond to a swipe gesture.

The Visual State Manager

The visual state manager (VSM) is a mechanism available to most of the XAML-based UI frameworks which allows a set of named visual states to be defined for a control. With each state, behaviors can be defined that affect the appearance of the control's various elements when the control is put into the state.

`VisualState` values are defined within a named **`VisualStateGroup`** element, where the **`VisualStateGroup`** contains a set of mutually exclusive visual states. The control can only be in one of the states within any group at any given time. While states from several different groups may certainly apply to a control simultaneously, it is important that two or more states defined in different groups should not modify the same property. If they do, the results will be unpredictable as to exactly how these multiple applications will affect the control. For example, if State 1 in Group A sets a control's **`Visibility`** to **`Collapsed`**, and State 2 in Group B sets the same control's **`Visibility`** to **`Visible`**, when both State A1 and State B2 are applied it is unclear which **`Visibility`** value should "win." Visual state groups are defined for a control by using the **`VisualStateManager.VisualStateGroups`** attached property.

The changes made to the UI elements for a given **VisualState** value are specified as a set of animations contained within a **Storyboard**. The following XAML illustrates a sample **VisualStateGroup** defined for a control which includes two states: **Normal** and **Abnormal**. When the control enters the **Abnormal** state, a control named **DemoButton** is located and its background is set to **Purple**. When the control leaves the **Abnormal** state, the control's background is reset to its original color.

```
<VisualStateManager.VisualStateGroups>
    <!-- Define a state group called DemoStates. -->
    <VisualStateGroup x:Name="DemoStates">
        <!-- DemoStates has 2 states: Normal and Abnormal. -->
        <VisualState x:Name="Normal"/>
        <!-- In the Abnormal state, the DemoButton control turns purple. -->
        <VisualState x:Name="Abnormal">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="DemoButton"
                                               Storyboard.TargetProperty="Background">
                    <DiscreteObjectKeyFrame KeyTime="0" Value="Purple"/>
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

A control is put into a visual state by making a call to the **VisualStateManager** static **GoToState** method as a result of some application logic. This method is given a reference to the control to which the state should be applied, as well as the name of the state it is being told to enter. A **Boolean** value indicates whether any defined state transitions should be applied as a result of the call.

```
// Tell the control to enter the Normal state.
VisualStateManager.GoToState(control, "Normal", false);

// Tell the control to enter the Abnormal state.
VisualStateManager.GoToState(control, "Abnormal", false);
```

Beyond being useful for providing a logical abstraction between a control's state and its appearance, visual states and the VSM play an important role in responding to device layout and orientation changes in Windows Store apps. This concept will be discussed in more detail later in this chapter.

Styles

Having to repeatedly specify the same property values in markup in order to provide a consistent look and feel throughout an application can be a tedious and error-prone process. Fortunately, XAML-based UIs can take advantage of styles to organize and reuse formatting values. Basically, a style is a collection of property values that can be applied to an element in a single call. This is similar in concept to how CSS entries are used in HTML page layout.

Styles are defined in markup within an element's resource dictionary, or within the application's resource dictionary, or within a merged dictionary. Values in styles are set using one or more **`Setter`** elements, where each **`Setter`** is used to specify the desired value for a named property. Note that the values being set can be simple and defined with a string, or they can be complex values expressed using the property-element syntax described previously within a **`Setter.Value`** element.

Styles can be either explicit or implicit. All styles require a **`TargetType`** property to be set with the name of the object types to which they will be applied. Explicit styles are declared with a specific key identifier and are applied to an element by setting the element's **`Style`** property to locate a specific named **`Style`** resource. The following sample shows a set of buttons using a style defined in the resource collection of the grid they are contained in. Note that individual style properties can be selectively overridden in the objects to which a style is applied, as is done for the background of the second button in the following code:

```
<Grid>
    <Grid.Resources>
        <!-- Style defined on a button to specify the background color and width. -->
        <Style x:Key="ButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="Blue"/>
            <Setter Property="Width" Value="250"/>
        </Style>
    </Grid.Resources>
    <StackPanel>
        <Button Content="Style Applied As-Is"
                Style="{StaticResource ButtonStyle}"/>
        <Button Content="Style Applied and Modified"
                Style="{StaticResource ButtonStyle}"
                Background="Orange"/>
    </StackPanel>
</Grid>
```

Implicit styles are created by omitting the **`x:Key`** identifier, and instead they only include the **`TargetType`** designation. When a style is defined in this way, any elements of the indicated type that occur as descendants of the element where the style is defined will automatically have this particular style applied. In the following code, the previous example has been updated to include an implicit style, and a new button has been added to the display which implicitly picks up the new style. Note that the buttons that have their style attributes explicitly set do not pick up the value set up in the implicit style declaration.

```
<Grid>
    <Grid.Resources>
        <!-- Style defined on a button to specify the background color and width. -->
        <Style x:Key="ButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="Blue"/>
            <Setter Property="Width" Value="250"/>
        </Style>
        <!-- Implicit style defined on a button to specify the foreground color. -->
        <Style TargetType="Button" BasedOn="{StaticResource ButtonStyle}">
            <Setter Property="Foreground" Value="Black"/>
        </Style>
    </Grid.Resources>
    <StackPanel>
```

```
        <Button Content="Style Applied As-Is"
                Style="{StaticResource ButtonStyle}"/>
        <Button Content="Style Applied and Modified"
                Style="{StaticResource ButtonStyle}"
                Background="Orange"/>
        <Button Content="Implicitly Applied Style"/>
    </StackPanel>
</Grid>
```

Note that in the new example, the implicit style includes a **BasedOn** attribute. Style definitions support inheritance where one style's definition can be based on another's. Styles cannot be inherited implicitly, but explicit styles can be inherited through the use of this **BasedOn** attribute.

Tip: As mentioned previously, the StandardStyles.xaml resource dictionary is included with Windows Store app projects created using Visual Studio templates other than the Blank App template. StandardStyles.xaml defines dozens of styles that can be applied to elements in Windows Store apps in order to give them a standard look and feel that adheres to the Microsoft style for user interface design.

One of the important tenets of XAML-based user interface controls is the idea of controls being "lookless." The concept of looklessness basically means that a control is actually defined by its functionality rather than the visual elements that are used to expose that functionality—a button is an interface element that can be clicked, regardless of whether it has a rectangular or circular outline that moves when users interact with it with their finger or a mouse. The mechanism available for providing this separation between appearance and function in XAML controls is the control template. A control's control template is actually a separate chunk of XAML markup that describes the look and content of the control.

Working with control templates is an advanced topic whose further discussion is beyond the scope of this book, but control templates are being mentioned here for two reasons. First, control templates are usually managed by applying styles in which updated templates have been defined. Second, the StandardStyles dictionary includes several examples where standard control templates have been overridden, and understanding why this is desirable can provide insight for developers interested in taking a closer look at how the custom styles in the StandardStyles dictionary have been defined.

Tip: Working with control templates is another area where Expression Blend excels. Expression Blend provides the ability to extract a control's template in order to use it as the basis for revision, and also provides several visual tools for navigating between hierarchies of controls, styles, and templates. While some of the advanced functionality for working with control templates that used to be unique to Expression Blend has been included in Visual Studio 2012, Blend continues to retain the more complete feature set.

Data Binding

Data binding is the name given to the technique of connecting a property of one object—typically a user interface object—in such a way that it will automatically get its value from and set its value to a different property on another object, or occasionally another property on the same object. XAML-based user interfaces typically take advantage of data bindings that are declared in markup and connect a dependency property on an element that derives from **`FrameworkElement`** (the target) to a property on some object from which the data will be retrieved (the source). The type used to establish and manage this connection is the **`Binding`** class. The use of data binding simplifies and otherwise eliminates a lot of boilerplate event handling and property setting code that would normally be required in order to flow data values back and forth. Removing this code also goes a long way toward taking advantage of the natural separation that a XAML-based user interface implementation provides between the user interface layout and design, and the data and business logic the user interface is implemented to display.

Before discussing the mechanisms and options available for configuring data binding, the concept of data context needs to be discussed. Any element that derives from the **`FrameworkElement`** class inherits a **`DataContext`** property. Within a XAML layout, the **`DataContext`** value is inherited—child elements inherit the **`DataContext`** of their parents until a new value is specified, which propagates down the tree of elements from that point until it is overridden or the tree runs out of elements. To illustrate this concept, in the case of the simple markup that follows, if the **`DataContext`** of the page is set to an instance of the **`Person`** class as shown, then the data context for the **`Grid`** element is the same **`Person`** object, as is the data context for the **`Button`** element.

```
<!-- The Page's XAML markup. -->
<Page
    x:Class="WindowsStoreAppsSuccinctly.DemoBlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WindowsStoreAppsSuccinctly"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sys="system."
    mc:Ignorable="d">
    <!-- The Grid inherits the Page's data context. -->
    <Grid>
        <!-- The Button inherits the Grid's data context. -->
        <Button/>
    </Grid>
</Page>

// Content from the Page's code-behind file.
public DemoBlankPage()
{
    this.InitializeComponent();
    this.DataContext = new Person
    {
        LastName = "Garland",
        FirstName = "John",
        IsEditable = true
    };
```

```
}
```

The concept of data context was presented because unless otherwise specified, data binding declarations use an element's data context as the source data to supply information. To specify a data binding value for a property in XAML, the binding markup extension is used. A simple binding declaration would take the syntax **<TextBlock Text="{Binding LastName}"/>**, where the **TextBlock** element's **Text** property, the text to display in the UI, displays the value of the **LastName** property from the object that is in the current data context. In this case, if the **LastName** property is either a dependency property or is a property in a class that participates in property change notification via an implementation of the **INotifyPropertyChanged** interface, changes to the **LastName** property would be reflected in the **TextBlock**.

Note: The `INotifyPropertyChanged` interface should be familiar to most .NET developers as a mechanism for providing property change notifications. The interface requires one item to be implemented—the `PropertyChanged` event—which when raised includes the name of the property that was changed. A simple example of a `Person` class that implements `INotifyPropertyChanged` follows. It is worth noting the `CallerMemberNameAttribute` introduced with .NET 4.5 simplifies the calls to the helper method that actually raises the event by removing the need to explicitly provide the property name in the function call.

```
// Utility class to demonstrate data context and binding concepts.
public class Person : INotifyPropertyChanged
{
    private String _firstName;
    public String FirstName
    {
        get { return _firstName; }
        set
        {
            _firstName = value;
            OnPropertyChanged();
        }
    }

    private String _lastName;
    public String LastName
    {
        get { return _lastName; }
        set
        {
            _lastName = value;
            OnPropertyChanged();
        }
    }

    private Boolean _isEditable;
    public Boolean IsEditable
    {
        get { return _isEditable; }
        set
```

```
        {
            _isEditable = value;
            OnPropertyChanged();
        }
    }

    public event PropertyChangedEventHandler PropertyChanged = delegate { };
    private void OnPropertyChanged([CallerMemberName]String caller = null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(caller));
    }
}
```

While the syntax specified in this example is the simplest possible syntax to use for data binding, there are several additional properties that can be specified when binding data. These include:

- **`Mode`**: Specifies a member of the **`BindingMode`** enumeration, which includes **`OneTime`** for bindings that only apply the data once, **`OneWay`** for bindings where the target only receives its data from the source, and **`TwoWay`** for bindings where the changes flow in both directions.
- **`ElementName`**: Used to specify an element in the current XAML that can be used as the source for data instead of the current data context.
- **`Source`**: Specifies a specific object, usually a static resource, to be used as the object for the binding.
- **`RelativeSource`**: A value of **`Self`** indicates the target element is the source for the data binding, which is useful for binding the values of properties on the same element to each other. A value of **`TemplatedParent`** instructs the binding to use the control where the control template is applied as the source for the binding data.
- **`Path`**: Specifies the property name in the data context, element, source, or relative source item the binding should be connected to. The example of **`<TextBlock Text="{Binding LastName}"/>`** is equivalent to **`<TextBlock Text="{Binding Path=LastName}"/>`**. If no path value is provided, the entire object referenced by the data context, element, source, or relative source is returned. Paths can include additional notations to indicate nested properties, attached properties, and both integer and string-based indexers.
- **`Converter`**, **`ConverterParameter`**, and **`ConverterLanguage`**: Used to specify a value converter and related properties to be used for the binding. Value converters will be discussed shortly.

In some cases, the values being provided by the objects that serve as the source for a data binding are not the proper type for the target property they are being bound to. A classic example of this mismatch is when a bound object exposes a **`Boolean`** property but the target property is a member of the **`Visibility`** enumeration (which has two values: **`Hidden`** and **`Visible`**) that is used to show or hide user interface elements in XAML-based UIs. To help deal with these kinds of mismatches, the binding system supports the use of value converters.

A value converter is a type that implements the **IValueConverter** interface. This interface specifies the methods **Convert** and **ConvertBack** which can be used to convert from the source data type to the target type and vice versa. For the **Boolean**/**Visibility** situation described previously, projects based on the templates supplied by Visual Studio other than the Blank App template provide an implementation of the **BooleanToVisibilityConverter** class, defined as follows:

```
/// <summary>
/// Value converter that translates true to <see cref="Visibility.Visible"/>
/// and false to <see cref="Visibility.Collapsed"/>.
/// </summary>
public sealed class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                                                                    string language)
    {
        return (value is bool && (bool)value)
            ? Visibility.Visible
            : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                                                                    string language)
    {
        return value is Visibility && (Visibility)value == Visibility.Visible;
    }
}
```

This implementation simply translates between a **Boolean** value and the corresponding member of the **Visibility** enumeration.

Value converters can also optionally receive parameters which specify the language to be considered for the conversion, as well as an arbitrary parameter that can be used to provide some other additional information.

Converters are specified as resources referenced in the binding expression that intends to use them. The following markup shows the use of a value converter to translate between the **Boolean IsEditable** property on the **Person** object, which is bound to a **CheckBox**, and the visibility of the **TextBox** that allows editing the person's **LastName** property:

```
<StackPanel Orientation="Horizontal">
    <StackPanel.Resources>
        <common:BooleanToVisibilityConverter x:Key="BoolConverter"/>
    </StackPanel.Resources>
    <CheckBox Content="Is Editable" IsChecked="{Binding IsEditable, Mode=TwoWay}"/>
    <TextBlock Text="Last Name: " VerticalAlignment="Center" Margin="0,10,10,10"/>
    <TextBox Text="{Binding LastName, Mode=TwoWay}"
         Visibility="{Binding  IsEditable, Converter={StaticResource BoolConverter}}"/>
</StackPanel>
</StackPanel>
```

Adding Content

The XAML examples up to this point have included several different kinds of items without necessarily explaining their individual purposes and uses. This section will introduce several different categories of controls that can be included in a XAML-based UI and provide a high-level explanation for the strengths and uses of several of the most common controls.

In addition to the controls included out of the box, developers may opt to create new controls—either via compositing existing controls and providing code to expose their specific behavior, or by taking the significantly more complex task of creating new controls programmatically from the ground up. Besides custom controls created by developers, Microsoft offers a variety of specialized controls for Windows Store apps within various SDKs that can be downloaded. Furthermore, third-party tool manufacturers are already providing many unique and helpful controls for Windows Store apps that can be purchased, and in many cases the tools include trial modes for evaluating the functionality they provide.

Note: As an example, this chapter has already mentioned the Essential Studio for WinRT XAML control suite sold by Syncfusion, which also happens to be the publisher of this book. Additional information about this control suite can be obtained at http://www.syncfusion.com/products/winrt.

Layout Panels

The first set of controls to consider are those whose purpose is to contain and lay out other controls. Most of the work that developers will do to lay out a UI for a Windows Store app occurs within the **`Page`** element. However, **`Page`** elements can only contain a single content element. In the rare case that a page only requires a single element to be displayed, this will be easy enough to work with. However, most of the time developers will want to include a variety of elements on a page, with different needs as to exactly how the elements should be positioned relative to each other and how changes in screen orientation and size should be reflected in this layout. To handle these needs, Windows Store apps can take advantage of several different kinds of panel elements, including the **`Grid`**, the **`StackPanel`**, and the **`Canvas`**. This section will also consider a couple of special panel controls which can be used within controls that are built to present lists of items to users: the **`WrapGrid`** and the **`VariableSizedWrapGrid`**.

Grid

The **`Grid`** is perhaps the most widely used and most versatile layout panel. A grid can be configured with a number of rows and columns into which other UI elements can be placed. Placement of items within the grid is managed primarily through the use of the **`Grid.Row`** and **`Grid.Column`** attached properties although there are a few other properties that can be applied to affect the layout as well. Row heights and column widths in a grid can be set to a fixed point size, automatically sized to the content they contain, or set in a notation called “star sizing” which allows a kind of dynamic, proportional sizing to take place. Star sizing is so named because it is indicated by using an asterisk and an optional multiplier value. A multiplier of 1 is used as the default if an explicit value is omitted.

Star sizing behaves as follows:

- For star-sized rows or columns, the total available size for the grid in that direction (horizontal or vertical) is determined.
- The total number of stars in a given direction is calculated by summing the multipliers of those rows or columns.
- The single-star size is computed by dividing the available size by the number of stars.
- The width for each column is determined by multiplying the single-star size by its multiplier.

This relationship is illustrated in the following XAML. It defines an 800-point wide grid with two rows and four columns. The first column is set to be a fixed width. The second is automatically sized to the size of its contents. The third and fourth columns are set to be 1-star and 3-star sized, respectively. As is shown in the comments, the second column sizes to 75 points due to the size called out in the content targeted to that column. The third and fourth are sized to 150 and 450 points each based on the 1-star and 3-star sizes.

```
<Grid Width="800">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <!-- Explicitly set to 125 points wide. -->
        <ColumnDefinition Width="125"/>
        <!-- Set to 75 points wide because of contents' size. -->
        <ColumnDefinition Width="Auto"/>
        <!-- 150 points wide = (800-(125+75))/(1+3) x 1 star. -->
        <ColumnDefinition Width="*"/>
        <!-- 450 points wide = (800-(125+75))/(1+3) x 3 stars. -->
        <ColumnDefinition Width="3*"/>
    </Grid.ColumnDefinitions>
    <Button Grid.Row="0" Grid.Column="0" Content="Column 1"/>
    <Button Grid.Row="0" Grid.Column="1" Content="Col. 2" Width="75"/>
    <Button Grid.Row="0" Grid.Column="2" Content="Column 3"/>
    <Button Grid.Row="0" Grid.Column="3" Content="Column 4"/>

    <Button Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3"
              Content="This stretches over 3 columns"/>
</Grid>
```

This example also illustrates how content is added to grid cells using the **`Grid.Row`** and **`Grid.Column`** attached properties. It also shows the use of the **`Grid.ColumnSpan`** attached property to allow content to span multiple columns. The expected **`Grid.RowSpan`** counterpart is also available.

There are two additional things to be said about the Grid control. The first is that items placed within the same grid cell are composed; they are drawn on top of each other. The second is that a grid needs a background color to be set in order to respond to user interaction events like the **Tapped** event. By default, a grid's background color is null. If the layout logic is such that the Grid should be able to raise these events, but no specific background color is desired, the background color can be set to **Transparent**, e.g., **<Grid Background="Transparent" Tapped/>**.

StackPanel

The next panel to consider is the **StackPanel**. The StackPanel does as its name implies: it stacks the elements it contains either vertically (the default) or horizontally, based on the value that is set on the panel's **Orientation** property. The StackPanel is often used in concert with controls that provide scrolling in order to present lists of information. The following markup shows a simple stack panel that stacks two buttons vertically along with another stack panel that stacks three buttons horizontally.

```
<StackPanel>
    <Button Content="Vert Stack 1"/>
    <Button Content="Vert Stack 2"/>
    <StackPanel Orientation="Horizontal">
        <Button Content="Horiz Stack 1"/>
        <Button Content="Horiz Stack 2"/>
        <Button Content="Horiz Stack 3"/>
    </StackPanel>
</StackPanel>
```

Note: There is a specialization of the `StackPanel` available called the `VirtualizingStackPanel`. This control is useful when a large amount of items is being placed within the `StackPanel`. Under normal circumstances, the `StackPanel` will render all of the items it contains, regardless of whether they are outside the boundaries of the current screen. For large enough collections, this can cause performance issues and lead to a UI that seems unresponsive. The `VirtualizingStackPanel` only renders the items that appear on-screen, deferring the calculations and other resources necessary for the remaining items until they are somehow scrolled into view. This deferred calculation will result in a decrease in the "smoothness" of scrolling since items will only be able to be scrolled on screen one entire item at a time. Nonetheless, doing so can improve the overall responsiveness of the app by not making it responsible for tracking rendered items that are not being displayed to users. Ultimately, using the `VirtualizingStackPanel` is a trade-off whose applicability depends on each situation.

Canvas

The main panels described so far have been based on the concept of providing flexible rather than coordinate-based layout. To support such a layout where items are placed at explicitly defined coordinates, the **Canvas** panel is available. While the **Canvas** panel is most likely to resonate with application developers coming from a background in WinForms or its predecessors, the control itself isn't particularly suited to the layout flexibility that is expected in Windows Store apps. Therefore its use should be constrained to situations where it is targeted and best suited to the task at hand, which is primarily drawing-based application functionality and other specialized uses.

In addition to providing **Canvas.Left** and **Canvas.Top** values for the x-coordinates and y-coordinates for laying out items on a canvas, it is also possible to provide a **Canvas.ZIndex** value that determines how the items positioned on the canvas are overlaid relative to each other. Note that the **Canvas.ZIndex** property actually can be applied to other **Panels** where images are composited, such as the **Grid** panel discussed previously.

```
<Canvas>
    <Button Content="Positioned at 75,20" Canvas.Left="75" Canvas.Top="20"/>
</Canvas>
```

Special Purpose Panels

There are two additional panels worth mentioning, though they are constrained in terms of only being able to be used in specific situations. The **WrapGrid** and the **VariableSizedWrapGrid** controls provide the ability to automatically position content in a wraparound grid rather than having to explicitly set row and column values. The benefit these controls offer is when the UI changes due to an orientation or screen size change, the content is automatically redistributed within the grid to accommodate the new size. Both controls support the ability to indicate whether the items should be oriented horizontally or vertically, which determines the order in which items are placed in the UI. Horizontal orientation adds content left-to-right, and then adds new rows as necessary; whereas vertical orientation adds content top to bottom, adding new columns as necessary. The determination to move to a new row or column is based on the value set in the **MaximumRowsOrColumns** property. For horizontal orientation, this value indicates how many column entries are made before a new row is added, and for vertical orientation it indicates how many row entries are made before a new column is added.

The **VariableSizedWrapGrid** enhances the behavior of the **WrapGrid** by allowing elements to specify a **VariableSizedWrapGrid.RowSpan** and a **VariableSizedWrapGrid.ColumnSpan** attached property. These properties are used to determine if the element should be given extra space in the UI, providing access to a mechanism that allows UI elements to be included that can take up additional space like the wide tiles do in the Windows 8 Start screen.

As previously noted, the use of these controls is restricted; they can only be used as panels for laying out elements inside controls that inherit from **ItemsControl**. Item controls will be discussed a little later in this chapter.

The following markup shows the **WrapGrid** and **VariableSizedWrapGrid** used to provide layout:

```
<!-- WrapGrid sample. -->
<ItemsControl>
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapGrid ItemHeight="50" ItemWidth="200"
                      MaximumRowsOrColumns="2" Orientation="Vertical"/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <Button Content="Wrap1"/>
    <Button Content="Wrap2"/>
    <Button Content="Wrap3"/>
    <Button Content="Wrap4"/>
    <Button Content="Wrap5"/>
    <Button Content="Wrap6"/>
</ItemsControl>


<!--VariableSizedWrapGrid sample. -->
<ItemsControl>
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <VariableSizedWrapGrid ItemHeight="50" ItemWidth="200"
                                   MaximumRowsOrColumns="2" Orientation="Vertical"/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <Button Content="VarWrap1" Width="400" VariableSizedWrapGrid.ColumnSpan="2"/>
    <Button Content="VarWrap2"/>
    <Button Content="VarWrap3"/>
    <Button Content="VarWrap4"/>
    <Button Content="VarWrap5"/>
    <Button Content="VarWrap6" Height="90" VariableSizedWrapGrid.RowSpan="2"/>
</ItemsControl>
```

Content Controls

The next set of controls to discuss is the controls used to add functionality to the display of a single piece of content (for the most part). Most of these kinds of controls are known as content controls because they derive from the **`ContentControl`** class.

Data Templates

As was mentioned, the **`ContentControl`** is used to display some content. This content can be another UI element, providing simple nesting of items, or it can be a piece of data, perhaps as a result of data binding. Content controls introduce the extremely powerful and useful concept of data templates. Data templates are pieces of XAML that typically include data bindings and are used to describe how a piece of non-visual data is to be rendered as a visual element in the UI. A data template is specified in markup using the **`DataTemplate`** class and assigning it to the content control's **`ContentTemplate`** property. If no data template is provided through the **`ContentTemplate`** property, data content is simply rendered as a string. Otherwise, the data template is used to provide the layout that is to be applied to present the data.

Tip: For Windows Store apps, the `ContentTemplateSelector` property can be used to specify a `DataTemplateSelector` instance. The `DataTemplateSelector` takes the concept of data templates a step further in that the implementation of a class that inherits from the `DataTemplateSelector` can provide logic to examine the data being presented in order to dynamically select the data template to be used, providing simple and powerful data-driven UI.

The following code shows several different approaches to displaying data in a basic **`ContentControl`**. The first control simply renders text, the second displays composite content within a horizontal **`StackPanel`**, and the third binds the content to the current **`DataContext`** and then applies a **`DataTemplate`** to render the content.

```
<StackPanel>
    <StackPanel.Resources>
        <local:Person x:Key="SamplePerson"
                      FirstName="John"
                      LastName="Garland"
                      IsEditable="True"/>
        <Style TargetType="TextBlock" BasedOn="{StaticResource BasicTextStyle}"/>
    </StackPanel.Resources>

    <!-- ContentControl that shows simple text content. -->
    <ContentControl Content="This is text content"/>

    <!-- ContentControl that includes complex/composite content. -->
    <ContentControl>
        <ContentControl.Content>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Composite Content"/>
                <Button Content="Click Me"/>
            </StackPanel>
        </ContentControl.Content>
    </ContentControl>

    <!-- ContentControl that uses a DataTemplate to work with a bound Person object. -->
    <ContentControl DataContext="{StaticResource SamplePerson}" Content="{Binding}">
        <ContentControl.ContentTemplate>
            <DataTemplate>
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding LastName}"/>
                    <CheckBox Content="Editable"
                              IsChecked="{Binding IsEditable, Mode=TwoWay}"/>
                </StackPanel>
            </DataTemplate>
        </ContentControl.ContentTemplate>
    </ContentControl>
</StackPanel>
```

While data templates are often useful in content controls, their benefits really light up when used with items controls, as will be discussed later.

Common Content Controls

Having discussed the foundations of content controls, the next interesting aspect is the fact that many of the custom controls that are used within XAML UIs are in fact derived from content controls. This includes, among others:

- The **Button** control and its descendants (**CheckBox**, **RadioButton**, **RepeatButton**, **ToggleButton**, **ToggleSwitch**): Having the **Button** derive from the **ContentControl** makes it trivial to include more than just text content in button displays, including images and shapes.
- The **Border** control: The **Border** control is a lightweight control that can be used to provide outlines and background fills for the child element it contains.
- The **ScrollViewer** control: The **ScrollViewer** control allows content to be included that is potentially larger than the available display area, and it will handle the logic necessary to present scroll bars in order to bring the additional content into view.
- The **AppBar** control: The **AppBar** control provides an implementation of the application bar that is used as a standard mechanism for displaying commands in Windows Store apps. The **AppBar** will be discussed in more detail shortly.

The following code shows how simple it is to include both an image and some text inside of a **Button** control, taking advantage of its nature as a **ContentControl**:

```
<!-- A button showing the ability to include composite content. -->
<Button>
    <Button.Content>
        <StackPanel Orientation="Horizontal">
            <Image Source="ms-appx:///Assets/Query.png" Height="20"/>
            <TextBlock Text="Button with Composite Content"/>
        </StackPanel>
    </Button.Content>
</Button>
```

Items Controls

Whereas content controls are focused on presenting a single item of content, items controls are used to show collections of items in the UI. These controls derive from the **ItemsControl** class.

With items controls, the collection of items to be displayed is maintained in the **Items** property. Values can be added directly to this collection, or data binding can be used through the **ItemsSource** property to connect to the collection of data items to be displayed. Collections that are data-bound must implement the **IEnumerable** interface, and if there is a desire to display any updates in the UI, they must also implement the **INotifyCollectionChanged** interface. The .NET Framework provides a useful class that implements both of these interfaces: **ObservableCollection<T>**.

Note: It is possible to update the display of bound collections that do not implement *INotifyCollectionChanged* ***as long as the property they are exposed with participates in the*** *INotifyPropertyChanged* ***implementation. However, doing this refreshes the binding itself, causing the entire collection to be reset. This usually results in visible***

flickering and "losing your place" for collection displays that involve scrolling or selection. This may or may not provide an acceptable user experience, depending on the specific circumstances.

The following code shows the two ways the **Items** list of an **ItemsControl** can be populated:

```
<!-- An ItemsControl with content added directly to it. -->
<ItemsControl>
    <ItemsControl.Items>
        <x:String>First Item</x:String>
        <x:String>Second Item</x:String>
        <x:String>Third Item</x:String>
    </ItemsControl.Items>
</ItemsControl>

<!-- An ItemsControl data-bound to a collection. -->
<ItemsControl ItemsSource="{Binding SimpleItems}"/>
```

Configuring Individual Item Display

Just as with content controls, the way that individual items are displayed depends on a few factors. If the **DisplayMemberPath** property has been defined, the object being displayed is examined for the property specified in the **DisplayMemberPath** (nested properties can be used through "dot-down" syntax), and if it is found, that value is used as the item being displayed instead of the entire item. This provides a quick and handy way to bind a list of objects to an **ItemsControl**, but to display a particular property from each of those items. Without a **DisplayMemberPath**, if the item is a UI element, that element is rendered. Otherwise, the item's **ToString** method is used to render the item as text, unless a data template has been provided.

A data template is provided for the items being displayed through the **ItemTemplate** property. When this value is set to a valid **DataTemplate**, the items in the list will be rendered using the XAML provided in the **DataTemplate**, with the **DataContext** of each item being rendered set to the respective list item. Coupled with data binding in the contents of the specified template, this is an extremely powerful way to present complex UI representations contextualized to individual items within list elements. Windows Store apps can tap into even more power and flexibility by using the **ItemTemplateSelector** to be able to select which **DataTemplate** should be presented based on the type or other attributes of the individual data items being displayed.

The following code shows an **ItemsControl** displaying a bound collection of **Person** objects using both **DisplayMemberPath** and a data template:

```
<!-- ItemsControl with binding using DisplayMemberPath to display a single property. -->
<ItemsControl ItemsSource="{Binding ComplexItems}" DisplayMemberPath="LastName"/>

<TextBlock Text="Using Data Templates" Margin="0,10,0,0"/>
<!-- ItemsControl with binding using ItemTemplate to control the display. -->
<ItemsControl ItemsSource="{Binding ComplexItems}">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <StackPanel.Resources>
```

```
                    <Style TargetType="TextBlock"
                           BasedOn="{StaticResource ItemTextStyle}"/>
                </StackPanel.Resources>
                <TextBlock Text="{Binding LastName}"/>
                <TextBlock Text=", "/>
                <TextBlock Text="{Binding FirstName}" FontSize="12"/>
            </StackPanel>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Changing the Display Surface

One additional property that can be set to affect the display of items in an **ItemsControl** is to change the panel used to determine how items are placed relative to each other. The normal default panel used in most items controls is the **StackPanel** control, which provides a vertical layout where items are stacked on top of each other. The actual panel used to control item placement is specified through the **ItemsPanel** property. Changes to this property can be as simple as using a **StackPanel** with its **Orientation** set to **Horizontal** to make the list stack left to right instead of top to bottom. As has been discussed, however, there are some additional panels that can be specified which provide additional layout options for content in an **ItemsControl**. These include the **VirtualizingStackPanel** used to conserve system resources when displaying large lists, or the **WrapGrid** and **VariableSizedWrapGrid** which can change the orientation and relative sizing of displayed items. The following code shows the panel swapped out to use a **WrapGrid**.

```
<!-- ItemsControl with an alternate panel to control the display. -->
<ItemsControl ItemsSource="{Binding ComplexItems}" DisplayMemberPath="LastName">
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapGrid ItemWidth="100" MaximumRowsOrColumns="2"
                      Orientation="Horizontal"/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
</ItemsControl>
```

Common Items Controls

The **ItemsControl** discussed so far does have its fair share of uses, especially when wrapped inside of a **ScrollViewer** control and used to display items where very simple list presentation is desired. However, in many cases, there's some additional functionality required when presenting lists of items, such as single and multiple selection, item highlighting, and others. Windows Store apps have access to several out-of-the-box controls that inherit from the **ItemsControls** to get this additional functionality, while providing the **DataTemplate**, **Panel**, and other rich features offered by the **ItemsControl** (there are many third-party controls that offer excellent functionality as well). These controls include, among others:

- The **ComboBox** control: A familiar control used to provide a selection from a list of items that is displayed only when a selection is being made; otherwise it is in a closed state to preserve UI space.

- The **ListView** control: Used to provide an interactive display of a list of items to be shown vertically. The **ListView** includes support for displaying item groupings, single and multiple item selection, and responding to an item being clicked on. It also supports incremental loading of data for data sources that implement the **ISupportIncrementalLoading** interface. It is also one of the controls that can participate in a **SemanticZoom** view, which will be discussed later.
- The **GridView** control: A new control introduced for Windows Store apps. Like the **ListView**, the **GridView** control also provides an interactive display of a list of items, with its focus being the horizontal display of information. The **GridView** supports item groupings, single and multiple item selection, and responding to an item being clicked on. It also supports incremental loading of data for data sources that implement the **ISupportIncrementalLoading** interface. Like the **ListView** control, it can participate in a **SemanticZoom** view. The **GridView** control will be discussed in more detail later.
- The **FlipView** control: The **FlipView** control is also a new control introduced for Windows Store apps. It is a basic **ItemsControl** used in scenarios where a collection of items is to be presented one at a time, allowing either swiping or pressing small buttons on the control's margins to move between items in the list. While ideally suited for image galleries, it is also very useful for other situations where a detailed view of one item in a list is desired without needing to see information about the remaining items. Pages in a book or a monthly calendar view are good candidates for a **FlipView** control.

The following code shows a **ComboBox** control bound to a collection of **Person** objects, along with a **TextBlock** that uses data binding to display the last name of the item selected in the **ComboBox**:

```
<!-- ComboBox populated from a set of Person objects, displaying the last name. -->
<ComboBox x:Name="ComboBoxDemo" ItemsSource="{Binding ComplexItems}"
          DisplayMemberPath="LastName"/>

<!-- Bind to display the item selected in the ComboBox. -->
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Selected Person:"/>
    <TextBlock DataContext="{Binding SelectedItem, ElementName=ComboBoxDemo}"
               Text="{Binding LastName}"/>
</StackPanel>
```

Working with the GridView Control

The **GridView** is one of the more important items controls available for use in Windows Store apps. In fact, it is such a cornerstone of Windows Store app development that two of the three out-of-the-box, Visual Studio application-focused project templates are starter apps essentially built around interaction with a **GridView** control—the Grid App and the Split App project templates to be exact. As mentioned previously, the **GridView** is an **ItemsControl** focused on presenting collections of items that scroll horizontally. The **GridView** supports some advanced collection presentation features, including the ability to present grouped data. It is also one of the built-in controls that can work with Semantic Zoom, which will be discussed in the next section.

The **GridView** is actually an instance of a class derived from **ItemsControl** called the **Selector** class. The **Selector** class adds properties and events related to the selection of an item such as the **SelectedIndex** property, the **SelectedItem** property, and the **SelectionChanged** event. The **GridView** augments these selection members with a **SelectionMode** property that can be used to indicate whether the control currently supports item selection, and if so, whether the selection mode supports single selection, multiple selection, or extended mode selection. Extended selection allows multiple separate items to be selected when holding Ctrl, and multiple contiguous items to be selected when holding Shift. It also provides a **SelectedItems** property to support the mentioned multiple-selection modes.

Note: The `GridView` control also supports an `IsItemClickEnabled` property that determines whether clicking or tapping on individual items raises the control's `ItemClick` event. There is a subtle interplay between the `IsItemClickEnabled` property and the `SelectionMode` value. When `IsItemClickEnabled` is true, tapping or left-clicking on an item raises the `ItemClick` event. If the single, multiple, or extended `SelectionMode` is applied, swiping down on an item or right-clicking on it will toggle the item's selection state. When `IsItemClickEnabled` is false and the `SelectionMode` is set to one of these values, tapping or left-clicking and swiping or right-clicking both toggle an item's selection state.

GridView Basics

The XAML setup for a very basic **GridView** control is shown in the following code. It uses data binding to bind the control's **Items** collection to an **AllItems** value in the current **DataContext**. It also sets the **ItemTemplate** property to a **DataTemplate** defined within the resources collection, and establishes event handlers for the **SelectionChanged** and **ItemClick** events which are enabled because the **SelectionMode** is set to **Multiple** and the **IsItemClickEnabled** property is set to **true**:

```
<GridView x:Name="DemoGridView"
          ItemsSource="{Binding AllItems}"
          ItemTemplate="{StaticResource Standard250x250ItemTemplate}"
          SelectionMode="Multiple"
          IsItemClickEnabled="True”
          SelectionChanged="HandleGridViewSelectionChanged"
          ItemClick="HandleGridViewItemClick">
</GridView>
```

The **ItemClick** event handler can retrieve which item was clicked through the arguments sent in the event:

```
private async void HandleGridViewItemClick(object sender, ItemClickEventArgs e)
{
    // Determine the data item that was clicked.
    var clickedItem = (SampleDataItem)e.ClickedItem;
```

```
    // Show a message indicating the clicked item.
    var message = String.Format("Item {0} was clicked", clickedItem.Title);
    var clickMessage = new MessageDialog(message, "GridView Demo");
    await clickMessage.ShowAsync();
}
```

Likewise, the **SelectionChanged** event handler receives information about which items have been selected and deselected. It is also possible to get the list of selected items by interrogating the **GridView** control's **SelectedItems** property directly:

```
private void HandleGridViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    // From the arguments, determine the items newly selected or deselected.
    var newlySelectedItems = e.AddedItems;
    var deselectedItems = e.RemovedItems;

    // Get the comprehensive list of selected items.
    var allSelectedItems = DemoGridView.SelectedItems;
}
```

Tip: The control in the previous example was styled using the Standard250x250ItemTemplate data template. This template is included in the StandardStyles.xaml resource dictionary previously discussed in this chapter. This template is used in the default GridView pages created with Visual Studio Project templates and the Items and GroupedItems pages available from the Add New Item dialog. It is useful as either a baseline template or as a reference for getting started with creating GridView DataTemplates for use in Windows Store apps.

Working with Groups

One of the big advantages of using a **GridView** is its support for presenting grouped data. To present data in groups, the **GridView** can work with the **CollectionViewSource** class. The **CollectionViewSource** class adds a layer of functionality over a collection of data, providing information about the relationship between group items and their child collections.

A **CollectionViewSource** is typically declared in a resources collection accessible to the **GridView** or other controls it will be used for. To provide grouping functionality, the **CollectionViewSource** item's **Source** is either set directly or data-bound to the collection of groups to be displayed, the **IsSourceGroup** property is set to **true**, and its **ItemsPath** is used to identify the property within each of the group items that exposes the collection of items within that particular group.

```
<!-- Define a CollectionViewSource bound to the GroupedItems collection. -->
<!-- The ItemsPath property defines where items are located within each group. -->
<CollectionViewSource
    x:Name="GroupedItemsViewSource"
    Source="{Binding GroupedItems}"
    IsSourceGrouped="true"
    ItemsPath="Items"/>
```

Once the **CollectionViewSource** is configured, it can be bound to the **ItemsSource** property of a **GridView** control. The **GridView** control provides a **GroupStyle** property which is used to set the visual elements for presenting grouped data. The **GroupStyle** class includes a **HeaderTemplate** property that accepts a **DataTemplate** defining how the headers for each group item will appear. The **Panel** property defines the panel to be used to lay out the individual items in the group, and the **HidesIfEmpty** property is used to indicate whether empty groups should be shown or omitted from the display. The following code defines a **HeaderTemplate** as a **Button** displaying both its group's title and a decorative chevron character. The button's click event is configured to call a handler method named **HandleGroupedHeader**. The panel used for the groups is also defined, with each panel separated by 80 pixels.

```
<GroupStyle.HeaderTemplate>
    <DataTemplate>
        <Grid Margin="1,0,0,6">
            <Button Click="HandleGroupedHeaderClick"
                    Style="{StaticResource TextPrimaryButtonStyle}" >
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding Title}" Margin="3,-7,10,10"
                               Style="{StaticResource GroupHeaderTextStyle}" />
                    <TextBlock Text="{StaticResource ChevronGlyph}"
                               FontFamily="Segoe UI Symbol" Margin="0,-7,0,10"
                               Style="{StaticResource GroupHeaderTextStyle}"/>
                </StackPanel>
            </Button>
        </Grid>
    </DataTemplate>
</GroupStyle.HeaderTemplate>

<GroupStyle.Panel>
    <ItemsPanelTemplate>
        <VariableSizedWrapGrid Orientation="Vertical" Margin="0,0,80,0"/>
    </ItemsPanelTemplate>
</GroupStyle.Panel>
```

Note: The `GridView` control shares much of its programming interface with the corresponding `ListView` control. As has been mentioned, whereas the `GridView` control is primarily used to display horizontally scrolling data, the `ListView` control is used to display vertically scrolling data. Because of their inherent similarities, the majority of the concepts presented in this section for the `GridView` control translate directly and can be applied to the `ListView` control.

Implementing Semantic Zoom

The concept of Semantic Zoom involves displaying different collections of interface elements as a user zooms in or out, usually presented as higher or lower levels of abstraction depending on the zoom direction. When applied to the display of grouped data, a user experience that supports Semantic Zoom typically shows a main zoomed-in view that presents individual data items, possibly segmented by some sort of grouping or other categorization. When users select the zoomed-out view, the interface then shows only the groups that contain those individual items. This experience is primarily helpful for scenarios where a large amount of data is being presented and a mechanism is needed to facilitate navigation to key locations within that set of data.

Several of the experiences built into Windows 8 provide Semantic Zoom implementations, including:

- **People app**: In its zoomed-in view, it displays a list of contact tiles grouped by letter. In its zoomed-out view, it displays a set of tiles corresponding to letters in the alphabet that can be used to quickly navigate to the corresponding section of the available contacts.
- **Store app**: In its zoomed-in view, it displays collections of tiles linked to information about either individual applications or organized collections of applications, grouped by application categories. In its zoomed-out view, it shifts focus to displaying the application categories.
- **Windows 8 Start screen**: In its zoomed-in view, it displays the users' app tiles and app tile groups. In its zoomed-out view, it abstracts the individual tiles and focuses on displaying the groups into which the tiles have been clustered, allowing quick navigation to one of these groups.

Windows Store apps can include Semantic Zoom experiences through the use of the **`SemanticZoom`** control. This control exposes both **`ZoomedInView`** and **`ZoomedOutView`** properties. The values assigned to these properties must implement the **`ISemanticZoomInformation`** interface—the two controls that implement this interface are the **`GridView`** and **`ListView`** controls. The control provided for the **`ZoomedInView`** will be the primary UI display.

The code that follows shows a **`SemanticZoom`** control populated with two **`GridView`** controls. The **`ZoomedInView`** displays the same content as was shown in the previous discussion concerning the **`GridView`**. The **`ZoomedOut`** view binds to the same **`CollectionViewSource`** instance, but uses a path to the **`CollectionGroups`** property exposed by the **`CollectionViewSource`** to only display the group-level items. Each group is displayed within a 250 × 250 grid using the familiar 250 × 250 template discussed previously.

```
<SemanticZoom x:Name="SemanticZoomView">
    <SemanticZoom.ZoomedInView>
        <!-- The zoomed-in / primary view. -->
        <GridView
            x:Name="itemGridView"
            ScrollViewer.IsHorizontalScrollChainingEnabled="False"
            ItemsSource="{Binding Source={StaticResource GroupedItemsViewSource}}"
            ItemTemplate="{StaticResource Standard250x250ItemTemplate}"
            <GridView.GroupStyle>
                <GroupStyle>
```

```
                    <!-- Content omitted for brevity. -->
                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>
    </SemanticZoom.ZoomedInView>
    <SemanticZoom.ZoomedOutView>
        <!-- The zoomed-out view -->
        <GridView x:Name="ZoomedOutList"
                  ScrollViewer.IsHorizontalScrollChainingEnabled="False"
                  ItemsSource="{Binding Source={StaticResource GroupedItemsViewSource},
                                        Path=CollectionGroups}">
            <GridView.ItemsPanel>
                <ItemsPanelTemplate>
                    <WrapGrid ItemWidth="250" ItemHeight="250"
                              MaximumRowsOrColumns="2"
                              VerticalChildrenAlignment="Center" />
                </ItemsPanelTemplate>
            </GridView.ItemsPanel>
            <GridView.ItemTemplate>
                <DataTemplate>
                    <ContentControl DataContext="{Binding Group}"
                       ContentTemplate="{StaticResource Standard250x250ItemTemplate}"/>
                </DataTemplate>
            </GridView.ItemTemplate>
        </GridView>
    </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

The Application Bar

Windows Store apps make use of the application bar (app bar) to display application and navigation commands. The app bar will appear at either the bottom of a page, the top of a page, or both. The app bar is hidden by default and can be displayed through user interaction by swiping from the bottom of the screen up or from the top of the screen down, right-clicking if using a mouse, or pressing Windows logo key+Z if using a keyboard. It can also be invoked programmatically, which is most often used to display contextual commands when a list item is selected.

The general guidance for the use of the app bar indicates that the lower app bar should display application commands, whereas the top app bar should be used to display commands related to navigation through the application. Furthermore, care should be taken to ensure that the commands placed in the app bar do not duplicate functionality that is otherwise available in the charms bar, such as functionality related to settings, search, and sharing (these items will be discussed in a later chapter). Microsoft has published a set of guidelines illustrating proper use of the app bar in a Windows Store app. These guidelines can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh781231.aspx

In XAML-based Windows Store apps, the **`AppBar`** control is used to display content in an app bar. The **`AppBar`** control is actually a content control, meaning it can host a single child, so one of the previously mentioned layout panels must be hosted within the **`AppBar`** control in order to place controls for multiple commands on the app bar. The **`Page`** class contains **`BottomAppBar`** and **`TopAppBar`** properties that accept **`AppBar`** controls.

App bar command buttons usually follow a very specific visual style, especially when they are included in the bottom app bar. The previously mentioned StandardStyles.xaml resource dictionary includes a definition of the **AppBarButtonStyle**, which can be applied to a **Button** control to share this common look and feel. When using the **AppBarButtonStyle**, it is necessary to provide a glyph (the icon displayed inside the circular button outline) and a label to appear beneath the circled glyph. The Segoe UI Symbol font is ideally suited to provide symbols that can be used within a button with the **AppBarButtonStyle** applied. The content is specified using the Unicode character code for the desired glyph, which can be obtained from the Character Map application included in Windows. The label is provided by setting the **AutomationProperties.Name** property. The following code will produce the app bar buttons shown in Figure 8:

```
<!-- Sample custom app bar buttons.-->
<Button Style="{StaticResource AppBarButtonStyle}"
        Content="&#xE125;"
        AutomationProperties.Name="Demo 1"/>
<Button Style="{StaticResource AppBarButtonStyle}"
        Content="&#xE109;"
        AutomationProperties.Name="Demo 2"/>
<Button Style="{StaticResource AppBarButtonStyle}"
        Content="&#xE108;"
        AutomationProperties.Name="Demo 3"/>
```

Figure 8: The Corresponding App Bar Buttons

The StandardStyles.xaml dictionary also contains several predefined styles for common-purpose app bar buttons. By default, however, these styles are all commented out in order to keep unneeded styles from unnecessarily consuming application resources. In order to use the styles they can either be selectively uncommented directly within the StandardStyles resource dictionary, or the styles that are desired can be copied into an appropriate resource dictionary elsewhere in the application.

```
<!-- Sample predefined app bar button styles. -->
<Button Style="{StaticResource AddAppBarButtonStyle}"/>
<Button Style="{StaticResource RemoveAppBarButtonStyle}"/>
```

Another important item included in app bar guidance relates to how app bars should behave when they contain commands that relate to selectable items on the current page, such as items contained within a **ListView** or **GridView** that is configured to support selection. In these cases, the app bar should be automatically displayed when one or more items are selected and should be collapsed when all items have been deselected. This can be accomplished using the **AppBar** control's **IsSticky** and **IsOpen** properties. The following code can be included in one of these controls' **SelectionChanged** event handlers and looks to see if any items are selected. If so, it sets the app bar to "sticky mode" and opens the app bar. Otherwise it clears the sticky mode and collapses the app bar.

```
// If any items are selected, force the app bar to be "sticky"
// and display it. Otherwise, clear the sticky behavior and
// collapse the bar.
if (AppBarListView.SelectedItems.Any())
{
    this.BottomAppBar.IsSticky = true;
    this.BottomAppBar.IsOpen = true;
}
else
{
    this.BottomAppBar.IsOpen = false;
    this.BottomAppBar.IsSticky = false;
}
```

If extra information is needed to perform a command selected from the app bar, it may make sense to show an additional UI element to allow users to enter that information. In Windows Store apps, this is typically accomplished by using a **Popup** control. Furthermore, as will be covered in the discussion concerning view states later in this chapter, care should be taken in Windows Store apps to ensure the app bar can accommodate the number of commands being presented, not only in the app's landscape view, but also in portrait and snapped views. In some cases it may make sense to group several related app bar commands into a single parent command which when selected displays a **PopupMenu** control that lists the surrogate commands. The **Popup** and **PopupMenu** controls will both be explored in a subsequent section.

The Viewbox Control

The **Viewbox** control is a container control capable of holding a single element, but it is technically not a content control, hence its exclusion from that particular discussion. The purpose of the **Viewbox** control is to automatically resize the content it contains. There are four resize mode options available to the **Viewbox** control, which is set with members of the **Stretch** enumeration through the control's **Stretch** property:

- **None**: The content is not resized
- **Fill**: The content is resized to fill the **Viewbox** control without preserving its original aspect ratio.
- **Uniform**: The content is resized to fill the **Viewbox** control while preserving the original aspect ratio. This is the default setting.
- **UniformToFill**: The content is resized to fill the **Viewbox** control while preserving the aspect ratio, though in this case content is clipped in order to make sure the content fills the entire **Viewbox** surface.

The direction in which content is resized in the **Viewbox** can also be limited to **UpOnly**, **DownOnly**, or **Both** directions through the **StretchDirection** property. **Both** is the default setting.

```
<!-- Viewbox resizing shape and text content. -->
<Viewbox Height="200" Stretch="Uniform" StretchDirection="Both">
    <StackPanel>
        <Rectangle Fill="Orange" Width="20" Height="20"/>
        <TextBlock Text="Viewbox"/>
        <Ellipse Fill="Blue" Width="30" Height="20"/>
    </StackPanel>
</Viewbox>
```

Text Controls

There are several controls available for the display and collection of text in Windows Store apps. The controls that are meant strictly for displaying text include the **TextBlock**, **RichTextBlock**, and **RichTextBlockOverflow** controls. Text-based input can be provided with the **TextBox**, **RichEditBox**, and **PasswordBox** controls.

The **TextBlock** control is a stalwart XAML control and is the primary control used to display read-only text in Windows Store XAML apps. Content in **TextBlock**s can be broken up into sections using **Run** elements that can be further separated by **LineBreak** elements.

The **RichTextBlock** control provides additional text formatting options beyond what is provided by the **TextBlock** control, including the ability to include UI elements inline and a model that allows specifying an overflow container that will be used when the text overflows a **RichTextBlock**'s boundaries. Content in a **RichTextBlock** is contained within **Paragraph** elements. Within these elements, an **InlineUIContainer** can be included that allows XAML markup elements to flow with the displayed text. Finally, the **RichTextBlock**'s **OverflowContentTarget** property allows the name of a **RichTextBlockOverflowControl** to be bound. This control will be used to display any text that overflows the original container's boundaries. The **RichTextBlockOverflowControl** also includes an **OverflowContentTarget** property that allows the overflow containers to be chained to whatever depth is desired.

The following code shows the **TextBlock** and **RichTextBlock** controls used to display various text combinations.

```
<!-- TextBlock with mixed text displays provided by Run elements. -->
<TextBlock>
    <Run Foreground="Green">
        This text is green...
    </Run>
    <Run Foreground="Yellow">
        and this is yellow...
    </Run>
    <LineBreak/>
    <Run>
        And this is a new line.
    </Run>
</TextBlock>
```

```
<!-- RichTextBlock with overflow and an inline image. -->
<RichTextBlock OverflowContentTarget="{Binding ElementName=FirstOverflowContainer}"
               Width="60" Height="40" HorizontalAlignment="Left">
    <Paragraph>
        <Bold>
            <Span Foreground="Green">This text is green and bold...</Span>
        </Bold>
        and this text is neither.
    </Paragraph>
    <Paragraph>
        This text is a new paragraph
        <InlineUIContainer>
            <Image Source="ms-appx:///Assets/query.png" Height="20"/>
        </InlineUIContainer>
        with an inline image.
    </Paragraph>
</RichTextBlock>

<!--Since the block above is thin/short, text will overflow between “and” and “this”.-->
<RichTextBlockOverflow x:Name="FirstOverflowContainer"/>
```

The **TextBox** control is used to enter and edit unformatted text. It can be made read-only, and can also be set to display either as a single-line or as a multiple-line control. The **RichEditBox** similarly allows text input, and includes support for formatted text, hyperlinks, and images. Both the **TextBox** and **RichEditBox** optionally allow enabling spell checking (including autocorrect) and text prediction (autocomplete). They both also allow **InputScope** values to be specified. **InputScopes** provide “hints” that provide guidance as to which layouts should be displayed by the on-screen keyboards that appear when a user taps into a text editing control. These hints are defined in the **InputScopeNameValue** enumeration. Commonly used values include:

- **Url**: The keyboard includes additional symbols that are useful for entering URL data. For example, a **/** button and a **.com** button are added near the Spacebar, and Enter reads “Go”.
- **EmailSmtpAdddress**: The keyboard includes additional symbols useful for entering URL data, such as an @ symbol and a **.com** button.
- **Number**: The keyboard is displayed in numeric mode by default.
- **TelephoneNumber**: Same as numeric.
- **Search**: The Enter key is changed to read "Search".

Note: The `InputScopes` are only used when the on-screen keyboard is displayed. They are meant to provide guidance to Windows to set the on-screen keyboard’s initial display to a mode that will help users accomplish whatever the current edit control is intended to do. `InputScopes` are neither meant to be nor do they function as a validation mechanism; they do not restrict keyboard entry into the field.

The following code sample demonstrates the use of a **ComboBox** to display several common input scopes and bind the selection to configure a **TextBox** to show that scope.

```
<!-- Allow selection of some common InputScope values. -->
<ComboBox x:Name="InputScopeCombo" Margin="0,0,0,10">
    <x:String>Default</x:String>
    <x:String>Url</x:String>
    <x:String>EmailSmtpAddress</x:String>
    <x:String>Number</x:String>
    <x:String>TelephoneNumber</x:String>
    <x:String>Search</x:String>
</ComboBox>
<!-- Apply the selected InputScope value via binding. -->
<TextBox InputScope="{Binding SelectedItem, ElementName=InputScopeCombo}"/>
```

The final text control to discuss is the **PasswordBox** control. As its name implies, this control is typically used for entering password information. The control allows text to be entered, but the control itself displays a mask character instead of the actual typed value. The **IsPasswordRevealButtonEnabled** property indicates if a button will be included in the control that allows users to toggle the visibility of the text they entered. The values entered by users are available through the control's **Password** property.

```
<PasswordBox x:Name="PasswordBoxDemo" IsPasswordRevealButtonEnabled="True"/>
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Password: "/>
    <TextBlock Text="{Binding Password, ElementName=PasswordBoxDemo}"/>
</StackPanel>
```

The Image Control

The **Image** control is used to display bitmap content in Windows Store apps. It provides a **Source** property that designates what is to be displayed in the control. The **Source** is set via an instance of the **ImageSource** class, though in XAML, specifying the **Source** property with a **String** URI will implicitly create a **BitmapImage** object—which inherits from **ImageSource**—with the corresponding URI.

In addition to specifying HTTP-based URIs for the **BitmapImage UriSource**, Windows Store apps support several special URL schemes that can be used to specify content included in the app package or stored in the app's local settings folders. These options will be discussed in more detail in the next chapter, and additional information can be found online at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/Hh965322.aspx. Since the content obtained from the **UriSource** is loaded asynchronously, invalid URIs do not result in exceptions, but instead cause the **BitmapImage** to raise an **ImageFailed** event, which is echoed through the same event exposed by the **Image** control.

There are several options for how bitmap content is scaled to fit the dimensions of the **Image** control in which it is being displayed, which is determined by the value of the **Image** control's **Stretch** property. A value of **None** indicates the source image is not resized. **Fill** resizes the source image to fill the **Image** control without preserving aspect ratio. **Uniform** fills the control as best as possible, maintaining the original image's aspect ratio. Finally, **UniformToFill** also preserves the source image's aspect ratio, but will ensure the entire **Image** control is filled, clipping the source image where necessary. Furthermore, Windows Store apps can specify **NineGrid** rendering values, which are used to specify how different segments of an image are stretched rather than allowing the entire image to be proportionately stretched.

The following markup features several different options for specifying the layout of content to be included in an **Image** control. The first option passes a URL to a web-based image (note that the app will need to have the **Internet Client** capability declared in order to access a web-based image). The second option shows the **Image** control configured to load content that's been included in the Windows Store app project. The third loads an image from the application's app data store (app data and storage will be discussed in the next chapter). The final option explicitly specifies the **ImageSource** as a **BitmapImage** instance. This allows the **BitmapImage** **DecodePixelHeight** to be specified, which restricts the size of the image loaded into memory as opposed to loading the full image into memory and resizing the display (**DecodePixelWidth** is also an option, and specifying one but not the other allows the omitted value to be dynamically calculated).

```
<!-- Reference a web-based image source. -->
<Image Source=
    "http://www.syncfusion.com/Content/en-US/Downloads/Images/ebooks/ebookbanner.png"/>
<!-- Reference an image that was packaged with the project as content. -->
<Image Source="ms-appx:///Assets/Logo.png"/>
<!-- Reference an image in the current application data folder. -->
<Image Source="ms-appdata:///Image.png"/>
<!-- Specify decoding information for the image to reduce memory impact. -->
<Image Height="75">
    <Image.Source>
        <BitmapImage UriSource="ms-appx:///Assets/Logo.png" DecodePixelHeight="75"/>
    </Image.Source>
</Image>
```

Note: It is also possible to populate an Image control with an image built dynamically using an instance of the `WriteableBitmap` class as the `ImageSource`. The `WriteableBitmap` allows access to an image's underlying pixels through its `PixelBuffer` property, allowing the bitmap to be written to and updated.

The WebView Control

The **WebView** control allows embedding a web browser window within a Windows Store app. The browser technology used is supplied by the IE10 browser, though some features in the HTML 5 spec are not supported within these windows. The **WebView** can be instructed to navigate to a URI with the **Navigate** method or can be given a string of HTML to process with the **NavigateToString** method.

```
<!-- Gather a URL to display in a WebView control. -->
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <TextBox Grid.Column="0" x:Name="BrowserURL"/>
    <Button Grid.Column="1" Content="Go" Click="HandleBrowseToURLClick"/>
</Grid>
<WebView x:Name="WebViewDemo" Height="200"/>


// Code-behind event handler.
private void HandleBrowseToURLClick(object sender, RoutedEventArgs e)
{
    // Code to validate the supplied URL omitted for brevity.
    var targetUrl = new Uri(BrowserURL.Text);
    WebViewDemo.Navigate(targetUrl);
}
```

The ProgressRing Control

The **ProgressRing** control is used to communicate to users that an operation is ongoing, similar to the **ProgressBar** used in previous Windows UI development environments. Using the **ProgressRing** for operations whose duration is indeterminate instead of the **ProgressBar** control will provide a look and feel that is consistent with that of other Windows Store apps.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <!-- Use a ToggleSwitch to turn a ProgressRing on and off. -->
    <ToggleSwitch Grid.Column="0" x:Name="ProgressRingSwitch"
                  Header="Progress Ring Sample"/>
    <ProgressRing Grid.Column="1" Visibility="Visible" Width="70" Height="70"
                  IsActive="{Binding IsOn, ElementName=ProgressRingSwitch}" />
</Grid>
```

Using Popups

One of the characteristics of user interface design in Windows Store apps is a departure from the use of modal dialogs for simple data collection or command presentation. Modal dialogs present a sharp interruption in the workflow of an app, and generally tend to stand in contrast to the idea of a fluid user interface.

In order to support a lightweight UI for these activities when dedicating an entire UI page would be excessive, Windows Store apps can make use of the **Popup** and the **PopupMenu** classes. Both of these classes feature the ability to be displayed in a light-dismiss mode, meaning that the control will be automatically closed if users tap or click outside of its boundaries.

The Popup Control

The **Popup** control can be used in Windows Store apps to host custom user interface elements as a "flyout" display. The **Popup** control can host a single element in its **Child** property, which is usually implemented as a custom **UserControl** into which additional UI elements are placed. The **Popup** control instance can then be programmatically opened or closed using its **IsOpen** property, as well as set to close automatically by setting the control's **IsLightDismissEnabled** property to **true**. Additionally, the control can raise **Opened** and **Closed** events at the corresponding times.

Note:** Popup **control contents should not generally rely on the** Closed **event to set application properties. Instead, changes should be committed automatically as they are entered. If the content being entered is data used as input for a particular action, a button can be included on the** Popup **control to trigger the action with the data collected in the control, closing the control when the action is triggered.

The following code shows a **Popup** control being created to host an instance of the **DemoPopupContentControl** class. For an optimal user experience, the popup control should be positioned close to where the action that invoked it occurred. It includes setting handlers for the **Opened** and **Closed** events, adds the control to the page's **LayoutRoot** container (see the tip following the code sample), and finally displays the **Popup**.

```
// Create the popup.
var popup = new Popup()
{
    Child = new DemoPopupContentControl(),
    HorizontalOffset = ctrlLeft,  // Distance to left side of app window.
    VerticalOffset = ctrlTop,  // Distance to top of app window.
    IsLightDismissEnabled = true,
};

// Event handlers for Opened and Closed events.
popup.Opened += HandlePopupOpened;
popup.Closed += HandlePopupClosed;

// Add the popup to the page's layout root to accommodate the on-screen keyboard.
LayoutRoot.Children.Add(popup);
```

```
// Display the popup.
popup.IsOpen = true;
```

Tip: If the** Popup **control being displayed contains any text entry controls, it is normally a good idea to add the** Popup **control to the current page's visual tree before the pop-up is displayed. When the control is parented to a visual element within the page, it will automatically be moved up on the page to accommodate the appearance of the on-screen keyboard if and when it is invoked by a touch action within one of the text edit controls. Adding the** Popup **control to the page's visual tree can occur by including the** Popup **definition within the page's XAML declaration, or by adding the** Popup **element programmatically as a child to one of the panel controls contained within the page, such as the root layout panel.

The PopupMenu Control

The **PopupMenu** class provides a quick way for showing a list of up to six commands that users can select from. Spacers can be inserted between commands, but they are included in the six item cap. Command items in **PopupMenu** controls are added with **UICommand** class instances which can be configured with a display label, a command to be executed when the command is invoked, and an object that can be used to identify the command or, since it is an **Object**, perhaps carry a data payload to be used when the command is executed.

PopupMenu controls are displayed through either **ShowAsync** or **ShowForSelectionAsync**. **ShowAsync** accepts a **Point** parameter which provides a location for the lower baseline of the **PopupMenu** display. Alternatively, the **ShowForSelectionAsync** function takes a **Rect** and an optional **Placement** parameter which dictates on which side of the **Rect** the popup should be aligned (the default is **Above**).

A **PopupMenu** is dismissed either by selecting a command or by clicking outside the surface of the control. In addition to executing the selected command's callback function (or as an alternative if one is not provided) the **ShowAsync** and **ShowForSelectionAsync** functions return the selected command when the control is dismissed. If the control is dismissed via light-dismiss, the calls will return a value of null. The construction and use of a **PopupMenu** is shown in the following code:

```
// PopupMenu - up to 6 items - can be UICommands or UICommandSeparator()
var popupMenu = new PopupMenu();
popupMenu.Commands.Add(new UICommand("Show a Command");

// Show a separator.
popupMenu.Commands.Add(new UICommand("Then Show a Separator"));
popupMenu.Commands.Add(new UICommandSeparator());

// Show an option that invokes a Message Box.
popupMenu.Commands.Add(new UICommand("Show a Message Box", async command =>
    {
        var dialog = new MessageDialog("Message Box Contents", "Message Box Title");
```

```
        await dialog.ShowAsync();
    }));

// In addition to the selected command's callback (if it has one), the Show
// command will either return the selected item or null if dismissed without a command.
var result = await popupMenu.ShowForSelectionAsync(GetRect(sender), Placement.Default);
```

MessageDialog

No discussion of user interface elements would be complete without mentioning the venerable message box control. A message box is a modal dialog meant to communicate urgent information to users, display error information, or ask a critical question that users must answer in order to proceed with the application's workflow. Because of the emphasis on not impeding a user's ability to interact with Windows Store apps, message boxes should be used sparingly; pop-up content that features light-dismiss functionality is preferred over the modal nature of the message box, except for the specific circumstances just mentioned.

In Windows Store apps, the **`MessageDialog`** control is used to show a message box.

```
// Show a message box.
var dialog = new MessageDialog("Message Box Contents", "Message Box Title");
await dialog.ShowAsync();
```

Working with Pages

This chapter began by showing an example of a simple **`Page`** control as a basis for hosting UI content in a Windows Store app. This section will discuss the concepts related to the various layout modes that should be supported in Windows Store apps, as well as the mechanism that is provided for Windows Store apps to navigate between pages.

Layout and View States

As mentioned in the first chapter, Windows Store apps can be presented in several different viewing modes. An app that takes up the whole screen in landscape orientation is said to be running in fullscreen-landscape mode. When the display is rotated, the app is then turned to run in fullscreen-portrait mode. When the hardware is in a landscape orientation, there is also the ability to have two Windows Store apps on the screen simultaneously, with one app taking up a reduced sized display that is fixed at 320 pixels wide. This reduced display mode is referred to as snapped mode. When one app is running in snapped mode, the other app running alongside it that uses the remainder of the display area (the primary app) is said to be running in filled mode.

Note: Snapped mode is only available for displays that have a horizontal resolution of at least 1,366 pixels. Smaller display resolutions—such as the venerable 1024 × 768 resolution—will not allow applications to be positioned in snapped mode. Also note that there is no support for snapped mode when the system is in portrait orientation. When

apps are being displayed in snapped and filled mode, switching the orientation to portrait mode results in the filled app transitioning to fullscreen-portrait, while the snapped app is suspended (suspension will be covered in a later chapter). When the system is rotated back to a landscape orientation, the view will be returned to display the filled and snapped app pair.

While these various display modes provide end users with added functionality in terms of being able to orient their devices to suit their needs, as well as being able to snap an app to the side to have convenient access to one app alongside another, this added experience does mean that developers need to account for these additional display modes. This can be accomplished with a combination of the facilities provided by the **`LayoutAwarePage`** class and the visual state manager, which was discussed previously in this chapter.

The LayoutAwarePage Class

Earlier sections in this chapter have mentioned some additional content that is included when a Windows Store app is built using project templates other than the Blank App template. This content is also optionally provided when any of the page types other than Blank Page are added to an existing Visual Studio project. Regardless of the mechanism used to acquire the content, the content itself is placed into a **Common** folder within the project. Among the items included in this folder is the **`LayoutAwarePage`**.

As its name implies, the **`LayoutAwarePage`** class includes built-in support for layout changes. With the exception of the Blank Page, any of the default pages added from the Visual Studio **Add New Item** dialog are derived from the **`LayoutAwarePage`** and are inherently set up to support reacting to layout change events. The **`LayoutAwarePage`** provides this support by listening to **`SizeChanged`** events on the current application window. When these events occur, it determines the application's current **`ViewState`** by retrieving the **`ApplicationView.Value`** property, which returns a member of the **`ApplicationViewState`** enumeration—each value of which maps to one of the portrait, landscape, etc., view modes discussed previously. It then calls **`VisualStateManager.GoToState`** for the current page with the corresponding state, triggering **`VisualState`** animation storyboards corresponding to the view states, and allowing the application UI to be tailored to the current view state.

While that process may sound a little convoluted, it is all managed within the internals of **`LayoutAwarePage`**. The only work necessary for a page to react to layout changes is for the page to provide the appropriate **`VisualState`** definitions that correspond to the states where custom layout is desired.

Providing Custom Markup for VisualStates

When a **`LayoutAwarePage`** is created, its XAML markup will include a **`VisualStateManager.VisualGroups`** element within the root grid element, which includes a **`VisualStateGroup`** element named **`ApplicationViewStates`** and **`VisualState`** entries for each of the four available **`VisualStates`**. In the simplest case (e.g., the Basic Page type), these states will only have storyboards defined for the **`FullScreenPortrait`** and **`Snapped`** views, which address changes in the positioning of the page's **Back** button and its title.

These **VisualState** declarations can be extended to provide additional UI changes needed to support the different view modes for Windows Store apps. For example, it is very typical (and part of the default Grouped Items Page type) to include both **GridView** and **ListView** controls on a page and to use these states to toggle the **Visibility** property values of these controls in response to switching to a snapped mode where the vertical **ListView** is displayed in snapped mode and the **GridView** is used in the others.

The following markup shows this process in action, where **TextBlock** controls indicating the current state are enabled or disabled based on the current **ViewState**. Note that the **Visibility** of the **TextBlock** controls is initially set to **Collapsed**, and the respective **ViewStates** provide overrides for this value that are only applied as long as the page is in that particular state. When the **ViewState** changes, the override is released and the **Visibility** property returns to its previous inherent value. This is due to the hierarchical value resolution provided by dependency properties discussed earlier in this chapter.

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="TextBlock" BasedOn="{StaticResource SubheaderTextStyle}">
            <Setter Property="Visibility" Value="Collapsed"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock x:Name="FullScrLandscapeTxt" Text="Enabled in Fullscreen Landscape"/>
    <TextBlock x:Name="FilledTxt" Text="Enabled in Filled"/>
    <TextBlock x:Name="FullScrPortraitTxt" Text="Enabled in Fullscreen Portrait"/>
    <TextBlock x:Name="SnappedTxt" Text="Enabled in Snapped"/>
</StackPanel>


<!-- Visual states reflect the application's view state. -->
<VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape">
        <Storyboard>
            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="FullScrLandscapeTxt"
                                           Storyboard.TargetProperty="Visibility">
                <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
            </ObjectAnimationUsingKeyFrames>
        </Storyboard>
    </VisualState>
    <VisualState x:Name="Filled">
        <Storyboard>
            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="FilledTxt"
                                           Storyboard.TargetProperty="Visibility">
                <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
            </ObjectAnimationUsingKeyFrames>
        </Storyboard>
    </VisualState>
    <VisualState x:Name="FullScreenPortrait">
        <Storyboard>
            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="FullScrPortraitTxt"
                                           Storyboard.TargetProperty="Visibility">
                <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
            </ObjectAnimationUsingKeyFrames>
        </Storyboard>
    </VisualState>
```

```
    <VisualState x:Name="Snapped">
        <Storyboard>
            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="SnappedTxt"
                                           Storyboard.TargetProperty="Visibility">
                <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
            </ObjectAnimationUsingKeyFrames>
        </Storyboard>
    </VisualState>
</VisualStateGroup>
```

Previewing Orientation-Specific Layout in Visual Studio

The orientation-specific layouts that are defined through **`ViewState`** values can be previewed in Visual Studio at design time. This can be accomplished through the **Device** panel, which is typically displayed as a tab adjacent to the designer's **Toolbox** panel. The **Device** panel includes a **View** option which, when selected, will toggle the design surface to display the indicated view state. When selected, the **Enable state recording** check box will also apply changes made in the design surface to the current view state. The designer showing the snapped view state is shown in the following figure:

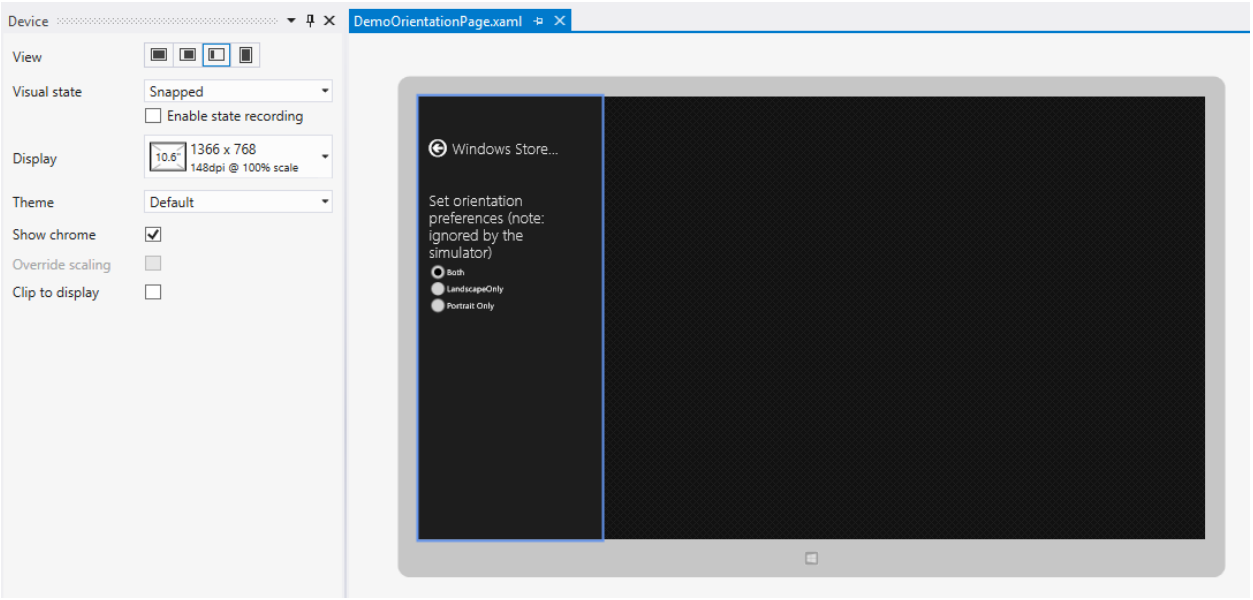


Figure 9: The Visual Studio Designer Showing the Snapped View State

Suppressing Orientation Changes

It is possible to suppress orientation changes in a Windows Store app. This is useful when the data being displayed is strongly tied to a particular layout—examples include games built exclusively for portrait orientation or data reporting presentations that conform to a specific document layout. While many of the available hardware devices support screen orientation lock controls, Windows Store apps are also able to indicate which orientations they support.

There are two mechanisms for setting the supported orientations in Windows Store apps. First, the preference can be set for the entire app through the app manifest file. The **Application UI** tab provides **Supported rotations** settings which allow selection between landscape, portrait, landscape-flipped, and portrait-flipped orientations. If none of the values are selected, all values are assumed to be supported. These options are pictured in the following figure:

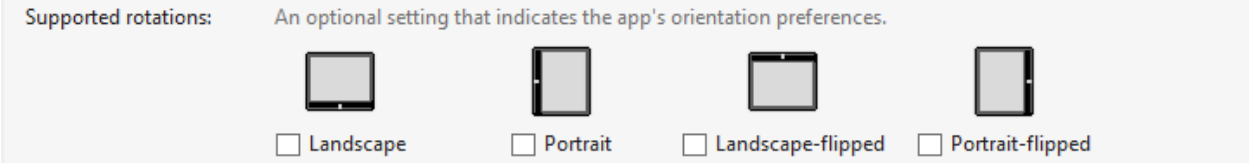


Figure 10: Indicating Supported Rotations in the App Manifest

In addition to making an app-wide selection, supported rotations can be set programmatically, allowing a supported rotation to be set for a given page. This is accomplished by setting the **`DisplayProperties.AutoRotationPreferences`** property to the desired combination of the **`DisplayOrientations`** enumeration. The following code shows the device's rotation preferences being set to only support standard and inverted landscape. The original value is obtained so that at some later point in time (perhaps when leaving the page) the original settings can be reapplied.

```
// Store the initial rotation preferences.
var originalRotationPreferences = DisplayProperties.AutoRotationPreferences;
// Set the display to only support being rotated in the landscape orientation.
DisplayProperties.AutoRotationPreferences =
    DisplayOrientations.Landscape | DisplayOrientations.LandscapeFlipped;

// Some time later, reset the rotation preferences to their original values.
DisplayProperties.AutoRotationPreferences = originalRotatinPreferences;
```

Note: While it is possible to indicate a preference for landscape or portrait views in an app, it is not possible to prevent displaying an app in snapped mode. It is also important to be aware of the fact that the simulator ignores orientation preferences.

Additional Considerations

In addition to handling whatever repositioning and/or restyling is necessary for on-screen controls when the application view is changed, the application's app bar should also be considered. In portrait and snapped display modes, the app bar has considerably less horizontal space in which it can display commands.

The **AppBarButtonStyle** provided in StandardStyles.xaml does include visual states to help provide a better fit in the **FullScreenPortrait** and **Snapped** view states. However, by default, the app bar buttons are not connected to the **ViewState** changes. The **LayoutAwarePage** provides a registration and unregistration mechanism that will allow these buttons (and any other controls for that matter) to be notified of and react to view state changes.

To enlist a control like an app bar button in these notifications, the control's **Loaded** and **Unloaded** events should simply be configured to call the page's **StartLayoutUpdates** and **StopLayoutUpdates** event handlers. In the case of the app bar, this will reduce the size of the button and remove the label when in fullscreen-portrait or snapped modes, providing more space to include commands.

```
<!-- Register the app bar button with StartLayoutUpdates and StopLayoutUpdates. -->
<Button Style="{StaticResource AddAppBarButtonStyle}"
        Loaded="StartLayoutUpdates"
        Unloaded="StopLayoutUpdates"/>
```

Note: Depending on the number of commands involved, simply resizing the app bar buttons may not be enough to accommodate all of the commands when switching to snapped mode. Common techniques for resolving this situation include selectively reducing the number of commands being displayed when in snapped mode or splitting the contents of an app bar into two separate rows. The correct approach will vary depending on the specific application functionality that is needed.

Page Navigation

In Windows Store apps, pages are hosted within an instance of the **Frame** control. The **Frame** control coordinates navigation between pages, including recording and managing navigation history. When a Windows Store app is created, one of the first things that happens is a **Frame** is created and set to the current window content. Usually this occurs in the **OnLaunched** method defined in the app's **Application** class.

Invoking Navigation

Navigation is triggered by calling the **Frame** control's **Navigate** method and providing the type of the page to be navigated to, along with an optional parameter to be passed to the destination page instance. Additionally, a page can move to its predecessor in the navigation stack by first checking if there is an available entry through the **Frame** control's **CanGoBack** property, and then performing the navigation through the **GoBack** method. When a **Page** instance is created as a result of a **Navigate** call, the page's **Frame** property is set to the containing **Frame**, making it easy to access in order to initiate navigation from within a **Page**. The following code shows typical ways in which navigation is invoked within a Windows Store app page.

```
// Navigate to an instance of the DemoNavigationPage without a navigation parameter.
```

```
Frame.Navigate(typeof(DemoNavigationPage));

// Navigate to an instance of the DemoNavigationPage with a navigation parameter.
Frame.Navigate(typeof(DemoNavigationPage), "Navigation parameter");

// Navigate back, if possible.
if (Frame.CanGoBack)
{
    Frame.GoBack();
}
```

Note: Pages based on the `LayoutAwarePage` include a `BackButton` implementation that is wired up to the `Frame` `CanNavigateBack` and `GoBack` members, removing much of the need to handle this boilerplate code. Additionally, the `LayoutAwarePage` provides a `GoHome` method which will navigate back to the first element in the frame's navigation history.

Processing Navigation

When a navigation is requested, the **`Frame`** object coordinates a series of events and method calls that allow information about the navigation activity. This sequence includes:

- The **`Frame`** raises a **`Navigating`** event. This event indicates what kind of navigation is occurring (e.g., **`New`**, **`Forward`**, **`Back`**, **`Refresh`**) and the type of page being navigated to. It also exposes a **`Cancel`** property that allows subscribers to the event to cancel the navigation request at this point.
- If there aren't any subscribers to the **`Navigating`** event that set the **`Cancel`** property, the page that is being navigated away from receives a call into its **`OnNavigatingFrom`** method override with the same arguments that were provided to the **`Navigating`** handler. This provides the current page a convenient opportunity to cancel a navigation request without needing to explicitly subscribe to an event.
- Following the **`OnNavigatingFrom`** override, an instance of the target page is constructed if necessary (see the paragraph following this list).
- Next, the **`Frame`** object raises a **`Navigated`** event. The arguments to this event include a **`Content`** parameter that refers to the instance of the page being navigated to as well as a reference to the navigation parameter, if one was supplied.
- Like the **`Navigating`**/**`OnNavigatingFrom`** tandem, the page being navigated away from can also override an **`OnNavigatedFrom`** method override that receives the **`Navigated`** event arguments.
- The entire process concludes with the target page's **`OnNavigatedTo`** event being called.

By default, navigation in a Windows Store app always tries to create a new instance of the page being navigated to. However, pages in Windows Store apps can opt to participate in the navigation cache. When a page participates in the navigation cache, requests through frame-based navigation for a page of that type will first examine the cache for a matching page. If one is found, it is returned; otherwise a new instance of the page is created and placed in the cache, being available for subsequent requests.

There are two modes in which a page can participate in the navigation cache. First, it can set its **`NavigationCacheMode`** property to **`Required`**, which means that every call will use the cache. The second option is to set the page's **`NavigationCacheMode`** property to **`Enabled`**. In this mode, the navigation cache works with the **`Frame`** control's **`CacheSize`** property. If the frame has cached more pages than the **`CacheSize`** specifies, it drops the oldest page with a **`NavigationCacheMode`** set to **`Enabled`** from the cache in order to make room for the new page. Cached pages with a **`NavigationCacheMode`** set to **`Required`** do not count against the **`Frame`** control's cache size limit.

Recap

This chapter has provided an overview of several fundamental concepts related to developing XAML-based user interfaces for Windows Store apps. As part of this coverage, the building blocks of a user interface—the controls—were reviewed, including some new controls that have been introduced for Windows Store apps, as well as some new ways in which older controls can be used. To provide context related to how these controls will be hosted and how users can switch between user interface displays, the related **`Page`** and **`Frame`** elements were also discussed. While quite a few elements were examined in this chapter, the landscape of XAML user interface design has grown tremendously, and hopefully this chapter has addressed the needs relevant to structuring most Windows Store apps.

The next chapter will introduce the life cycle of a Windows Store app, which relates to when and how a Windows Store app will actually run. The discussion will include coverage of several tools and techniques available for preserving and restoring application state information as well as other types of application data.

Chapter 3 Application Life Cycle and Storage

Windows desktop application developers may find it surprising to learn that there is usually only one Windows Store app running at any given time. When an app is moved from the main display screen and into the background, it enters a suspended state where it ceases to receive any CPU, disk, or network time, and is for the most part completely dormant with no impact on system performance other than its in-memory footprint. In fact, as will be discussed further, the app may be completely unloaded from memory under certain circumstances, including if the OS decides it needs to reclaim that memory. This is a fundamental shift from desktop application development, where it is assumed that end-users will manage application lifetimes. For Windows Store apps, the system takes on that responsibility and it is no longer the end users' concern.

Note: Some readers may wonder about Windows Store apps being displayed side-by-side when one of the apps is displayed in the snapped view. In this case, there are two apps running at the same time, but the concepts presented here still apply, since it is still a very limited number of apps, and an app can be taken out of snapped view by either expanding the primary app to fullscreen-landscape view, or if the hardware supports it, by rotating the device and changing the display to fullscreen-portrait view.

It turns out that in many cases, and perhaps with some small adjustments to some old habits, apps simply don't need to run when users aren't interacting with them. This limited lifetime model actually presents benefits in terms of enhanced battery lifetimes, system responsiveness, and reduced resource contention. This ultimately benefits end users, albeit sometimes at the expense of additional development complexity in order to ensure that these lifetime transitions are as seamless as possible.

With all that in mind, this chapter will discuss the life cycle of Windows Store apps, related events, and other interaction points that developers can use to work within this lifetime model. A key part to doing this is the ability to store and retrieve application state and other related data, so this chapter will then discuss the file storage options available to Windows Store apps, including how to use application data, file and folder pickers, and options available for programmatic file system access.

Life Cycle of a Windows Store App

The following diagram illustrates the lifetime transitions for a Windows Store app:

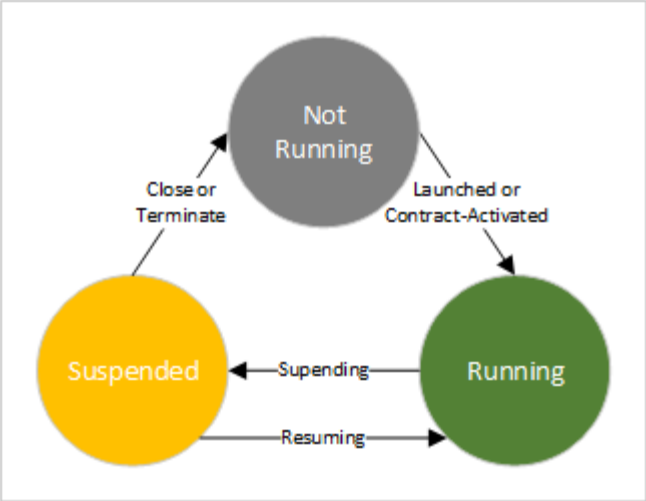


Figure 11: Life Cycle of a Windows Store App

As expected, the app is initially in the **`NotRunning`** state. When the app is launched by a user, Windows starts the process of launching and activating the app. The app's splash screen will be displayed based on the image and background color indicated in the app's manifest file. The app has 15 seconds to complete activation by showing its first window, or Windows may elect to terminate the app (this will also result in the app failing certification to get into the app store).

Note: Contracts and extensions are implementations of standard functionality for interacting with Windows, other apps, or both, which an app can participate in to provide alternate ways in which an app can be invoked. Technically, it is a misnomer to indicate that tile-based invocation is distinct from a contract—it is actually a behind-the-scenes implementation of the Launch contract. Contracts will be discussed further in the next chapter.

Once the app is showing its UI screens, it is in the **`Running`** state. If it is moved to the background as a result of another app being brought to the foreground, it will enter the **`Suspended`** state following a brief pause. The pause allows users time to quickly change their mind and restore the app without incurring any of the expense of handling entry into the **`Suspended`** state. Apps also enter the **`Suspended`** state when the machine is entering standby mode, the user is logging out, or the system is being shut down. In the latter cases, the **`Suspended`** state is being entered on the way to the app being closed or terminated, as will be discussed shortly.

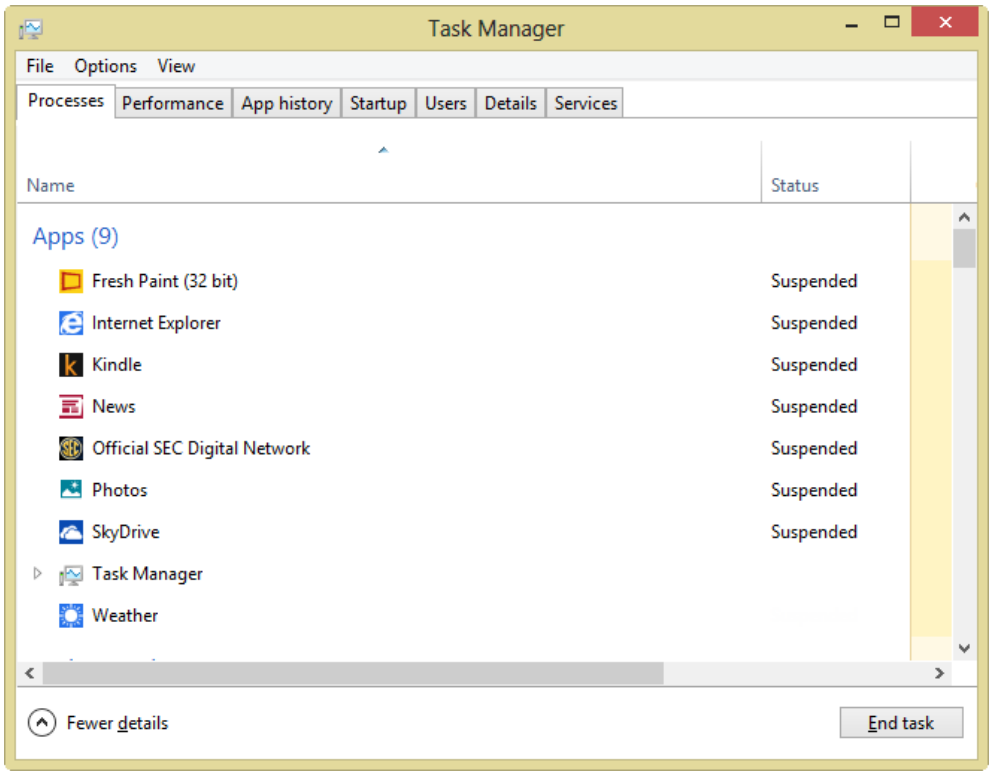


Figure 12: Task Manager Showing Active Weather App and Several Suspended Apps

When an app is in the **`Suspended`** state, it is not scheduled for any CPU time by the OS kernel, all threads are suspended, and no disk or network input/output is consumed. However, the app does remain in memory. Because of this, when a suspended app is invoked, Windows is able to immediately return it to the **`Running`** state. An app has five seconds to handle any activities related to going into the **`Suspended`** mode, or Windows may consider it unresponsive and terminate the app (and again, this circumstance will cause the app to fail the Windows Store certification process.)

There is one more state transition to consider. When an app is closed by the user, or if Windows determines there are too many apps in the **`Suspended`** state and needs to reclaim some memory, the app will be unloaded from memory—a process called termination. The app will not receive any notification that it is being terminated—it simply ceases to be in memory anymore. Running apps being closed by either the user or the OS first enter the **`Suspended`** state on their way to the terminated state, but the process is irrevocable at that point—there is no way to prompt a user and cancel the closing process, and there is no special notification as the app enters the **`Suspended`** state that it is on its way to being terminated.

From these descriptions, it should start to become apparent that it is fairly important to save an app's state before it enters the **`Suspended`** state, since there's no way to know for sure if it will remain in memory before it is launched again. Fortunately, there are some application events and methods that allow an opportunity to do just that.

Application Activation

When an app is activated, the system sends an **Activated** event that includes one of several **ActivationKind** values indicating how the activation occurred. This event is handled by the **Application** class and surfaces through one of several overridable methods; the specific method being called depends on the **ActivationKind** value. In cases where the app is invoked from its start tile—recall that this is a result of an invocation of the Launch contract—the **ActivationKind** is set to **Launch**, and the **Application** object's **OnLaunched** method will be called. Other methods that handle different activation circumstances include **OnActivated**, **OnCachedFileUpdatedActivated**, **OnFileActivated**, **OnFileOpenPickerActivated**, **OnFileSavePickerActivated**, **OnSearchActivated**, and **OnShareTargetActivated**. Several of these other methods will be visited in more detail in later chapters.

The **OnLaunched** method receives a **LaunchActivateEventArgs** value, which includes several useful properties including the **PreviousExecutionState** value. This can be used to determine if the app was previously terminated or if it was closed normally. If the **PreviousExecutionState** value is **ApplicationExecutionState.Terminated**, then the app was previously closed by a system-initiated termination event and it is likely that the application's saved state needs to be loaded to bring the app back to where it was prior to suspension. This can be seen in the default **OnLaunched** implementation provided by Visual Studio:

```
// The OnLaunched method override.
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    // ...code omitted for brevity.
    if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
        //TODO: Load state from previously suspended application.
    }

    // Create the page to use for the Current Window Content and activate it.
    if (Window.Current.Content == null)
    {
        Window.Current.Content = new MainPage();
    }
    Window.Current.Activate();
}
```

When restoring state, remember that the app has 15 seconds to activate its first window, which it signals by calling **Window.Current.Activate**, or it may be terminated by the OS. If restoring state could take longer, it may be better to run code in **OnLaunched** that handles the state restore as a background operation. Using an extended splash screen for this purpose is discussed in a later section.

Note: If the app would benefit from returning to a previous state following system-initiated termination or being explicitly closed by the user, the `ApplicationExecutionState.ClosedByUser` value can be included in the check demonstrated in the previous code sample. Other values for the `ApplicationExecutionState` enumeration and the circumstances that lead to their use

can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/windows.applicationmodel.activation.applicationexecutionstate.aspx.

Application Suspension

The **`Application.Suspending`** event is fired prior to an app being suspended, and allows the app an opportunity to save any state it may need to retrieve in case the app is terminated while it is suspended. It also releases any exclusive resources and file handles to allow other apps access to them while the current app is inactive. The key consideration is that there is no way to know if a suspended app will be terminated, so it must be assumed that it will be.

If some of the operations being called in the **`Suspending`** event handler are asynchronous—as many of the IO operations are—extra work must be done to prevent the **`Supending`** event handler from completing, and to prevent the app from signaling it has finished its work prior to the asynchronous activity being completed. To address this, an object called a **`SuspendingDeferral`**, or simply a deferral, may be obtained from the event handler's provided arguments. When the asynchronous operation completes, the **`Complete`** method on the deferral must be called to indicate that the app is ready to be suspended. Note that even when a deferral is requested, the app still only has five seconds to complete any necessary handling or it may be terminated by the OS. The following code illustrates registering and handling the **`Suspending`** event:

```
// The application class constructor.
public App()
{
    this.InitializeComponent();
    this.Suspending += OnSuspending;
}

// The Suspending event handler with an asynchronous operation.
private async void OnSuspending(Object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background activity.
    await SomeAsynchronousMethod();
    deferral.Complete();
}
```

Resuming From Suspension

Most apps do not need to do anything to recover from simply being suspended. When resumed, the application's in-memory state is right where it was before the suspension. However, if the app is displaying information that is in any way time-sensitive, such as an app that displays periodically updated data (e.g., sports scores or weather information), or an app that displays real-time information (e.g., a stopwatch), it is likely that the correct behavior following a return to the **Running** state is to refresh the application's data in some way, especially since the app could have theoretically been suspended for any amount of time—from just a few seconds to several hours or even longer. To facilitate this, the app can make use of the **Application.Resuming** event. The code for responding to the **Resuming** event follows:

```
// The application class constructor.
public App()
{
    this.InitializeComponent();
    this.Resuming += OnResuming;
}

// The Resuming event handler with an asynchronous operation.
private void OnResuming(Object sender, Object e)
{
    //TODO: Refresh data.
}
```

Handling Long-Running Start-up Activities

There are generally three scenarios for how application content will be displayed to a user when an app is launched:

- The app either doesn't require any significant initial data load or doesn't do anything to defer the data load. Any desired data load is performed prior to calling **Window.Current.Activate**, and the app's main window is fully populated and immediately available for use at this point.
- The app requires extensive or otherwise time-consuming data load or initialization. While the main page is displayed shortly after launch, data for the page is loaded asynchronously and gradually filled into an initially empty or partially complete UI.
- The app requires extensive or otherwise time-consuming data-load or initialization. However, instead of showing an incomplete UI to users while this processing occurs in the background, a copy of the splash screen is first displayed instead of the normal app landing page. Once the data load has been completed, the extended splash screen is dismissed and users are shown a main window fully populated and immediately available for use. This particular approach is known as showing an extended splash screen and will be discussed in this section.

To display an extended splash screen, the app uses the **SplashScreen** API methods to obtain the positioning information about the graphic element on the standard splash screen, which it then uses to create and display a matching window, perhaps augmented with a progress ring or other information. Once the time-consuming initialization has completed, the app displays its landing page.

To set up the splash screen, start with a blank **Page**. Add an **Image** control within a **Canvas** and set the content color to match the app's splash screen color. In the following sample, a **ProgressRing** control is also added to provide some extra feedback.

```
<Grid Background="#FF8000"><!-- Matching color from app splash screen setting. -->
    <Canvas Grid.Row="0">
        <Image x:Name="SplashImage" Source="ms-appx:///Assets/SplashScreen.png" />
        <ProgressRing x:Name="ProgressRing" Width="60" Height="60" IsActive="True"/>
    </Canvas>
</Grid>
```

Next, set up the extended splash screen page's code-behind to set the initial splash element positions to match the original splash screen, as well as to react to orientation changes and other screen resize events.

```
public sealed partial class ExtendedSplash : Page
{
    private readonly SplashScreen _splash;

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

    public ExtendedSplash(SplashScreen splashScreen)
        : this()
    {
        _splash = splashScreen;
        Window.Current.SizeChanged += (o, e) =>
        {
            // This will run in response to view state or other screen size changes.
            UpdateSplashContentPositions();
        };

        // Set the initial position(s).
        UpdateSplashContentPositions();
    }

    private void UpdateSplashContentPositions()
    {
        if (_splash == null) return;
        var splashImageRect = _splash.ImageLocation;
        SplashImage.SetValue(Canvas.LeftProperty, splashImageRect.X);
        SplashImage.SetValue(Canvas.TopProperty, splashImageRect.Y);
        SplashImage.Height = splashImageRect.Height;
        SplashImage.Width = splashImageRect.Width;

        // Position the extended splash screen's progress ring.
        var progressRingTop = splashImageRect.Y + splashImageRect.Height + 20;
        var progressRingLeft = splashImageRect.X + (splashImageRect.Width / 2) -
```

```
                                                                ProgressRing.Width / 2;
        ProgressRing.SetValue(Canvas.TopProperty, progressRingTop);
        ProgressRing.SetValue(Canvas.LeftProperty, progressRingLeft);
    }
}
```

This code sets up a constructor that accepts and stores a reference to the original splash screen, hooks the **`Window.SizeChanged`** event to make a call to update the displayed elements' screen positions in the event of screen resolution or orientation changes, and makes a call to set the initial position values based on the original splash screen's image position.

In the application's **`OnLaunched`** method override, start the long-running start-up task, set the current window to an instance of the extended splash screen which has been provided a reference to the original splash screen, and call **`Window.Current.Activate`**.

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    // Other launched code omitted for brevity…
    DoLongStartup();

    if (Window.Current.Content == null)
    {
        Window.Current.Content = new ExtendedSplash(args.SplashScreen);
    }
    Window.Current.Activate();
}
```

Finally, asynchronously perform the long-running startup task and then navigate to the real landing page for the app.

```
private async void DoLongStartup()
{
    // Simulate the long-running task by delaying for 10 seconds.
    await Task.Delay(10000);
    // On completion, navigate to the real main landing page.
    Window.Current.Content = new MainPage();
}
```

Using the Suspension Manager

As mentioned in previous chapters, Visual Studio provides some helper classes with certain project templates or when some files are added to the project. Among these are the **`LayoutAwarePage`** and **`SuspensionManager`** classes. The **`SuspensionManager`** provides several boilerplate features related to persisting and restoring application state across **`Suspend`**/**`Terminate`**/**`Resume`** sequences. First, the **`SuspensionManager`** works with the **`Frame`** object to which it is associated to save and restore the sequence of pages through which the user has navigated so that the **`Frame`**'s navigation stack can be restored following a termination without requiring developers to implement such a scheme. These classes also work together to provide a simple solution for saving page state across these lifetime events.

To use the **SuspensionManager**, it must be configured when the app is activated in the activation handler method. This configuration involves providing the **SuspensionManager** the **Frame** with which it will be working, and then calling the **RestoreAsync** method when the application is invoked following a termination. The relevant code in a typical **Application.OnLaunched** method is highlighted in the following sample:

```
protected override async void OnLaunched(LaunchActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;
    // Do not repeat app initialization when the Window already has content;
    // just ensure that the window is active.
    if (rootFrame == null)
    {
        //Create a Frame to act as the navigation context and navigate to the first page.
        rootFrame = new Frame();

        //Associate the frame with a SuspensionManager key.
        SuspensionManager.RegisterFrame(rootFrame, "AppFrame");

        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            // Restore the saved session state only when appropriate.
            try
            {
                await SuspensionManager.RestoreAsync();
            }
            catch (SuspensionManagerException)
            {
                //Something went wrong restoring state.
                //Assume there is no state and continue.
            }
        }

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

    if (rootFrame.Content == null)
    {
        // When the navigation stack isn't restored, navigate to the first page,
        // configuring the new page by passing required information as a navigation
        // parameter.
        if (!rootFrame.Navigate(typeof(SomeAppPage)))
        {
            throw new Exception("Failed to create initial page");
        }
    }
    // Ensure the current window is active.
    Window.Current.Activate();
}
```

When suspending, the **SuspensionManager** is asked to save state in a handler for the **Application.Suspending** event.

```
private async void OnSuspending(Object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    await SuspensionManager.SaveAsync();
    deferral.Complete();
}
```

Note: This setup code is provided out-of-the-box with the Visual Studio Grid App and Split App projects. However, the Blank App project type does not include it. If an item is added to a project started with that template, and includes the `SuspensionManager` and `LayoutAwarePage` classes, this initialization code must be explicitly added and configured. Also, if an extended splash screen is used, the relevant code needs to be moved out of the `OnLaunched` method and called after the long-running operation has completed and the app is ready to display its regular UI.

Once the **SuspensionManger** is set up and configured to save and restore state, code can be added to any page that derives from **LayoutAwarePage** to participate in state saving and restoring by overriding the **LoadState** and **SaveState** methods. Each of these methods receives a **Dictionary** object which accepts strings for keys, into which the values are to be loaded and saved, respectively. The following code shows this in the code-behind for a page where the contents of a text box are saved and retrieved:

```
protected override void LoadState(Object navigationParameter,
    Dictionary<String, Object> pageState)
{
    // Retrieve the stored value(s) and reset the control's state.
    if (pageState != null && pageState.ContainsKey("StateTextBox"))
    {
        var text = pageState["StateTextBox"].ToString();
        StateTextBox.Text = text;
    }
}

protected override void SaveState(Dictionary<String, Object> pageState)
{
    // Save the control's state.
    pageState["StateTextBox"] = StateTextBox.Text;
}
```

Note: The content of the `pageState` dictionary is written to disk as an XML file in the app's local application data storage. As a result, care should be taken when including large content like bitmaps. It may be more optimal to exchange a path reference to where such a file may be obtained rather than the file itself.

Background Transfers and Tasks

While it is true that Windows Store apps cannot run unless they are in the foreground, there are situations in which this imposes excessive limitation on the app's functionality. For example, it would be an unreasonable imposition to require a user to keep an app up and running in order to be able to upload or download large files over a network. Other scenarios include monitoring external resources for available content to be retrieved and displayed to users, perhaps in the event they do not have network access the next time they launch the app—an example of this would be detecting and downloading new incoming messages from a central server or service. To address these needs, Windows Store apps have access to a Background Transfer and Background Task infrastructure. The Background Transfer APIs provide Windows Store apps with the ability to upload and download files from both HTTP and HTTPS endpoints, as well as download from TFP endpoints, all while the app is not actively running. Background Task provides a restricted-execution environment in which small work items can be run under system supervision with regards to the resources and amount of CPU time available to the task. These tasks do not run constantly, but instead run as a response to some system event that acts as a trigger for the task. Among other goals, the system management is intended to help prevent CPU or other resource-intensive processes from executing unfettered and depleting system resources such as battery life, or perhaps running up high network bandwidth costs.

Working with Background Transfer and Background Task are advanced concepts whose detailed implementation is beyond the scope of this book. For more information on these topics, please refer to the MSDN documentation. Details for using Background Transfer can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh452975.aspx. Details related to supporting a Windows Store app with Background Task can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh977056.aspx.

Data Storage in Windows Store Apps

While the previous section discussed how to participate in the Windows Store app life-cycle process and the importance of strategically saving and retrieving data for the application, how that data can be saved and retrieved wasn't addressed. This section will explain how various file storage mechanisms available to Windows Store apps can be used to complete that picture.

The data that an app stores and retrieves generally falls into one of two categories: application data, which includes settings and application state information, and user data, which includes data users produce or consume while using the application. In the case of a simple text editor program, the application data may include a user's last selected font value, what document was last opened, and where in that document the cursor was last positioned. The application data may also include the setting chosen for Tab key behavior. For all of these sets of values, the data is likely to be stored in a location and format specific to the app, and will usually be meaningless to any other application. On the other hand, the user data is the actual text file on disk the user is actively editing, which can be opened by any text editor.

Tip: Data storage in Windows Store apps is mainly achieved either through the classes in the `Windows.Storage` namespace or its descendants—primarily the `StorageFile` and `StorageFolder` classes. Locations are usually identified using predefined constants or user interface controls. In some circumstances, a few special URI schemes have been

created which can provide access to certain file locations on disk, including the ms-appx scheme to access files stored inside the application package and the ms-appdata scheme to access files stored in one of the application data locations. Additional information about available URIs can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/Hh965322.aspx. The `StorageFile.GetFileFromApplicationUriAsync` method can take one of these URIs that points to a file and returns a corresponding `StorageFile` reference.

Working with Application Data

Windows Store apps have access to their own set of per-user data stores through the **`ApplicationData`** class. The **`ApplicationData`** instance is exposed to the API as a singleton object available through the **`ApplicationData.Current`** property. Access to three different data stores is provided by this class—local, roaming, and temporary. As its name implies, the local application data store is specific to the machine on which the app is installed. There is no quota for the amount of space avilable in the local store; it is limited by the disk space available where a user's Windows profile data is stored. The roaming data store actually supports replicating data across machines where the user has logged in with his or her Microsoft account (formerly Live ID). There is a maximum limit of 100 KB per app that can be synchronized, and unused roaming data is purged from its cloud storage location after 30 days of inactivity. Finally, while the temporary data store has no size or set expiration limit, its contents may be cleared at any time by system maintenance or if the user selects to clear temporary files from the Disk Cleanup utility.

Note: Data in the `ApplicationData` stores is preserved across application upgrades as new versions of the application become available. However, when the application is uninstalled, these data stores are removed along with the application and their values are lost (with the exception of values stored in the roaming store.)

Local Application Data

Within the local store, there are two ways to access data: the settings container or folders. Settings exposes a hierarchy of named "containers," which in turn may contain other containers (up to 32 containers deep) and key-value pairs. A "simple" value may be up to 8 KB in size, whereas a composite value using the **`ApplicationDataCompositeValue`** class may be up to 64 KB. Settings data is actually stored in a registry file, and values must be base Windows Runtime data types. The list of available types can be found at http://msdn.microsoft.com/en-us/library/windows/apps/br205768.aspx.

The following code demonstrates accessing the settings in the local storage.

```
var localSettingsStore = ApplicationData.Current.LocalSettings;

// Simple settings values.
localSettingsStore.Values["TextValue"] = "Value 1";
localSettingsStore.Values["IntValue"] = 42;
```

```
// Composite value.
var compositeValue = new ApplicationDataCompositeValue();
compositeValue["CompositeValue1"] = "Composite 1";
compositeValue["CompositeValue2"] = 42;
localSettingsStore.Values["CompositeValue"] = compositeValue;

// Using a custom container: data in.
var createdContainer = localSettingsStore.CreateContainer("TestContainer",
                                            ApplicationDataCreateDisposition.Always);
createdContainer.Values["SomeOtherValue"] = "Value 2";

// Using a custom container: data out.
var accessedContainer = localSettingsStore.Containers["TestContainer"];
var value = accessedContainer.Values["SomeOtherValue"];
```

To access data via folders, the **ApplicationData** object's **LocalFolder** property will provide access to the root folder, which is a **StorageFolder** instance. **StorageFolder** is the WinRT type for working with folders in the file system. Keep in mind that most of these methods will be asynchronous.

The following code creates a file, writes data into it, and then opens the same file for data retrieval.

```
var localFolderStore = ApplicationData.Current.LocalFolder;

// Create or open a file in LocalFolder and write some data into it.
var writeFile = await localFolderStore.CreateFileAsync("SampleFile.dat",
                                            CreationCollisionOption.ReplaceExisting);
await FileIO.WriteTextAsync(writeFile, "Some Text");

// Open the file from LocalFolder and read the text out of it.
var readFile = await localFolderStore.GetFileAsync("SampleFile.dat");
var text = await FileIO.ReadTextAsync(readFile);
```

Roaming Application Data

The roaming data store uses the cloud to allow application data to be synchronized across multiple machines where a user has signed in using the same Microsoft ID, enabling a continuous app experience that can span multiple devices. The roaming data store allows settings and folder access just like the local store, except through the **RoamingSettings** and **RoamingFolder** properties instead of their local counterparts. There are a couple of important differences to consider, however:

- The **RoamingStorageQuota** property can be used to see how much storage space is remaining for the app to use.
- The synchronization is not immediate. Under normal circumstances it occurs every few minutes, though network connectivity and latency can affect it. Since the sync can happen at any time, the app can subscribe to the **DataChanged** event to be informed as to when an update has occurred.

- A special setting value named **`HighPriority`** can be created in the **`RoamingSettings`** root that will receive special treatment. It can accept a regular or composite value, but it is limited to 8 KB. This value is synchronized considerably more frequently than other values, nearing instantaneous synchronization in some circumstances.
- There is behavior related to synchronizing versioned data. Application data versioning will be discussed in a subsequent section.

The following code shows how to obtain the roaming store quota, subscribe to data synchronization updates, and write to the **`HighPriority`** setting key.

```
// Obtain the remaining storage space.
UInt64 quota = ApplicationData.Current.RoamingStorageQuota;

// Subscribe to the data change event to be notified when data is synchronized.
ApplicationData.Current.DataChanged += (o, e) =>
{
    // Update the application based on the possibility of new data.
};

// Write to the "high Priority" value.
var roamingStore = ApplicationData.Current.RoamingSettings;
roamingStore.Values["HighPriority"] = "Some high priority value";
```

Tip: Even though `RoamingSettings` are the only settings that automatically raise the `DataChanged` event when data in the store changes, the `ApplicationData.SignalDataChanged` method can be called to explicitly force the event to be raised, regardless of what data stores are being used. This can allow scenarios where one part of the application notifies the other part that an application data change has occurred.

Temporary Application Data

The temporary application data store is similar to the local store, except that it only offers access via folders, which are accessed through the **`TemporaryFolder`** property. As previously mentioned, the data in the temporary store can be removed via system or user-initiated cleanup events, so the data cannot be counted on between application sessions. In most cases, this store is useful for caching data that can be conveniently recovered once removed.

ApplicationData Versioning

The **`ApplicationData`** class includes a mechanism for checking and setting a version identifier for the data, as well as for invoking custom logic to be called when the data needs to be upgraded or otherwise acted upon to accommodate a version change. The mechanisms for accomplishing this include the **`ApplicationData.Version`** property and the corresponding **`SetVersionAsync`** method.

The **Version** property simply allows interrogation of the current version value. **SetVersionAsync** allows specifying a target version number and a callback. After the callback is executed, the **ApplicationData.Version** value will be set to match the value provided to the method. The callback receives a **SetVersionRequest** parameter, which includes **CurrentVersion** and **DesiredVersion** values, so the code executed by the callback can appropriately update the data as necessary.

Clearing ApplicationData Stores

The contents of any one or all of the **ApplicationData** stores can be cleared programmatically via the **ClearAsync** method. The override that doesn't take a parameter simply clears all stores; otherwise, a value from the **ApplicationDataLocality** enumeration can be passed to specify a single store. Note that this enumeration is not a flag-type enumeration, so only one store can be cleared at a time with this particular override. These methods are illustrated in the following sample:

```
// Clear just the local store.
await ApplicationData.Current.ClearAsync(ApplicationDataLocality.Local);
// Clear all stores.
await ApplicationData.Current.ClearAsync();
```

Tip: Used judiciously, clearing `ApplicationData` can be useful in the `Application.OnLaunched` method when the app's `PreviousExecutionState` is not terminated as a technique to prevent any additional restoration code that may be scattered throughout the app from inadvertently restoring the app's state when it shouldn't.

Working with User Data

If storage outside of the application data stores is required for documents or other user data, Windows Store apps have some options available for them. User-initiated access to the file system may occur through file and folder pickers which allow users to select file system locations and where restrictions are applied based on user file system permissions. Based on the locked-down nature of Windows Store apps that has been previously discussed, programmatic file-system access is limited, and what is available outside of the application data stores and some select folders related to application deployment will generally require that application manifest settings be declared.

Using File and Folder Pickers

User-initiated file system access can be invoked via the file and folder picker members of the **Windows.Storage.Pickers** namespace. The main classes which will be used include the **FileOpenPicker**, **FileSavePicker**, and **FolderPicker**.

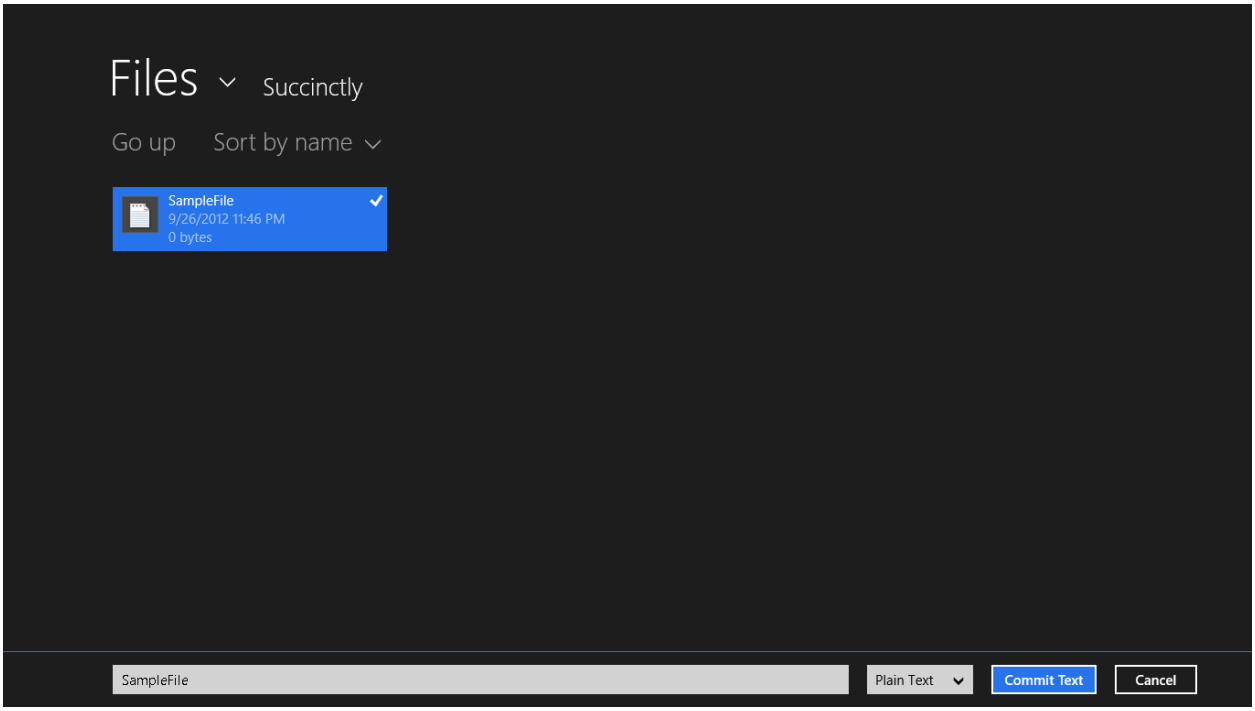


Figure 13: Showing the FileSavePicker Screen

As their names imply, the **`FileOpenPicker`** and **`FileSavePicker`** classes offer access to UI screens that allow users to select a file to open and save, respectively. Similarly, the **`FolderPicker`** presents a UI screen that allows users to select a folder.

Note: The `FileSavePicker` and `FileOpenPicker` actually offer the ability to be extended to browse into an apps' data through the use of the File Open Picker and File Save Picker contracts. An app that integrates with one or more of these contracts appears as a selectable element in the respective file picker's drop-down, and when selected can present UI elements that appear within the picker UI for selecting content. For an example of an app that extends the file pickers in this way, look at the SkyDrive app. These contracts will be discussed further in a later chapter.

For the most part, invocation is very similar for each of these three selection pages. An instance of the class is created, and various properties are set. All three selection pages share the ability to set the text that will appear on the page's **OK** or **Commit** button. They also all share a **`SuggestedStartLocation`**, which accepts a value of the **`PickerLocationId`**, and includes values for several common Windows folder locations. The **`SuggestedStartLocation`** is only used the first time the dialog is used after the app has been launched. After that, the most recent folder selected by the user will be used. If the app uses several pickers for different purposes, the **`SettingsIdentifier`** property allows a **`String`** token value to be associated with the control. Pickers with the same **`SettingsIdentifier`** token will share their recently used folder values. The **`FileOpenPicker`** and **`FolderPicker`** allow setting the display mode for the content between a list and a tile display, based on the **`PickerViewMode`** enumeration value that is selected.

If a user selects a value appropriate to the provided page and taps **Commit**, he or she will either be returned a **`StorageFile`** object for the **`FileSavePicker`** and **`FileOpenPicker`**, or a **`StorageFolder`** object for the **`FolderPicker`**. These values are then used by the application code to work with the selected file or folder as necessary.

```
// Save picker.
var savePicker = new FileSavePicker
{
    SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
    CommitButtonText = "Commit Text",
    SuggestedFileName = "Some file Name",
    DefaultFileExtension = ".txt",
};
savePicker.FileTypeChoices.Add("Plain Text", new List<String>() { ".txt", ".text" });
var chosenSaveFile = await savePicker.PickSaveFileAsync();

// Open picker.
var openPicker = new FileOpenPicker
{
    SuggestedStartLocation = PickerLocationId.ComputerFolder,
    CommitButtonText = "Commit Text",
    ViewMode = PickerViewMode.List,
};
openPicker.FileTypeFilter.Add(".txt");
openPicker.FileTypeFilter.Add(".text");
var chosenOpenFile = await openPicker.PickSingleFileAsync();

// Folder picker.
var folderPicker = new FolderPicker
{
    SuggestedStartLocation = PickerLocationId.Desktop,
    CommitButtonText = "Commit Text",
    ViewMode = PickerViewMode.Thumbnail,
};
// Use "." to show only folders and not display any files.
folderPicker.FileTypeFilter.Add(".");
var chosenFolder = await folderPicker.PickSingleFolderAsync();
```

Note: The `FileOpenPicker`, `FileSavePicker`, and `FolderPicker` cannot be displayed when the app is in snapped view. To be able to display these items, it is important to first check to see if the `ApplicationView.Value` is `ApplicationViewState.Snapped`. If it is snapped, a subsequent call to `ApplicationView.TryUnsnap` can be issued to try to take the app out of snapped mode. Ultimately, if that fails, users should be notified that the desired picker element cannot be displayed. This approach is shown in the next code sample.

```
public Boolean EnsureUnsnapped()
{
    // FilePicker APIs will not work if the application is in a snapped state.
    var unsnapped = ApplicationView.Value != ApplicationViewState.Snapped;
    if (!unsnapped) unsnapped = ApplicationView.TryUnsnap();
    if (!unsnapped)
    {
        // TODO: Notify the user that the picker cannot be displayed since the
        // application cannot be unsnapped.
    }
    return unsnapped;
}
```

Programmatically Accessing Files and Folders

For the most part, file system access for user data should be handled through the file pickers. Most direct programmatic file access will be directed to application data folders and take place through the **`ApplicationData`** API. In a few cases, there may be cause to gain programmatic access to additional locations. As evidenced by the application data discussion in this chapter, there are certain file system locations that all Windows Store apps can programmatically access by default. These include:

Table 1: Locations Windows Store Apps Can Access by Default

Location	Access With
Application Data Directory	ApplicationData.Current.[Local | Roaming | Temporary]Folder Use the “ms-” URI: ms-appdata:///[local | roaming | temporary]/filename.
Application Installation Directory	Windows.ApplicationModel.Package.Current.InstalledLocation Use the “ms-” URI: “ms-appx:///filename”.
User’s Downloads Folder Content	Only files or folders in the user’s Downloads folder that were created by the app. DownloadsFolder.CreateFileAsync(“filename”); DownloadsFolder.CreateFolderAsync(“foldername”);

If the application needs programmatic access to additional locations, the app manifest will need to be modified. In most cases, a **`Capability`** value will need to be set for the specific location where access is needed, and additional file type associations may need to be declared to indicate which file types in the location your app can access.

Table 2: Folders Apps Can Access

To Access	Access With
Music Library	Add the MusicLibrary capability. Use KnownFolders.MusicLibrary to obtain the StorageFolder.
Pictures Library	Add the PicturesLibrary capability. Use KnownFolders.PicturesLibrary to obtain the StorageFolder.
Videos Library	Add the VideoLibrary capability.

To Access	Access With
	Use KnownFolders.VideoLibrary to obtain the StorageFolder.
Homegroup Libraries	Add any one of the MusicLibrary, PicturesLibrary, or VideoLibrary capabilities. Use KnownFolders.HomeGroup to obtain the StorageFolder.
Removable Devices	Add the RemovableDevices capability. Add File Type Associations to declare specific file types to access. Use KnownFolders.RemovableDevices to obtain the StorageFolder.
DLNA Devices	Add any one of the MusicLibrary, PicturesLibrary, or VideoLibrary capabilities. Use KnownFolders.MediaServerDevices to obtain the StorageFolder.
UNC Folders	Add the PrivateNetworkClientServer capability. Add one of the InternetClient or InternetClientServer capabilities. If domain credentials are required, add the EnterpriseAuthentication capability. Add file type associations to declare specific file types to access. Access a folder with StorageFolder.GetFolderFromPathAsync. Access a file with StorageFile.GetFileFromPathAsync.

Note: Programmatic storage access to the Documents library is only available under very specific circumstances, and in general, access to this folder is only available to Windows Store apps through the file pickers. For information on the specific cases where programmatic Documents library access is available, please see http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh464936.aspx.

An example of programmatically accessing the Pictures library to show an image in an app is shown in the following code. Note that for this example, the **`PicturesLibrary`** capability was added to the app.

```
// Locate the well-known folder, assuming the capability has been declared.
var storageFolder = KnownFolders.PicturesLibrary;
if (storageFolder == null) return;

// Locate a subfolder.
var succinctlyFolder = await storageFolder.GetFolderAsync("Succinctly");
if (succinctlyFolder == null) return;

// Locate a specific file.
var succinctlyImageFile = await succinctlyFolder.GetFileAsync("Succinctly.jpg");
if (succinctlyImageFile != null)
{
    // If the picture was found, load it into the on-page image control.
    using (var fileStream = await succinctlyImageFile.OpenAsync(FileAccessMode.Read))
    {
        var bitmapImage = new BitmapImage();
        bitmapImage.SetSource(fileStream);
        selectedImage.Source = bitmapImage;
    }
}
```

Additional Data Storage Options

There are a few other options for data storage that deserve a brief mention. First, the **`FileOpenPicker`** and **`FileSavePicker`** extensions provided by the Windows Store SkyDrive app were briefly mentioned. Beyond local file storage, SkyDrive is an interesting option for storing data in your application's data. To that end, Microsoft provides the Live Connect API for interacting with this and other services from your apps. More information about using the Live Connect API to make use of SkyDrive from within your application can be found at http://msdn.microsoft.com/live.

If it would be helpful for a Windows Store app to maintain its application data in a relational database, a version of the SQLite database engine has been made available that can be used from the Windows Runtime. Information about SQLite can be found at http://sqlite.org/ and instructions for using it from within a Windows Store app can be found at http://timheuer.com/blog/archive/2012/08/07/updated-how-to-using-sqlite-from-windows-store-apps.aspx.

Recap

This chapter introduced the Windows Store application life cycle and showed the related events and methods that can be used to preserve and restore state information based on this life cycle. It also showed how an extended splash screen could be used to provide a pleasant user experience for apps that require prolonged initialization at start up. The chapter then presented the options available to Windows Store apps for data storage, including the **`ApplicationData`** APIs that provide local, roaming, and temporary storage options, and concluded with a discussion of options for accessing the file system to store or retrieve user data either via the **`FilePicker`** members or through direct access to the limited set of folders and circumstances where this is available.

The following guidelines should serve as best practices for saving and restoring state in a Windows Store app:

- Save application settings immediately.
- Save user data incrementally as the app is used.
- When the app is being suspended, record the user's current location and state information.
- When an app is launched, check to see if it was previously terminated. If so, restore the users' session as if they had never left.
- When a suspended app is resumed, refresh any time-sensitive data. Otherwise, do nothing.

The next chapter will explore the contracts and extensions mechanisms that Windows Store apps can participate in to take advantage of enhanced integration with Windows and other Windows Store applications.

Chapter 4 Contracts and Extensions

Windows Store apps introduce the concept of contracts and extensions. Contracts and extensions provide extensibility points that allow a Windows Store app to participate in several common Windows user experience scenarios, such as searching an app for information, sharing data between apps, sending app information to a printer, and providing users with app settings options, among others. In many cases, when an app is built to participate in these contracts and extensions, doing so provides the app with additional ways in which it can be activated by the end user. In fact, as was briefly noted in the previous chapter's discussion about the Windows Store application life cycle, launching an app from its Start screen tile is actually an implementation of the Launch contract.

This chapter will focus on the implementation of several of the more common extensibility scenarios. Initially, this will include the facilities exposed through the Windows 8 charms bar: Search, Share, Device, and Settings. After that, the chapter will explore additional extensions that allow an app to participate in the file picker UI pages, and how an app can be set up to respond to requests to open certain file and URI types.

The Windows 8 Charms

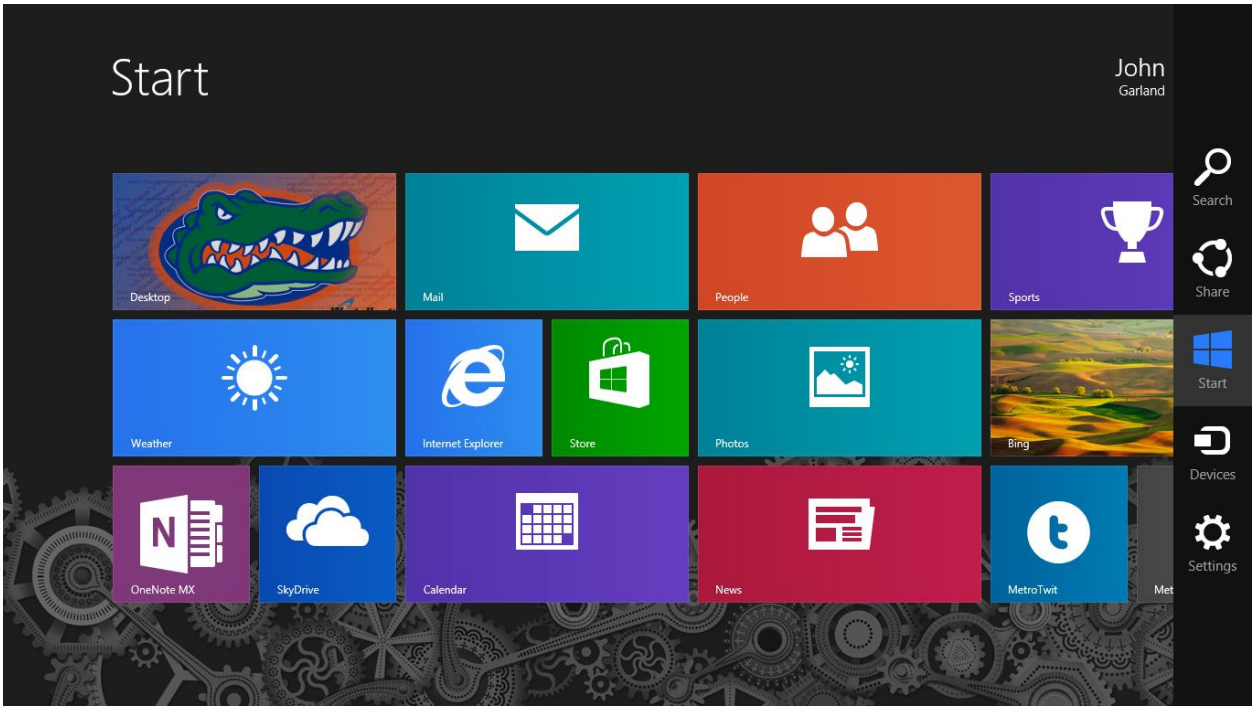


Figure 14: The Windows Start Screen Showing the Charms Bar

The Windows environment includes a new UI element called the charms bar. The charms bar is available on touch devices by swiping from the right side of the screen, and can be accessed on desktops by moving the pointer to the upper-right or lower-right screen corners, or by using the Windows logo key+C keyboard shortcut (remember it as “C for charms”).

There are five charms on the charms bar (top to bottom):

- **Search**: Allows users to search participating applications for content.
- **Share**: Allows users to exchange data from one application to another.
- **Windows**: Not an application-interactive button. Provides an on-screen equivalent to the Windows logo button commonly found on most Windows PC keyboards.
- **Devices**: Allows users to exchange content with hardware devices.
- **Settings**: Allows users to manage application settings.

As mentioned, each of these charms provides your application the ability to participate in an extension of behavior that is built into Windows in a consistent and predictable manner. As application users become more familiar with these Windows 8 features, they will more easily discover common application functionality they would have previously had to find in haphazard ways, like searching for menu names that vary wildly between applications. Fortunately, incorporating these behaviors into a Windows Store app is relatively straightforward, especially when the functionality and additional application exposure that is afforded by doing so is considered.

Searching for Content in an App

A user can bring up the search by tapping or clicking on the Search charm, or by using the Windows logo key+Q keyboard shortcut (remember it as “Q for query”). When the search is invoked, Windows shows a system-provided **Search** pane. This pane includes a text box for entering the search query, an area for the searched application to provide search suggestions, and a list of applications registered to participate in the Search contract. The order of the listed applications is based on application usage.

Participating in Search

The simplest way for an application to be set up to participate in the Search contract is to add the **`Search Contract`** item from the project’s **Add Existing Item** command. This step addresses three of the key steps that are required for search. First, it adds the **`Search`** declaration to the application’s manifest. This declaration is what allows Windows to know that when the application is installed, it should take the necessary steps to set up the application to be included in the list of available searchable apps.

Second, it adds an **`OnSearchActivated`** method override to the **`Application`** class. The **`OnSearchActivated`** method is called when the application is activated via the Search contract instead of the previously discussed **`OnLaunched`** method that is called when the application is launched from the Start screen. The arguments provided to this method include the user’s input language and a **`QueryText`** property, the latter of which contains the text entered in the **Search** text box by the user prior to submitting the query. Users submit queries by tapping **Search** next to the text box or pressing **Enter**.

Note: Just like the `OnLaunched` event, if the application activation is handled through the `OnSearchActivated` method, the app has 15 seconds to display a user interface by calling `Window.Current.Activate`, or Windows may decide to terminate it.

The final change provided is the inclusion of a starter results page for displaying the search results, as well as some UI elements related to filtering them. The **`OnSearchActivated`** method takes care of invoking this page and passing the **`QueryText`** value into it as its navigation parameter. The application logic to obtain and display the results will of course vary depending on the application's actual needs.

Interaction with the displayed search UI is managed through the **`SearchPane`** class, usually by obtaining a reference that is scoped to the current application by calling the static **`SearchPane.GetForCurrentView`** method. The following code illustrates some of the basic functionality that can be accessed through this object to coordinate an app's interaction with search:

```
// Retrieve the SearchPane reference.
var searchPane = Windows.ApplicationModel.Search.SearchPane.GetForCurrentView();

// Set the text to be shown in the Search box if the user hasn't entered any characters.
searchPane.PlaceholderText = "Text To Show";

// Retrieve any characters the user has entered into the Search box.
var queryText = searchPane.QueryText;

// Show the search pane if the user types any characters on the keyboard.
searchPane.ShowOnKeyboardInput = true;

// Show the search pane programmatically without and with indicated text.
searchPane.Show();
searchPane.Show("Pre-entered text");

// Set the search pane text without showing it.
// This is used to keep in sync with in-app search content.
searchPane.TrySetQueryText("Text to sync");

// Event raised when the text in the search pane is updated.
searchPane.QueryChanged += OnSearchQueryChanged;

// Event raised when the search is submitted.
searchPane.QuerySubmitted += OnSearchQuerySubmitted;
```

It is important to note some of the special circumstances related to when the **`OnSearchActivated`** method is called and when the **`QuerySubmitted`** event is raised:

- If the app does not subscribe to the **`QuerySubmitted`** event, submitting a query will simply result in the **`OnSearchActivated`** method being called whether the application is running or not.

- If the app subscribes to the **QuerySubmitted** event and is in the foreground when a query is submitted, the **QuerySubmitted** event handler will be called and the **OnSearchActivated** method will not be called.
- Even if the **QuerySubmitted** event has been subscribed to, if the app is snapped when a query is submitted, the app will be restored to full-screen and **OnSearchActivated** will be called with a **PreviousExecutionState** of **Running** instead of the **QuerySubmitted** event handler.
- Similarly, if the app is in a suspended state when the query is submitted, the **QuerySubmitted** event handler will also be skipped and **OnSearchActivated** will be called with a **PreviousExecutionState** of **Suspended**.

Note that using the **QuerySubmitted** event to handle query requests when the app is running offers a performance benefit over simply allowing **OnSearchActivated** to handle all of the requests. However, because the **OnLaunched** and **OnSearchActivated** methods can be called multiple times within a single application session, registering for the **QuerySubmitted** event in these methods is not advisable, as doing so may result in multiple registrations. As a result, the best place to subscribe to register for the **SearchPane** event handlers is within the **Application** object's overrideable **OnWindowCreated** method, which serves the purpose of being a single, consistent method that can be used for application initialization functions, especially those related to contract-related events:

```
protected override void OnWindowCreated(WindowCreatedEventArgs args)
{
    // Retrieve the SearchPane reference.
    var searchPane = Windows.ApplicationModel.Search.SearchPane.GetForCurrentView();

    // Event raised when the text in the search pane is updated.
    searchPane.QueryChanged += OnSearchQueryChanged;

    // Event raised when the search is submitted.
    searchPane.QuerySubmitted += OnSearchQuerySubmitted;

    base.OnWindowCreated(args);
}
```

Search Suggestions

In addition to providing a space for users to enter query text, the **Search** pane allows the foreground app to supply recommendations that will be shown to users based on the text they have entered. There are two kinds of recommendations that can be shown: query suggestions and result suggestions. Query suggestions are simply text values the application offers as hints for the completed value a user is entering, usually akin to the autocomplete functionality offered in many form-entry applications. When the user selects a query suggestion value, the application treats the value as if the user had typed it directly into the **Search** pane text box, and its entry point into the application is through the **`QuerySubmitted`** event logic as outlined in the previous sample. Result suggestions are more detailed values that are meant to show a specific record that matches the query and include a title, some descriptive text, and an icon. Typically when a user selects a result suggestion, the application navigates directly to the display of the selected record. When a result suggestion is selected, the app **`SearchPane`** object raises a **`ResultSuggestionChosen`** event. A combination of query and result suggestions can be displayed simultaneously, but a total of only five items can be displayed at any given time. There is a mechanism to provide some separator text anywhere in the list, but it will also count against the five-item total.

The following code shows the handling of a request for query suggestions, where up to two query results will be displayed (using a custom **`FindQuerySuggestions`** method to encapsulate whatever custom logic may be appropriate), followed by a separator, followed by up to two suggestion results (similarly using a custom **`FindResultSuggestions`** method for application-specific logic):

```
private void OnSearchSuggestionsRequested(SearchPane sender,
                                          SearchPaneSuggestionsRequestedEventArgs args)
{
    // Typically use the args.QueryText value to determine what selections to display.
    var queryText = args.QueryText;

    // Request a deferral to deal with async operations.
    var deferral = args.Request.GetDeferral();

    // Optionally cancel the request:
    // args.Request.IsCanceled = true;

    // Append the first two suggestion items.
    var querySuggestions = FindQuerySuggestions(queryText).Take(2);
    args.Request.SearchSuggestionCollection.AppendQuerySuggestions(querySuggestions);

    // Add just a single query suggestion:
    // args.Request.SearchSuggestionCollection.AppendQuerySuggestion(suggestion);

    // Insert a separator to distinguish between query and result suggestions.
    args.Request.SearchSuggestionCollection.AppendSearchSeparator("Separator Label");

    // Add the first two result suggestions.
    var resultSuggestionRecords = FindResultSuggestionRecords(queryText).Take(2);
    foreach (var record in resultSuggestionRecords)
    {
        // Note: The image to display should be 40 x 40.
        args.Request.SearchSuggestionCollection.AppendResultSuggestion(
            record.Title,
```

```
            record.Detail,
            record.Tag,
            record.Image,
            record.ImageAltText);
    }

    deferral.Complete();
}
```

In the event that one of the suggestion results is selected, the **ResultsSuggestionChosen** event will be raised.

```
private void OnSearchResultsSuggestionChosen(SearchPane sender,
                                    SearchPaneResultSuggestionChosenEventArgs args)
{
    var selectedItemTag = args.Tag;

    // Given the tag, find the item that matches that tag.
    var matchingItem = FindResultSuggestionItemByTag(selectedItemTag);

    // Do something to display the specific matching item.
    DisplayMatchingItem(matchingItem);
}
```

The previous code sample might result in the following **Search** pane contents:

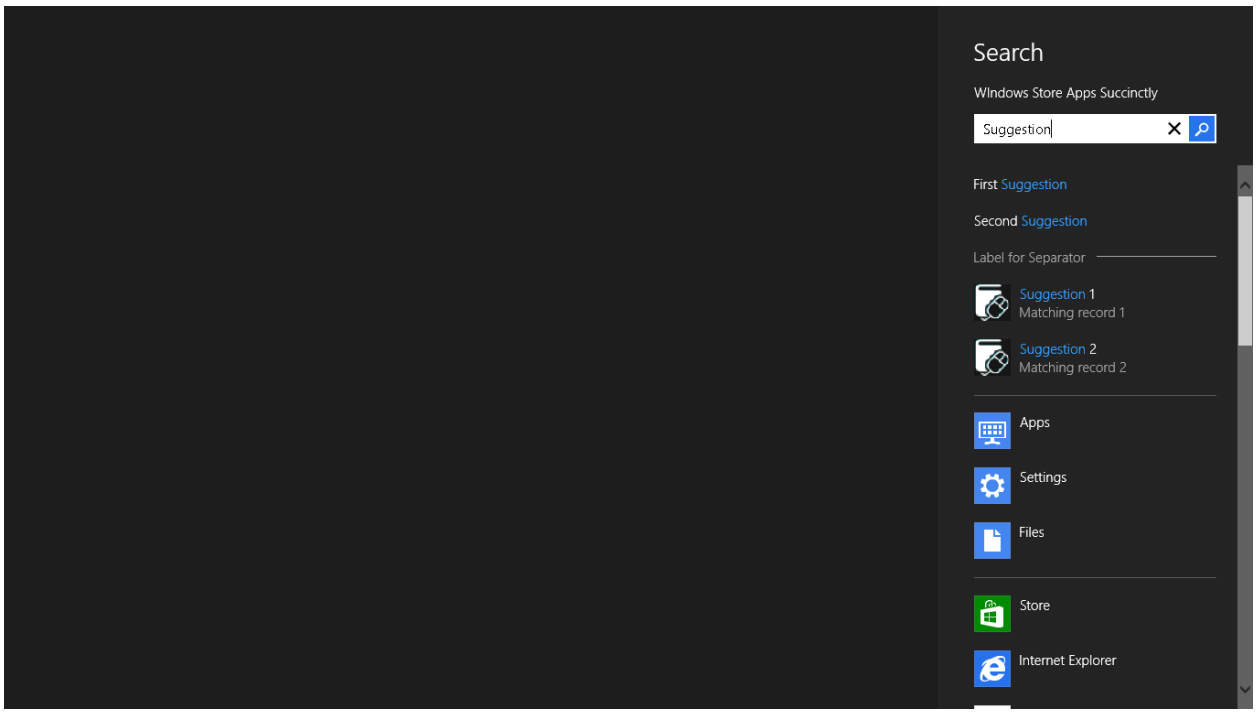


Figure 15: Search Pane with Suggestions

Tip: When a search is invoked while the app is running, debugging is straightforward. However, there is also a facility in Visual Studio that allows debugging apps that aren't currently running by actually waiting for the apps to run before attaching the debugger. This functionality is enabled from within the Debug tab on the Visual Studio project's Properties page. There is a check box next to "Do not launch, but debug my code when it starts" within the Start Action section. If this value is selected when Visual Studio is used to debug an application, Visual Studio will enter its debugging mode, but the app itself won't be launched. When the app is launched—either through a tile or activation through one of the contracts—Visual Studio will then be attached and any breakpoints or other diagnostic tools will be functional for the application. This technique is not limited to just search; it will work for activations related to any of the contracts discussed in this chapter.

Sharing Content between Apps

While Windows Store apps can continue to leverage the Clipboard as a mechanism for sharing data between apps, Windows Store apps have access to a new and more sophisticated system for sharing data that is exposed via the Share contract. This system allows Windows to broker the exchange of data, resulting in applications having the ability to exchange content without needing to be intimately aware of each other's implementation details. This also gives end users the ability to deliberately send information from one Windows Store app to another and interact with user interfaces specifically designed to facilitate this process.

Applications can participate in the Share contract in one of two ways: an app can be a share source, which provides data that can be consumed by other applications, or an app can be a share target, which means that it is capable of receiving data shared by other apps. Note that being a share source does not preclude an app from also registering and being a share target.

Users can share content out of their app by using the Share charm or by using the Windows logo key+H keyboard shortcut (remember it as "H for sHare"). This initiates the share life-cycle by sliding out the **Share** pane. The content types that can be shared by an app include:

- Text
- Richly formatted text
- HTML
- URI
- Bitmaps
- File references (a.k.a. **`StorageItems`**)
- Custom data (including waiting to determine the data until the target actually requests it, which will be discussed in the next section)

Share Life Cycle

The general life cycle followed by a share operation is illustrated in the following figure:

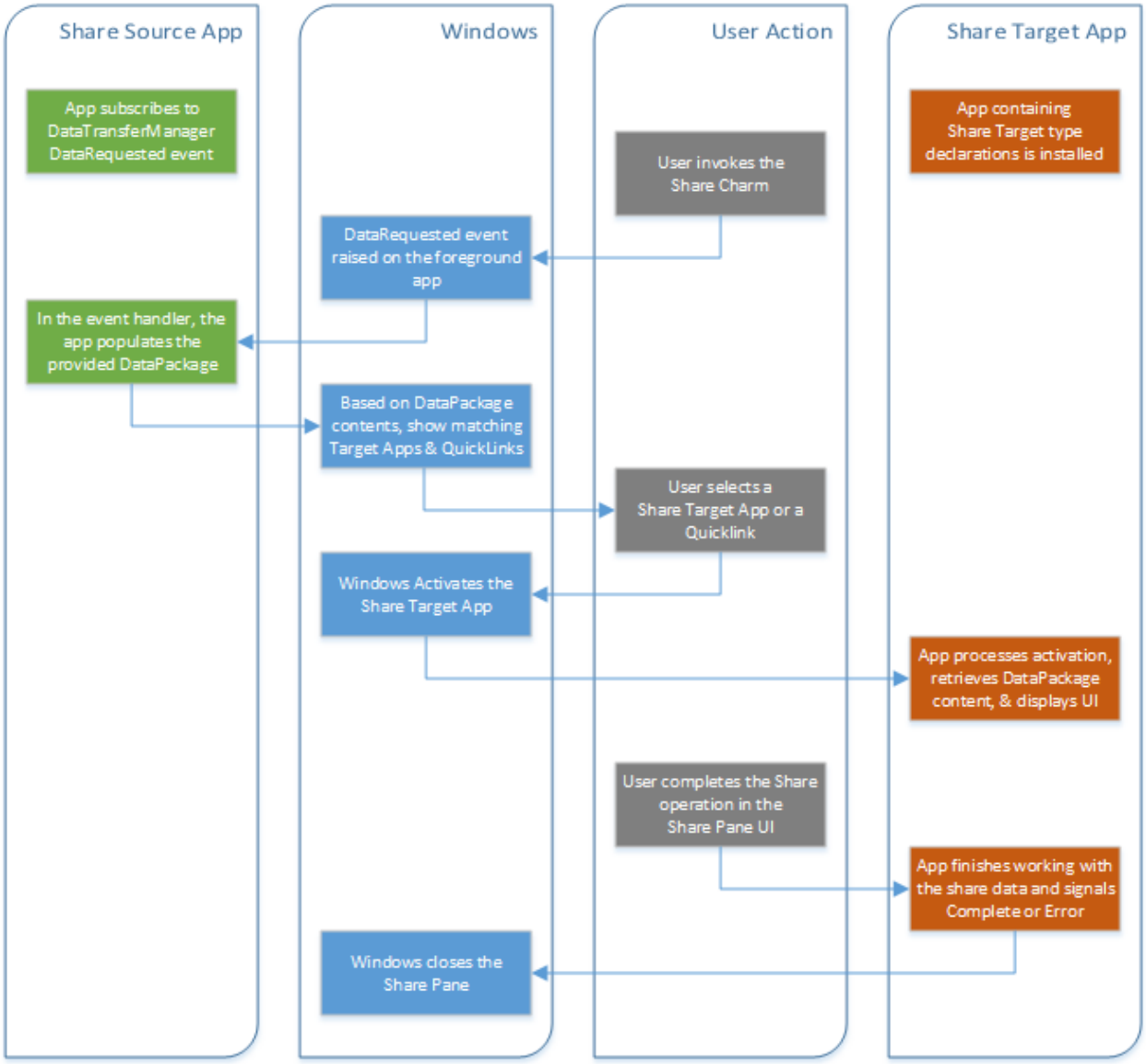


Figure 16: The Life Cycle of Sharing Data between Applications

The steps depicted in Figure 16 include the following:

1. Before users can invoke the Share contract to share data from a share source application, the foreground application must first register its intent to participate by subscribing to the **`DataRequested`** event exposed by the **`DataTransferManager`** class.
2. When users tap the **Share** charm, Windows will raise the **`DataRequested`** event in the foreground application.

3. In the event handler, the foreground application will place the data it intends to share into the data member of the event's arguments, which is a **`DataPackage`** object. The application can put several different data types into the data package at once. In fact, doing so increases the number of applications that can receive the share data, as well as the ways these applications can respond to receiving it.
4. When the event returns, based on the types of data that have been put into the data package, Windows will show a list of share target applications that have indicated they are capable of receiving one or more of the provided data types.
5. When users select one of the displayed apps, that app will be activated for sharing, resulting in its **`Application`** object's **`OnShareActivated`** method being called, with the shared information passed in as part of the method's arguments.
6. The share target app is given an opportunity to put UI content in the **Share** panel, so users can preview the content being shared, what the share target app intends to do with it, and control the process.
7. Users complete the operation in the **Share** pane UI provided by the share target app, resulting in Windows closing the **Share** pane.

Now that the general sequence of the share operation has been explained, the share source and share target app functionality will be discussed in detail, as well as a quick discussion of a couple of circumstances in which the behavior deviates slightly from this sequence.

Sharing Data from the Share Source App

As was mentioned in the share life-cycle discussion, a share source indicates itself as such by providing a handler to the **`DataRequested`** event on the **`DataTransferManager`** class. A reference to the **`DataTransferManager`** is obtained by calling the **`GetForCurrentView`** method on the **`DataTransferManager`** class.

Note: Some readers may notice that this pattern of calling `SomeManager.GetForCurrentView()` is repeated throughout the various contract and extension implementations in Windows Store apps.

The code for registering as a share source is as follows:

```
protected override void OnWindowCreated(WindowCreatedEventArgs args)
{
    // Obtain a reference to the DataTransferManager.
    var dataTransferManager =
       Windows.ApplicationModel.DataTransfer.DataTransferManager.GetForCurrentView();

    // Subscribe to the DataRequested event.
    dataTransferManager.DataRequested += OnShareDataRequested;

    // Subscribe to the TargetApplicationChosen event, which is
    // used to get the name of the app that requested the share,
    // mostly for analytics/statistical purposes.
    dataTransferManager.TargetApplicationChosen +=
       (sender, targetAppChosenEventArgs) =>
        {
```

```
            String selectedAppName = targetAppChosenEventArgs.ApplicationName;
            // Do something with the gathered app name.
        };

    base.OnWindowCreated(args);
}
```

Note: There is no need to register any kind of share source declaration in the app manifest file since a share source app is not activated or otherwise enumerated by Windows. The interaction is user-initiated when the app is already in the foreground.

When users invoke the **Share** charm while an app registered in this way is in the foreground, the app's **`DataRequested`** event handler will be called. Within that event handler, the app is responsible for filling the **`DataPackage`** object provided within the event's arguments. At a minimum, the **`Title`** property must be set and at least one data element must be provided, or the **Share** panel will indicate that an error has occurred. After the **`Title`** is provided, the app can provide one or more representations of the data to be shared.

Note: When an app is being displayed in the snapped state and the Share charm is invoked, the context for the share will be the foreground (large) app regardless of which of the two apps last had focus. If the Share panel was invoked programmatically using the `DataTransferManager.ShowShareUI()` static method, the app must first be unsnapped. To unsnap the app, see the `EnsureUnsapped` method shown in Chapter 3.

To provide data, the app calls the appropriate **`Set<DataType>`** method on the provided **`DataPackage`** instance. As previously mentioned, several different data types can be provided. It makes sense for an app to provide as many as possible in order to reach as many share target apps as possible, as well as to allow those targets to choose the data format that makes the most sense for their app. The following code shows different ways that data can be provided to the **`DataPackage`**:

```
var dataRequest = args.Request;
dataRequest.Data.Properties.Title = "Windows Store apps Succinctly Share";
dataRequest.Data.Properties.Description =
    "Content shared from the Windows Store apps Succinctly book.";

// This bitmap is used later.
var bitmapUri = new Uri("ms-appx:///Assets/Logo.png");
var bitmapReference = RandomAccessStreamReference.CreateFromUri(bitmapUri);

// Plain text.
dataRequest.Data.SetText("Plain Text to share");

// Richly formatted text.
dataRequest.Data.SetRtf(@"{\rtf1\ansi\pard This is some {\b bold} text.\par}");
```

```
//// A URI.
dataRequest.Data.SetUri(new Uri("http://www.syncfusion.com"));

// HTML
var rawHtml = "Windows Store apps Succinctly: <img src='assets/logo.png'>";
var formattedHtml = HtmlFormatHelper.CreateHtmlFormat(rawHtml);
dataRequest.Data.SetHtmlFormat(formattedHtml);
dataRequest.Data.ResourceMap.Add("assets\\logo.png", bitmapReference);

// Share a stream that contains a bitmap's contents.
dataRequest.Data.Properties.Thumbnail = bitmapReference;
dataRequest.Data.SetBitmap(bitmapReference);

// Allows defining a custom type of data to exchange beyond the system-provided values.
dataRequest.Data.SetData("Custom Format 1", new []{"Some text", "other text"});

// Allows defining an object that will only be fetched if it is explicitly requested.
dataRequest.Data.SetDataProvider("Custom Format 2", DelayedDataRequestCallback);

// Share StorageFile items:
// In case obtaining data to be shared is an async operation, use a deferral to prevent
// the DataRequested event from slipping through before the async operation completes.
var deferral = dataRequest.GetDeferral();

var file1 = await StorageFile.GetFileFromApplicationUriAsync(bitmapUri);
dataRequest.Data.SetStorageItems(new[] {file1}, true);

deferral.Complete();
```

From this sample, there are some important things to note:

- For HTML, the **`HtmlFormatHelper.CreateHtmlFormat`** call prepares the HTML fragment by ensuring all of the necessary HTML headers are included. Furthermore, the **`ResourceMap`** property is used to provide streams for the content that is referenced within the HTML fragment that would otherwise be inaccessible to the share target application.
- Note that when the bitmap is being set, a thumbnail is also being set. This will facilitate previewing the content in the share target application.
- Custom formats can be specified beyond the system-provided values, and can be exchanged as long as the share target app also knows about that type.
- A variation on the custom format option, a “data provider,” can be supplied which withholds the actual data payload until the share target requests the specific format ID, at which time the supplied callback method is run in the share source app to provide the requested data.
- Like other system-raised events where asynchronous operations may be called, it may be necessary to request a deferral in order to prevent control from being returned to Windows prior to the completion of the async operation.

If the app is not able to provide data (e.g., the app’s current page is showing a list of items where the selected item is what is shared, yet at present nothing is selected) it should provide a message indicating that sharing is not currently possible.

```
var dataRequest = args.Request;
var message =  “Nothing is currently selected. Please select an item to share”;
dataRequest.FailWithDisplayText(message);
```

Consuming Shared Data in the Share Target App

The other half of the Share contract is the share target app. Because this app is activated as the result of a share, there are a few more steps involved in setting it up than its share source counterpart. Once again, the simplest way for an application to be set up to implement the Share Target contract is to add the **`Share Target Contract`** item from the project's **Add Existing Item** command. This step addresses the three key steps required for share targets.

First, it adds the **`Share Target`** declaration to the application's manifest, and also sets up a couple of simple data formats: **`text`** and **`uri`**. These values can be removed, new formats can be added, or both, to indicate that the app can serve as a target for shares exposing one or more of these formats. In addition to formats, file types can be specified in the manifest in order to support shares exposing **`StorageFiles`** items that reference files of the given type (there is also a check box that can be selected to support any file type). If the app is able to receive one or more custom shared content types, those data formats must also be specified in the manifest.

The data format values specified in the manifest are used by Windows when the app is installed to record the sharing data types this app is able to receive. When a share is initiated, the value types present in the **`DataPackage`** provided by the share source app are used so that the list that Windows presents in the **Share** pane only includes apps that can handle one or more of the items in the **`DataPackage`**.

The second item provided by the Visual Studio template is the addition of an **`OnShareTargetActivated`** method override to the **`Application`** class. The **`OnShareTargetActivated`** method is called when the application is activated via its selection in the **Share** panel, similar to how the **`OnSearchActivated`** override behaves for the Search contract. The **`OnSearchActivated`** method will be called if the app is either being launched or resumed from a suspended state. Note that if the app is being resumed from a suspended state, the **`Resumed`** event is also going to be raised.

The last item added to the project in Visual Studio is a UI page that will be displayed within the **Share** pane when the app is activated as a share target. The provided **`OnSearchActivated`** method includes code to instantiate this page and then to call a provided **`Activate`** method, which accepts the arguments that were provided to the **`OnSearchActivated`** method. These arguments include a **`ShareOperation`** property which contains information specific to the Share contract. Within the **`Activate`** method implementation, various properties are set to display some of the shared data on the page, the current window is set to the page instance, and the **`Activate`** method is called to display the page UI.

It is up to the application to determine the page contents that are to be displayed in the **Share** pane. The general approach is to extract the shared data from the data package and display them in the UI in context of how the app will be consuming the shared data—for example, an email application UI that resembles an email editor window, including elements allowing users to select destination email addresses. As mentioned previously, the information related to the Share contract is provided in a **`ShareOperation`** object. In this object, the **`Data`** property provides access to a **`DataPackageView`**, which is a read-only version of the data package that was provided by the Share source application. This data includes methods that can be used to retrieve data from the package, as well as a **`Properties`** value that includes additional data about the share operation itself, including the **`Title`** and **`Description`** values that were provided.

Obtaining the data is usually a two-step operation. First, the **`DataPackageView`** is queried to see if it contains data of a particular format. If a format match is found, the data is then asynchronously retrieved using one of the provided methods, which basically all take the shape **`Get<DataType>Async()`**. An example of retrieving some of the data formats is shown in the following sample. In this example, the code returns as soon as a match is found, indicating preferred shared data types.

```
var dataPackageView = shareTargetActivatedEventArgs.ShareOperation.Data;

// Retrieve shared custom items.
if (dataPackageView.Contains("Custom Format 1"))
{
    var customItem = await dataPackageView.GetDataAsync("Custom Format 1");
    // TODO: Make use of the custom data format value.
    return;
}

// If a custom format entry is not found, retrieve shared RTF content.
if (dataPackageView.Contains(StandardDataFormats.Rtf))
{
    var rtfText = await dataPackageView.GetRtfAsync();
    // TODO: Make use of the shared text.
    return;
}

// If neither a custom format entry nor RTF content is found, retrieve plain text.
if (dataPackageView.Contains(StandardDataFormats.Text))
{
    var sharedText = await dataPackageView.GetTextAsync();
    // TODO: Make use of the shared text.
    return;
}
```

In most cases, the share target should try to retrieve and do something useful with data from each of the data types that have been indicated in the app manifest, even if the app stops as soon as it finds its first available data item.

Within the **Share** pane, in addition to previewing the shared data, users should also be provided with a means to "commit" the share operation, processing the data in whatever way is appropriate. When the data processing begins, the **ReportStarted** method on the **ShareOperation** object should be called to allow Windows to display UI elements that show the operation in progress, followed by either **ReportCompleted** or **ReportError** once the operation has finished.

```
try
{
    this._shareOperation.ReportStarted();
    // Process the shared data and any on-screen data entry the user has done.
    this._shareOperation.ReportCompleted();
}
catch (Exception ex)
{
    var message = "An error has occurred during sharing: " + ex.Message;
    this._shareOperation.ReportError(message);
}
```

Callback Data

There is a special case for sharing where the process is slightly different. When the share source application includes a custom type value in the **Data Package** by using the **SetDataProvider** call, it is indicating that it is going to wait to provide the data until the share target application actually requests it by calling **GetDataAsync** with a format type that matches the custom type provided. There is nothing special that needs to be done in the share target application, but the share source must provide and implement a callback method that will be invoked when the share target call to **GetDataAsync** is issued. This provides an opportunity for the share source application to avoid performing unnecessary work which may be resource intensive, or for which there is some other reason to be avoided until the data is actually needed and requested. An example callback method is shown in the following sample. Note the use of a deferral in case an async operation is used to prevent returning to the share target before the async operation has completed.

```
private async void DelayedDataRequestCallback(DataProviderRequest request)
{
    var deferral = request.GetDeferral();
    var data = await PerformSomeAsyncOperationToGetData();
    request.SetData(data);
    deferral.Complete();
}
```

QuickLinks

When a share target app handles a request for sharing data, it can optionally provide a QuickLink which contains information that can be used in a subsequent share request to quickly invoke the share with a packaged set of parameters, presumably those that were used in the previous invocation. For example, an email application would use this to add a QuickLink to the last recipient of an email via a share, with the presumption that users are likely to share to that email address again in the future.

To set up a QuickLink, when users commit a share in the share target app, an instance of the **QuickLink** class should be created with a descriptive title, potentially a thumbnail icon, and an ID value that can indicate what should be done when the **QuickLink** is invoked. It is also necessary to set the supported data formats and supported file types that the QuickLink will recognize—these are set independently of the values defined in the manifest for the share target app. Finally, the **QuickLink** object is provided to the **ReportCompleted** method call that is used to indicate that the share has been completed.

```
// Configure the QuickLink.
var bitmapUri = new Uri("ms-appx:///Assets/Logo.png");
var thumbnail = RandomAccessStreamReference.CreateFromUri(bitmapUri);
var quickLink = new QuickLink()
{
    Title = "Share to Succinctly App",
    Thumbnail = thumbnail,
    Id = "UI/user provided data to be reused"
};
// Indicate the data formats and file types this QuickLink will support.
quickLink.SupportedDataFormats.Add(StandardDataFormats.Text);
quickLink.SupportedDataFormats.Add("Custom Format 1");
//quickLink.SupportedFileTypes.Add(...);

// Indicate that the Share contract has been completed.
this._shareOperation.ReportCompleted(quickLink);
```

The resulting QuickLink in the Share UI is shown in the following figure:

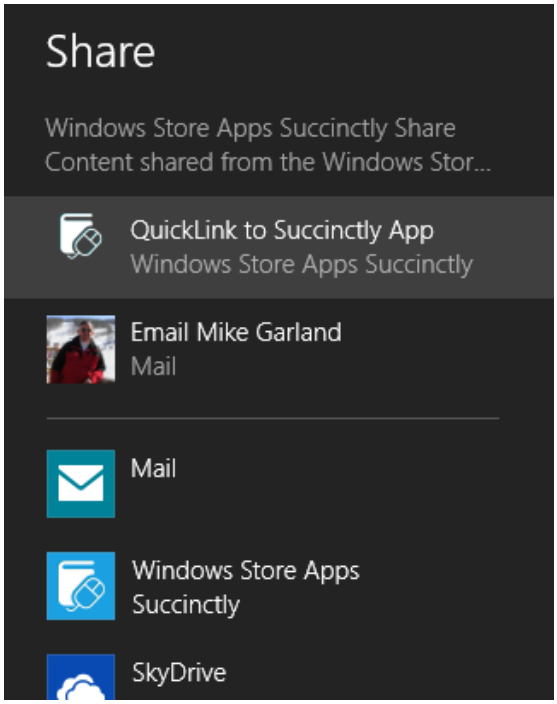


Figure 17: A Share QuickLink

When the QuickLink is invoked by the user during a share, the **ShareOperation** object contained in the arguments passed to the **OnShareTargetActivated** method will include a **QuickLinkId** property that contains the **Id** value supplied when the **QuickLink** object was created. It is up to the share target app to decide how to process that **Id** value to create the consistent experience intended for users. In some cases, the contents in the **QuickLinkId** string will be sufficient. In others, it may be that the **Id** value is all or part of a key into the **ApplicationData** settings or provides some other mechanism to obtain the state value from storage. Code for using the **QuickLinkId** to obtain values from **ApplicationSettings** follows:

```
// Check to see if Share was invoked with a QuickLink.
if (!String.IsNullOrWhiteSpace(args.ShareOperation.QuickLinkId))
{
    // Retrieve meaningful settings values.
    var settingsId = "QuickLink" + args.ShareOperation.QuickLinkId;
    var settings = ApplicationData.Current.LocalSettings.Values[settingsId];

    // Set up the UI with the share data and the value(s) obtained from settings.
    // ...
}
else
{
    // Just set up the UI with the share data.
    // ...
}
```

Sending App Content to Devices

The Devices charm provides the ability to send app content to devices, rather than to other apps, which is handled by the Share charm. Within the Devices charm, three key device sharing scenarios are available: Print To, Play To, and Send To. Print To supports sending content to a printer, Play To supports sending media content to a Microsoft-certified DLNA device, and Send To supports sending content to a device that is enabled for Near Field Communication (NFC). Since printing from an app is more likely to be the most relevant scenario, this section will limit its scope to a discussion of printing.

Note: For further information on Play To, please refer to the documentation on MSDN at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/Hh465183(v=win.10).aspx.

For further information on Send To, please refer to the documentation on MSDN at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/Hh465221(v=win.10).aspx.

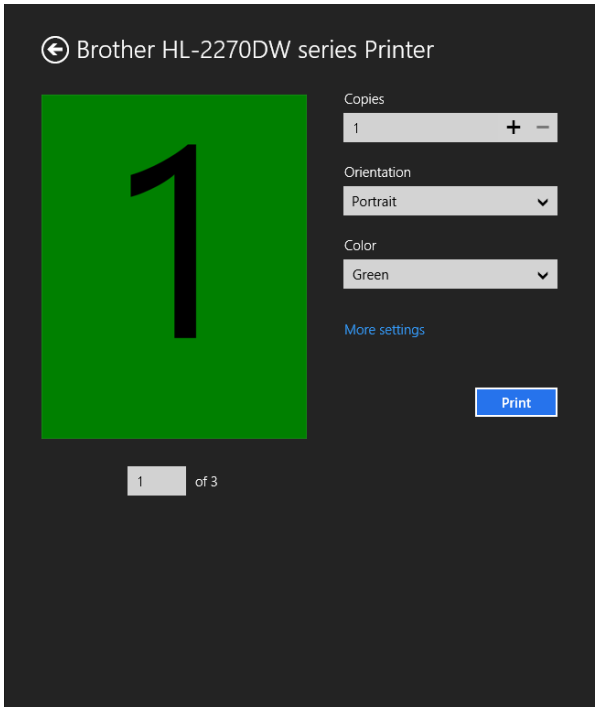


Figure 18: The Windows Print Panel

Implementing the Print Contract

At a high level, there are three essential steps required for an app to participate in the Print contract:

1. A **`PrintDocument`** object needs to be defined and set up to handle the events that will be raised on it as a response to user interaction with the Windows Print UI.
2. Second, the application needs to subscribe to the **`OnPrintTaskRequested`** event of the **`PrintManager`**. In this event handler, a new **`PrintTask`** needs to be created and a callback needs to be set up to provide the **`PrintTask`** with the previously created **`PrintDocument`**.
3. The layout of the content to be presented for previewing and printing needs to be provided. The printing system expects XAML-defined **`UIElement`** values to represent the layout of this content, which offers tremendous opportunities for reuse of existing layout controls, although some adjustments will need to be made to translate between layouts that are appropriate for an on-screen display and those that are meant to fit on a printed page.

Beyond these essential steps, options are available for receiving events related to the progress of the **`PrintTask`**, as well as for customizing the print options shown to users such as those specific to the current app's needs.

Note: There can only be one subscription to the OnPrintTaskRequested event in an app at any given time. Attempting to add another subscription will throw an exception. This is important to note because having a handler for this event notifies Windows that the application is currently in a context where it can print and that it is appropriate to show

printers when the Devices pane is shown. Because of this, there are times in a given app when the event should be handled and times when it should not be handled. This is typically done by subscribing to the event in a page's `OnNavigatedTo` method and unsubscribing in the `OnNavigatedFrom` method, rather than the single app-wide subscription within the `OnWindowCreated` override that is used in many of the other contracts. Failing to unsubscribe can result in a situation where a subsequent page attempts to create an error-causing second subscription.

The following code shows the typical process of setting up a **`PrintDocument`**, subscribing to its events, subscribing to the **`OnPrintTaskRequested`** event, and setting up the callback to provide the **`PrintTask`** with the **`PrintDocument`** contents.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    // Create the PrintDocument and subscribe to its events.
    _printDocument = new PrintDocument();
    _printDocument.Paginate += OnRequestPrintPreviewPages;
    _printDocument.GetPreviewPage += OnGetPreviewPage;
    _printDocument.AddPages += OnAddPrintPages;

    // If this handler is declared, printers are shown when the Device Charm is
    // invoked. Put another way, when there is a handler, there is an expectation that
    // printing is currently possible.
    // Likewise, if this handler is subscribed to twice, an exception is thrown
    var printManager = Windows.Graphics.Printing.PrintManager.GetForCurrentView();
    printManager.PrintTaskRequested += OnPrintTaskRequested;
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    // Unsubscribe when leaving the page.
    var printManager = Windows.Graphics.Printing.PrintManager.GetForCurrentView();
    printManager.PrintTaskRequested -= OnPrintTaskRequested;
}

private void OnPrintTaskRequested(PrintManager sender,
                                                PrintTaskRequestedEventArgs args)
{
    var printTask = args.Request.CreatePrintTask("Print task title",
    async taskSourceRequestedArgs =>
    {
        // Note that this needs to be marshalled back to the UI thread.
        await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
            () =>
                {
                    // Called when a printer device is selected.
                    taskSourceRequestedArgs.SetSource(_printDocument.DocumentSource);
                });
    });

    // Called before pages are requested for preview.
    printTask.Previewing += (s, e) => Debug.WriteLine("Submitting");

    // Called when the user hits "print".
    printTask.Submitting += (s, e) => Debug.WriteLine("Submitting");
```

```
    // Called once for each page that is printed.
    printTask.Progressing += (s, e) => Debug.WriteLine("Progressing - " +
                                                       e.DocumentPageCount);

    // Called after all pages are printed.
    printTask.Completed += (s, e) =>
    {
        // Completion values include Abandoned, Canceled, Failed, Submitted.
        if (e.Completion == PrintTaskCompletion.Failed)
        {
            //Notify user that an error occurred - needs to be marshalled to UI thread.
        }
    };
}
```

The following is being done in this code:

- In the page's **OnNavigatedTo** method, a new **PrintDocument** is being created and handlers are being provided for its events. The purpose of each of these handlers will be discussed later.
- Also in the page's **OnNavigatedTo** method, a handler is provided for the **PrintTaskRequested** event.
- On the page's **OnNavigatedFrom** event, the **OnPrintTaskRequested** method is being unsubscribed from the **PrintTaskRequested** event.
- In the **OnPrintTaskRequested** method, a new **PrintTask** titled "Print task title" is created through the provided event arguments. This is also being set up with a callback to set the task's source from the previously created **PrintDocument** source property. Note that this has to execute on the UI thread, even though the callback is likely not to be, so the call needs to be marshalled to the UI thread using the page's **Dispatcher** object's **RunAsync** method.
- Additionally, some optional event handlers are being provided for the created **PrintTask**. These handlers are executed throughout the various steps in the lifetime of the **PrintTask** and can be used to provide user feedback regarding the status of the printing operation.

The three essential event handlers for the **PrintDocument** are used to provide the content displayed in the **Print Preview** window, as well as the content being sent to the printer. These handlers need to be set up for the **Paginate**, **GetPreviewPage**, and **AddPages** events.

The **Paginate** event is fired when users have selected a printer and the Windows **Print** panel is going to display the print preview content for the print job. The handler for this event is responsible for using the provided **PrintTaskOptions**, which contain the current print settings concerning page size, orientation, etc., to determine the number of pages available to be previewed and calling the **SetPreviewPageCount** on the current **PrintDocument** with the result of this page count. In most cases, the act of calculating the pagination involves actually laying out the page contents, in which case it makes sense to store the elements for later use.

```
// This handler for the Paginate event is raised to request the count of preview pages.
private void OnRequestPrintPreviewPages(Object sender, PaginateEventArgs e)
{
```

```
    _printPages.Clear();

    #region Prepare Preview Page Content
    #endregion

    _printDocument.SetPreviewPageCount(_printPages.Count, PreviewPageCountType.Final);
    // Can also be Intermediate to indicate the count is based on non-final pages.
}
```

Note: In most cases, the act of calculating the pagination involves actually laying out the page contents, in which case it makes sense to store the elements for later use. Furthermore, the `GetPreviewPage` event does not provide information about the page settings, so the page size being targeted is not available at that point to help calculate the pagination.

The **`GetPreviewPage`** event is fired to request the actual UI elements that are to be displayed within the **Print Preview** panel for the current page. As mentioned previously, usually this simply provides elements that were calculated in the previous **`Paginate`** event. This event will be called once for every preview page that is being displayed.

```
// This requests/receives a specific print preview page.
private void OnGetPreviewPage(Object sender, GetPreviewPageEventArgs e)
{
    // e.PageNumber is the 1-based page number for the page to be displayed
    // in the preview panel.
    _printDocument.SetPreviewPage(e.PageNumber, _printPages[e.PageNumber - 1]);
}
```

The final event handler to be provided is for the **`AddPages`** event. This event is raised when the system is ready to start printing. In this method, the final page rendering should occur based on the print options provided in the event arguments, with a call to the **`PrintDocument AddPage`** method for each page. The call should complete with a call to the **`AddPagesComplete`** method of the **`PrintDocument`** to signal that all of the pages have been provided.

```
// This is called when the system is ready to print and provides the final pages.
private void OnAddPrintPages(Object sender, AddPagesEventArgs e)
{
    var finalPages = new List<UIElement>();

    #region Prepare Final Page Content
    #endregion

    foreach (var finalPage in finalPages)
    {
        _printDocument.AddPage(finalPage);
    }
    _printDocument.AddPagesComplete();
}
```

Customizing Print Settings

There are two ways that the print settings in the Windows **Print** panel can be customized. First, the quantity and order of the default values can be selected. Second, a custom setting value can be provided. In either case, before working with the options it is necessary to obtain an "advanced" version of the options by using the **`PrintTaskOptionDetails GetFromPrintTaskOptions`** method.

When the **`PrintTask`** is created, the print options that are displayed can be altered by working with the **`DisplayedOptions`** collection. The various standard options can then be simply added to the collection using the **`StandardPrintTaskOptions`** properties.

```
// Get an OptionDetails item from the options on the current print task.
var advancedPrintOptions =
PrintTaskOptionDetails.GetFromPrintTaskOptions(printTask.Options);

// Choose which of the "standard" printer options should be shown. The order in
// which the options are appended determines the order in which they appear in the UI.
var displayedOptions = advancedPrintOptions.DisplayedOptions;
displayedOptions.Clear();

displayedOptions.Add(StandardPrintTaskOptions.Copies);
displayedOptions.Add(StandardPrintTaskOptions.Orientation);
```

For custom properties, first it is necessary to define the custom option. The new item can then be added to the collection of displayed options. The last step is to provide a handler for when the value is changed and to instruct the **`PrintDocument`** to forcibly be refreshed so that the new options values can be used in recalculating the pagination, resulting in the **`Paginated`** and **`GetPreviewPane`** events being reraised.

```
// Create and populate a new custom option element - shown as a list option UI element.
var colorIdOption = advancedPrintOptions.CreateItemListOption("ColorId", "Color");
colorIdOption.AddItem("RedId", "Red");
colorIdOption.AddItem("GreenId", "Green");
colorIdOption.AddItem("BlueId", "Blue");
colorIdOption.TrySetValue("RedId");

// Add the custom option item to the UI.
displayedOptions.Add("ColorId");

// Set up a handler for when the custom option item's value is changed.
advancedPrintOptions.OptionChanged += async (o, e) =>
{
    if (e.OptionId == null) return;
    var changedOptionId = e.OptionId.ToString();
    if (changedOptionId == "ColorId")
    {
        await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            // Force the preview to be recalculated.
            _printDocument.InvalidatePreview();
        });
    }
};
```

It is up to the code in the **Paginate** and **AddPages** event handlers to check for the custom setting and incorporate the setting's value into the page layout calculations.

```
// Obtain the custom property value to use in pagination calculations.
var printTaskOptionDetails =
PrintTaskOptionDetails.GetFromPrintTaskOptions(e.PrintTaskOptions);
var colorIdOptions = printTaskOptionDetails.Options["ColorId"];
var colorId = colorIdOptions.Value.ToString();
var selectedColor =
    colorId == "RedId" ? Colors.Red :
    colorId == "GreenId" ? Colors.Green :
    colorId == "BlueId" ? Colors.Blue :
    Colors.Black;
```

Managing App Settings

The last charm to discuss is the Settings charm. Probably no other feature in Windows applications is more scattered throughout different menu and submenu combinations, toolbar icons, and other user interface controls than application settings. The Settings charm is intended to let Windows provide one common and predictable user interface element for users to manage their settings across all installed applications.

Users can invoke the **Settings** pane by either selecting the **Settings** charm or by using the Windows logo key+I keyboard combination. When invoked, the **Settings** pane will be opened in the context of the foreground app. In the event an app is being displayed in snapped view, the **Settings** pane will open for the filled app.

Note: The Settings pane can also be invoked programmatically by calling the `SettingsPane.Show()` static method—note that the app has to be unsnapped or this will throw an exception. To unsnap the app, see the `EnsureUnsapped` method shown in the previous chapter.

For an app to participate in the Settings contract, it needs to provide a list of **SettingCommand** objects and corresponding UI content to be displayed for each object. It is up to the app to display the UI content from within the handler callback that is specified in the **SettingsCommand** constructor. Since Settings does not launch the application, it does not require any special entries in the declarations section of the app manifest file. The following code shows an app registering for the **CommandsRequested** event, which is called when the Settings charm is invoked for the current app, to display **About** and **Options** entries in the **Settings** pane, plus empty lambda expressions for the handlers:

```
protected override void OnWindowCreated(WindowCreatedEventArgs args)
{
    var settingsPane = SettingsPane.GetForCurrentView();
    settingsPane.CommandsRequested += (o, e) =>
    {
        // Create "about" and "options" links in the Settings pane.
        var aboutCommand
            = new SettingsCommand("aboutCmdId", "About", handler => {/*...*/});
        var optionsCommand
            = new SettingsCommand("optionsCmdId", "Options", handler => {/*...*/});

        var settingsPaneCommandsRequest = e.Request;
        settingsPaneCommandsRequest.ApplicationCommands.Add(aboutCommand);
        settingsPaneCommandsRequest.ApplicationCommands.Add(optionsCommand);
    };

    base.OnWindowCreated(args);
}
```

When one of the links displayed by Windows for each added **SettingsCommand** is selected, the corresponding callback handler is invoked. Within this handler, the app should display the appropriate settings user interface elements.

Tip: When working with HTML and JavaScript, a SettingsFlyout control is provided for the purpose of displaying settings controls within the Settings pane region. The XAML/.NET frameworks do not currently include a corresponding control, so either one must be created or a pre-existing control needs to be obtained from somewhere. The Callisto controls available on GitHub provide a settings flyout that works quite well in this capacity. The Callisto controls are available at https://github.com/timheuer/callisto, and the `SettingsFlyout` control will be used in the upcoming examples in this section.

To display a panel, first a **UserControl** must be created that contains the desired UI elements for users to make the necessary settings changes. An instance of that control is created and placed within the **SettingsFlyout** as its **Content**. The **FlyoutWidth** property for the Callisto **SettingsFlyout** control can be either **SettingsFlyoutWidth.Narrow** or **SettingsFlyoutWidth.Wide**, corresponding to the 346-pixel wide or 646-pixel wide values indicated in the published guidelines for Settings pane contents. The **SettingsFlyout** is then displayed when its **IsOpen** property is set to **True**.

```
var optionsCommand = new SettingsCommand("optionsCmdId", "Options", handler =>
{
    // Display an "About" flyout.
    var flyout = new SettingsFlyout
    {
        Content = new OptionsControl(),
        FlyoutWidth = SettingsFlyout.SettingsFlyoutWidth.Narrow, // Wide
        IsOpen = true,
    };
});
```

Note: It is important to note that settings should be implemented as light-dismiss controls, meaning that the control will be dismissed when users touch some other part of the app. The app should apply settings changes as soon as users stop interacting with that specific setting control, and should not include a Save or a Commit button.

Other Extensibility Options

In addition to the contracts that are exposed through the charms, there are several other contracts and extensions that can be used to integrate and extend a Windows Store app. These additional options provide functionality for participating when a device is inserted into a computer, providing camera options and effects, integrating with the contact picker UI, handling files of a given extension, integrating with the file picker UIs, integrating with family safety options, and handling activation for a given protocol, among others. This section will focus on the more common file picker UI integrations, as well as the file type and URI schema handling. Additional information about the other contracts and extensions can be found in the MSDN documentation at http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx.

File Picker Contracts

As was discussed in the previous chapter, the File Save Picker and File Open Picker can be extended to integrate app data instead of just presenting files within the file system. When an app implements the File Picker contracts, it will be presented as an option within the file picker pull-down, and when selected, will present an application page within the file picker's content region.

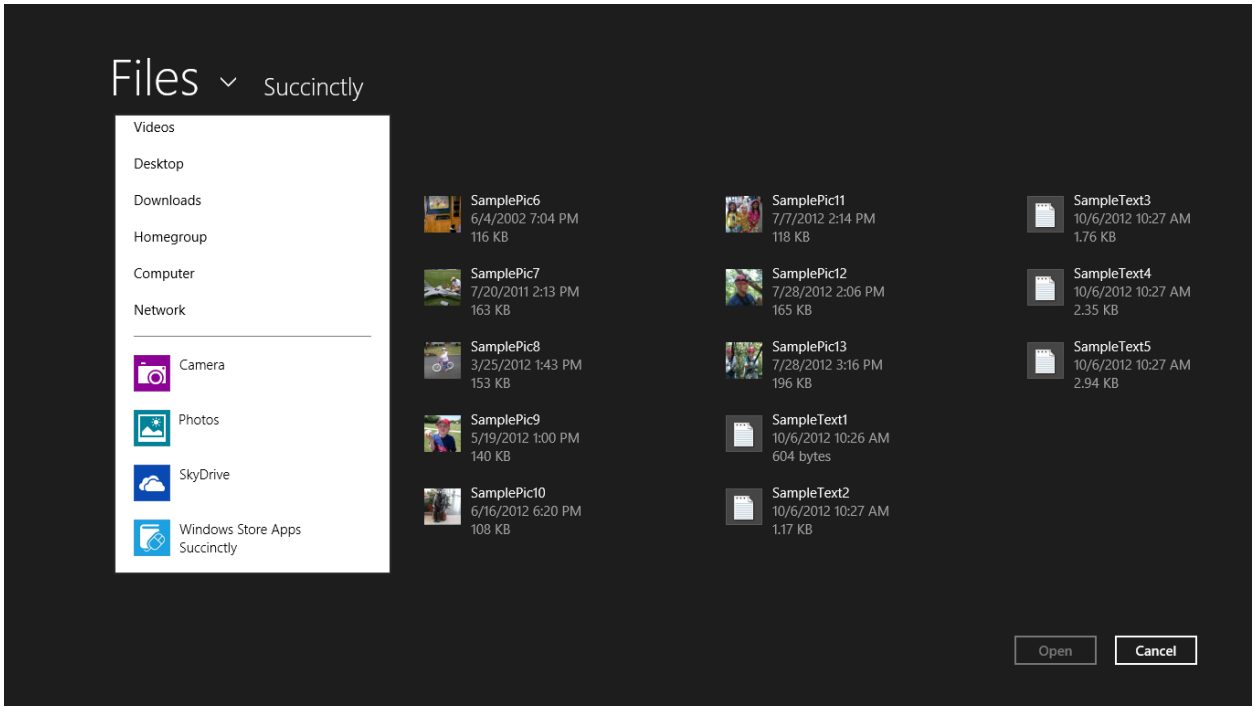


Figure 19: Apps Integrated with the File Open Picker

There are several reasons why it may be advantageous to have an app provide custom integration into the File Open Picker and File Save Picker. Some examples include:

- Allowing the app the ability to load or save content from the app's private **`ApplicationData`** storage contents. Through file picker integration, the app can provide other apps a managed view into these otherwise inaccessible storage areas.
- Providing access to items that are not actually files on disk, such as network content (like SkyDrive), device access (such as images obtained from an attached camera), or perhaps data stored within a private SQLite database. External apps will see these as though they were files on the file system.

Integrating with the File Save Picker

To integrate with the File Save Picker, an app needs to do the following:

- In the app manifest file, add the **`FileSavePicker`** declaration. Within this declaration, one or more file types need to be specified, or the **Supports any file type** check box must be selected. The file type specification is used to determine if the app should be shown in the File Save Picker pull-down based on the file types that have been specified by the calling app in the displayed File Save Picker instance. If an app cannot handle any of the file types the picker is set up to save, there is no value in presenting it as an option to users.
- In the **`Application`** class, provide an override for the **`OnFileSavePickerActivated`** method.
- Within the **`OnFileSavePickerActivated`** override, create a UI control (typically a **`Page`**) to be displayed in the File Save Picker content area, set it to the current window contents, and call **`Activate`**. This page is primarily used to allow users to navigate

whatever hierarchy the app's pseudo file system is intended to provide so that they can ultimately select a specific file location to save to.

- Subscribe to the **`TargetFileRequested`** event exposed by the **`FileSavePickerUI`** property on the activation arguments passed to **`OnFileSavePickerActivated`**. This event will be raised when users tap the **Save** or **Commit** button in the File Save Picker. In this event handler, the selected target file should be provided to the event's arguments through the **`TargetFile`** property. The application that invoked the File Save Picker UI will receive this value as the result from the call to display the File Save Picker, and can then process the file save operation.

The following code shows the **`Activate`** call and the related **`TargetFileRequested`** event handler that are provided by the user interface instantiated in the **`OnFileSavePickerActivated`** method:

```
public void Activate(FileSavePickerActivatedEventArgs args)
{
    _fileSavePickerUI = args.FileSavePickerUI;
    _fileSavePickerUI.TargetFileRequested += HandleTargetFileRequested;
    Window.Current.Content = this;
    Window.Current.Activate();
}

private async void HandleTargetFileRequested(FileSavePickerUI sender,
TargetFileRequestedEventArgs args)
{
    // Request a deferral to accommodate the async file creation.
    var deferral = args.Request.GetDeferral();

    // Create the file in ApplicationData to be returned as the user's selection.
    var localFolder = ApplicationData.Current.LocalFolder;
    args.Request.TargetFile = await localFolder.CreateFileAsync(sender.FileName,
                                        CreationCollisionOption.GenerateUniqueName);

    // Complete the deferral.
    deferral.Complete();
}
```

The app that calls the File Save Picker can then use this file as a target for its data:

```
private async void HandleSaveFileClicked(Object sender, RoutedEventArgs e)
{
    var picker = new FileSavePicker();
    picker.SuggestedFileName = "Sample File Name";
    picker.FileTypeChoices.Add("Image Files",
                                    new List<String>(new []{".png", ".jpg", ".bmp"}));
    var chosenFile = await picker.PickSaveFileAsync();
    if (chosenFile != null)
    {
        await _fileToSave.CopyAndReplaceAsync(chosenFile);
    }
}
```

Integrating with the File Open Picker

The process to integrate with the File Open Picker is very similar to integrating with the File Save Picker, though it is simplified somewhat because Visual Studio provides a File Open Picker Contract item that can be added to a project, which automates much of the process. Nonetheless, the steps are as follows:

1. In the app manifest file, add the **`FileOpenPicker`** declaration. Within this declaration, one or more file types need to be specified, or the **Supports any file type** check box must be selected. The file type specification is used to determine if the app should be shown in the File Open Picker pull-down, based on the file types that have been specified by calling the app in the displayed File Open Picker instance. If an app cannot handle any of the file types the picker is set up to open, there is no value in presenting it as an option to users.
2. In the **`Application`** class, provide an override for the **`OnFileOpenPickerActivated`** method.
3. Within the **`OnFileSavePickerActivated`** override, create a UI control (typically a **`Page`**) to be displayed in the File Open Picker content area, set it to the current window contents, and call **`Activate`**. This page is used to show users the files they can choose to open, as well as to provide navigation through the pseudo file system that the app is providing.
4. As users select content in the user interface provided, the app should update the **`FileOpenPickerUI`** that was provided in the activation arguments by calling the **`AddFile`** and **`RemoveFile`** methods. These values will be what are returned to the calling app when users tap the **Open** button in the picker. Note that File Open Pickers can be set to allow single or multiple selection. For single selection, only one item can be added at a time.
5. When using multiple selections in the File Open Picker, it is necessary to subscribe to the **`FileRemoved`** event on the **`FileOpenPickerUI`** element. The File Open Picker UI exposes a list of the chosen items, and users can remove the selected items from this list. This event is used to give the custom UI the opportunity to stay in sync with the list of selected items.

The following code shows the **`Activate`** call, the calls to **`AddFile`** and **`RemoveFile`** resulting from users making selection changes in the UI, and the **`FileRemoved`** event handler to keep the UI in sync with the File Open Picker's contents:

```
public void Activate(FileOpenPickerActivatedEventArgs args)
{
    this._fileOpenPickerUI = args.FileOpenPickerUI;
    _fileOpenPickerUI.FileRemoved += FilePickerUI_FileRemoved;

    Window.Current.Content = this;
    Window.Current.Activate();

    // Custom code to populate the UI with the files in the currently displayed
context.
    DisplayCurrentFolderFiles();
}

private void FileGridView_SelectionChanged(Object sender, SelectionChangedEventArgs e)
{
    // Add any newly selected files in the UI to the Picker.
    foreach (var addedFile in e.AddedItems.Cast<SampleFile>())
    {
        if (_fileOpenPickerUI.CanAddFile(addedFile.BackingFile))
        {
            _fileOpenPickerUI.AddFile(addedFile.Title, addedFile.BackingFile);
        }
    }

    // Remove deselected items from the Picker.
    foreach (var removedFile in e.RemovedItems.Cast<SampleFile>())
    {
        _fileOpenPickerUI.RemoveFile(removedFile.Title);
    }
}

private async void FilePickerUI_FileRemoved(FileOpenPickerUI sender,
                                                    FileRemovedEventArgs args)
{
    // Ensure the call occurs on the UI thread.
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
    {
        // Find the GridView item matching the item removed from the Picker UI list.
        var selectedFiles = fileGridView.SelectedItems.Cast<SampleFile>();
        var removedSelectedFile =
                            selectedFiles.FirstOrDefault(x => x.Title == args.Id);
        if (removedSelectedFile != null)
        {
            fileGridView.SelectedItems.Remove(removedSelectedFile);
        }
    });
}
```

Note: The `FileRemoved` event is actually not raised on the UI thread, so any interactions with the UI need to be marshalled over to the UI thread or a runtime exception will occur. This can be done by using the `Dispatcher.RunAsync` method.

The app that calls the File Open Picker can then use the selected files as it sees fit:

```
private async void HandleOpenFileClicked(Object sender, RoutedEventArgs e)
{
    var picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".txt");
    picker.FileTypeFilter.Add(".jpg");
    picker.FileTypeFilter.Add(".png");

    // Single file selection.
    var firstFile = await picker.PickSingleFileAsync();

    // Multiple file selection.
    var selectedFiles = await picker.PickMultipleFilesAsync();
}
```

While the examples listed so far have focused on implementing the contract to allow managed access to application data storage, the items being exchanged do not strictly have to be files within a file system. An alternative to consider involves using the **`StorageFile.CreateStreamedFileAsync`** method to create a **`StorageFile`** object around a stream of data, which could be data obtained from a device or a network call. For more information on this method, please refer to the documentation at http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.storagefile.createstreamedfileasync.aspx.

Handling File Types and Protocols

The final extensibility tools that will be discussed are the File Type Activation and Protocol Activation extensions. These extensions allow a Windows Store app to register as one that can be activated when some other part of the system attempts to launch a file with a certain extension, or a URI with a certain protocol, respectively.

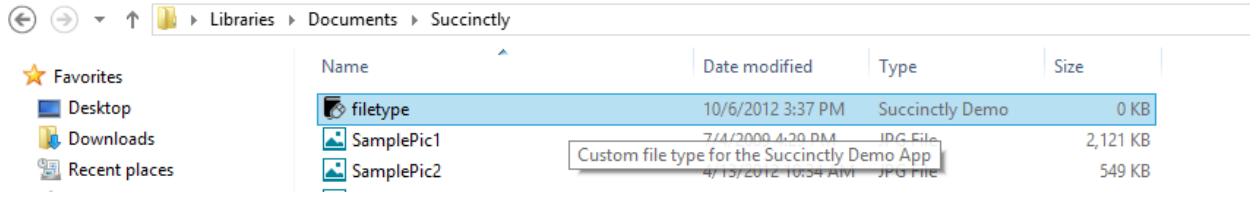


Figure 20: Display in Windows Explorer of an Associated File

The steps required to participate in these extensions are very similar for both cases:

- In the app manifest file, add either a file type associations or protocol declaration. Note that multiple declarations of these protocols are allowed within any one app to allow multiple association groups to be defined.
- For file type associations, a name must be provided which simply serves as a unique identifier for each entry (it must be all lowercase letters), and one or more supported file type values must be included, which include a file type extension with the period, and a content type (the MIME type). There are several other optional customization options

available, including the display name that is shown next to files of this type in Windows, a logo to be shown for files of this type, and the tooltip value to be shown.

- For protocol associations, a name must be provided which serves both as a unique identifier for the entry and the name of the protocol (it must be all lowercase letters). Optionally, a display name and a logo can also be provided.
- The corresponding activation method override needs to be provided in the application class:
    - For file types, the **`OnFileActivated`** override needs to be provided. The activation arguments that are provided include a list of files that lead to the activation in the **`Files`** property and a **`Verb`** property that identifies the activation verb related to this activation.
    - For protocols, the general-purpose **`OnActivated`** override needs to be provided, in which case the activation arguments' **`Kind`** parameter will be set to **`ActivationKind.Protocol`**, and the provided **`IActivatedEventArgs`** can be cast to a **`ProtocolActivatedEventArgs`** object. The activation URI is contained in the **`Uri`** property.

Examples of the activation method overrides are provided in the following sample:

```
protected override void OnFileActivated(FileActivatedEventArgs args)
{
    var listOfSelectedFiles = args.Files;
    var verb = args.Verb;

    // Display a UI that is appropriate to the item of the given file type.
    Window.Current.Content = new MainPage();
    Window.Current.Activate();
}

protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.Protocol)
    {
        var protocolArgs = args as ProtocolActivatedEventArgs;
        var activationUri = protocolArgs.Uri;

        // Display a UI that is appropriate to the item of the given protocol.
        Window.Current.Content = new MainPage();
        Window.Current.Activate();
    }
    else
    {
        // Handle an alternate activation.
    }
}
```

Recap

This chapter introduced several ways that Windows Store apps can participate in common Windows user experience scenarios, allowing information to be exchanged between the app and Windows, and in some cases, between apps. This included discussing the various contracts exposed to support interaction with the Windows charms bar, as well as other options that included integrating an app with the File Open Picker and File Save Picker, and registering a file to be used when a particular file type or URI protocol type is invoked. In several cases, these integration points involved adding ways in which an application could be activated, allowing an app to have even more opportunities to be consumed by end users.

The next chapter will look at additional options for a Windows Store app to provide information to users even when they aren't actually running, through support for tiles, toasts, and notifications.

Chapter 5 Tiles, Toasts, and Notifications

The previous chapters have discussed several different ways that a Windows Store app can be activated. These activations have occurred as responses to user interaction with Windows and as a result of the app participating in different Windows contracts and extensions—remember that launching an application from its Start tile is actually an implementation of the Launch contract. However, Windows Store apps have ways to provide information and seem alive even when the app itself hasn't been activated: tiles and toasts.

For Windows Store apps, the tiles that appear on the Start screen are more than just oversized desktop icons. These tiles are "live tiles"—their display can be updated with text, images, icons, and animations in response to notifications from both within the system and from external sources. Apps can have both primary and secondary tiles, the latter of which can be used to provide additional information that is passed to the application if they are used to launch the app.

Toasts provide apps a way to communicate information to users while they are using another app. Toasts appear as pop-up notifications in the upper corner of the app, and much like live tiles can include text and images. Toast messages also include information that can be passed to the application if users choose to use the toast to launch the application.

This chapter will show how live tiles can be used to add functionality to otherwise inactive Windows Store apps. Then, toasts will be explored, including configuring how they appear, controlling when they appear, and reacting to what occurs when they are used to launch an app. The chapter will conclude with a discussion of push notifications, which allow updates to be triggered from services external to both the app and the device itself.

Live Tiles

Every Windows Store app that is installed initially has a tile on the Windows Start screen that can be used to launch the application. By default, each app includes a square tile and optionally can include a larger "wide" tile, with the app's users deciding which size tile they want to include on their Start screen. The initial default content of an app's tile is set in the app manifest file, and includes the following options:

- The tile's standard-size logo image.
- The tile's wide-size logo image.
- Whether title text should be displayed when the standard, wide, both, or neither tile is shown.
- Whether the foreground for text and other content is light or dark if it is being shown.
- The background color of the tile.

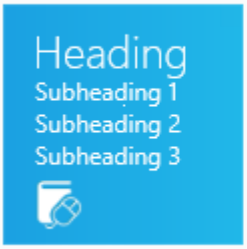
Note: If an app includes a wide tile, when the app is installed the wide tile will be the one that is initially displayed. Users can then choose to resize it to be displayed as the smaller tile, or even opt to remove the tile from the Start screen altogether. When an app is removed from the Start screen, it is still available to be launched by users from the results of a search in the Start screen unless they uninstall the app.

Beyond the content initially set for a tile, an app can affect the tile's appearance in several ways. This includes directly or periodically updating text and image content that is displayed within the tile, updating numeric or glyph "badge" content on the tile, or even creating secondary tiles that provide users the opportunity to directly access specific app functionality from their Windows Start screen.

Updating the Live Tile Content

Through updates to live tile content, an app can be made to seem alive and engaging, even when the app itself isn't actually running. Tiles can be updated immediately, on a predetermined repeating schedule, or at a set time. They can even be set up to cycle through a rotating queue of up to five tile designs. Both the standard and wide tile contents can be updated in these ways.

A tile's contents are defined by using XML that specifies a combination of text and images that are to be displayed. This XML is based on a set of predefined templates provided by Windows. There are 46 predefined template types—10 square and 36 wide—which can be chosen for tile layouts. Some of these layouts are considered "peek" layouts; their content includes an image that fills the tile and is scrolled in and out with alternate content that includes some text and perhaps an additional image. There are four square and 14 wide layouts that support peek.




```
<tile>
  <visual>
    <binding template="TileSquareText01">
      <text id="1">Heading</text>
      <text id="2">Subheading 1</text>
      <text id="3">Subheading 2</text>
      <text id="4">Subheading 3</text>
    </binding>
  </visual>
</tile>
```

Figure 21: An Updated Tile and the Corresponding XML Template

In addition to text and image information, a tile can also include a "branding" setting. A tile's branding indicates whether the content in the lower corner of the tile (lower left corner in left-to-right languages) will display the app's small logo, the app's title, or no content whatsoever. The small logo is the default.

Immediate Tile Updates

The first step in specifying a tile update is to define the XML to be used to define the tile's contents. The XML for each template can be found in the tile template catalog, which is available at http://msdn.microsoft.com/en-us/library/windows/apps/Hh761491.aspx. This XML can be compiled by hand, or the **GetTemplateContent** static method of the **TileUpdateManager** class can be used to provide the basic XML for each template, which can then be filled in with the values to be displayed. Once the XML is defined, call the **Update** method on an instance of the application's **TileUpdater**, which is retrieved from the **TileUpdateManager** class, with a **TileNotification** object that includes the XML, and optionally an expiration time for the tile. Note that if the app includes a wide tile, the notification should include populated templates for both the square and wide tiles, making it necessary to merge the XML. The following code shows an immediate tile update performed this way for an app that includes both wide and square tiles, with the tile set to expire two hours from when it is first shown (when it expires, it reverts to the app's original tile):

```
// Ensure updates are enabled.
var updater = TileUpdateManager.CreateTileUpdaterForApplication();
if (updater.Setting == NotificationSetting.Enabled)
{
    // Get the default XML for the desired tile template.
    var tileSquareXml =
                TileUpdateManager.GetTemplateContent(TileTemplateType.TileSquareText01);

    // Locate the elements to be modified.
    var textSquareElements = tileSquareXml.GetElementsByTagName("text").ToList();

    // More detailed searching of XML for specific attributes omitted for brevity...
    textSquareElements[0].InnerText = "Heading";
    textSquareElements[1].InnerText = "Subheading 1";
    textSquareElements[2].InnerText = "Subheading 2";
    textSquareElements[3].InnerText = "Subheading 3";

    // Get the default XML for the desired tile template.
    var tileWideXml =
                 TileUpdateManager.GetTemplateContent(TileTemplateType.TileWideText01);

    // Locate the elements to be modified.
    var textWideElements = tileWideXml.GetElementsByTagName("text").ToList();
    textWideElements[0].InnerText = "Wide Heading";
    textWideElements[1].InnerText = "Wide Subheading 1";
    textWideElements[2].InnerText = "Wide Subheading 2";
    textWideElements[3].InnerText = "Wide Subheading 3";

    // Inject the wide binding node contents into the visual element of the Square XML.
    var wideBindingNode = tileWideXml.GetElementsByTagName("binding").First();
    var squareVisualNode = tileSquareXml.GetElementsByTagName("visual").First();
    var importNode = tileSquareXml.ImportNode(wideBindingNode, true);
    squareVisualNode.AppendChild(importNode);

    var notification = new TileNotification(tileSquareXml);
    notification.ExpirationTime = DateTimeOffset.Now.AddHours(2);
    updater.Update(notification);
}
```

Note that this code first performs a check to ensure that updates are enabled. Users can disable tile updates on a tile-by-tile basis by selecting the tile in the Start screen and selecting **Turn live tile off** from the app bar. If a live tile's updates are disabled, they can be re-enabled by selecting **Turn live tile on**.

Note: This process of locating an XML template, looking up the corresponding XML in the MSDN documentation, and writing code based on that schema can be time consuming and prone to error. The App tiles and badges sample published by Microsoft includes the NotificationsExtensions project which can be used to build a reusable WinMD component. The sample can be obtained at http://code.msdn.microsoft.com/windowsapps/app-tiles-and-badges-sample-5fc49148, and instructions for how to use it in a Visual Studio project can be found at http://msdn.microsoft.com/en-us/library/windows/apps/Hh969156.aspx. This component provides a strongly typed object model for populating tile, badge, and toast templates, and as a result also provide IntelliSense and compile-time support to help prevent errors. The remaining discussion and examples in this chapter will make use of this component for its template definitions.

```
// Ensure updates are enabled.
var updater = TileUpdateManager.CreateTileUpdaterForApplication();
if (updater.Setting == NotificationSetting.Enabled)
{
    // Prepare the square tile.
    var tileSquareUpdate =
       NotificationsExtensions.TileContent.TileContentFactory.CreateTileSquareText01();
    tileSquareUpdate.TextHeading.Text = "Heading";
    tileSquareUpdate.TextBody1.Text = "Subheading 1";
    tileSquareUpdate.TextBody2.Text = "Subheading 2";
    tileSquareUpdate.TextBody3.Text = "Subheading 3";

    // Prepare the wide tile.
    var tileWideUpdate =
         NotificationsExtensions.TileContent.TileContentFactory.CreateTileWideText01();
    tileWideUpdate.TextHeading.Text = "Wide Heading";
    tileWideUpdate.TextBody1.Text = "Wide Subheading 1";
    tileWideUpdate.TextBody2.Text = "Wide Subheading 2";
    tileWideUpdate.TextBody3.Text = "Wide Subheading 3";

    // Inject the square tile contents into the wide tile.
    tileWideUpdate.SquareContent = tileSquareUpdate;

    // Send the notification.
    var notification = tileWideUpdate.CreateNotification();
    notification.ExpirationTime = DateTimeOffset.Now.AddHours(2);
    updater.Update(notification);
}
```

The previous code shows the same immediate tile update performed using the **`NotificationExtensions`** helper class.

Queued Tile Updates

In addition to setting a single tile update, it is also possible to queue up to five tiles that Windows will automatically cycle through. This is known as queuing, and simply requires that queuing be enabled. Windows will cycle through the five most recent tile updates, with the exception that tiles with the same **Tag** value will replace each other. The following code shows three tile sets being built up and queuing being enabled through a call to **EnableNotificationQueue**:

```
// Build up a list of tiles to be queued.
// Build the first tiles.
var tileSquareUpdate = TileContentFactory.CreateTileSquareText01();
tileSquareUpdate.TextHeading.Text = "Queue 1";
tileSquareUpdate.TextBody1.Text = "Subheading Q1-1";
tileSquareUpdate.TextBody2.Text = "Subheading Q1-2";
tileSquareUpdate.TextBody3.Text = "Subheading Q1-3";

var tileWideUpdate = TileContentFactory.CreateTileWideText01();
tileWideUpdate.TextHeading.Text = "Wide Queue 1";
tileWideUpdate.TextBody1.Text = "Wide Subheading Q1-1";
tileWideUpdate.TextBody2.Text = "Wide Subheading Q1-2";
tileWideUpdate.TextBody3.Text = "Wide Subheading Q1-3";
tileWideUpdate.SquareContent = tileSquareUpdate;

updater.Update(tileWideUpdate.CreateNotification());

// Build the second tiles.
var tileSquareUpdate2 = TileContentFactory.CreateTileSquarePeekImageAndText01();
// Tile property values omitted for brevity...
var tileWideUpdate2 = TileContentFactory.CreateTileWidePeekImage02();
// Tile property values omitted for brevity...
updater.Update(tileWideUpdate2.CreateNotification());

// Build the third tiles.
var tileSquareUpdate3 = TileContentFactory.CreateTileSquareImage();
// Tile property values omitted for brevity...
var tileWideUpdate3 = TileContentFactory.CreateTileWideImageAndText01();
// Tile property values omitted for brevity...
updater.Update(tileWideUpdate3.CreateNotification());

// Enable queuing.
updater.EnableNotificationQueue(true);
```

Scheduled Updates

The previous sections have discussed making immediate changes to the application tiles. Another option is for the tile to be updated at a later, predetermined time. In this case, the tile XML and a delivery time are used to create a **ScheduledTileNotification**, and instead of calling **Update** on the application's **TileUpdater** instance, the **ScheduledTileNotification** is passed to a call to **AddToSchedule**, as shown in the following code:

```
// Set up the tile that is to appear at a later time.
var tileSquareUpdate = TileContentFactory.CreateTileSquareText01();
// Tile property values omitted for brevity...
```

```
var tileWideUpdate = TileContentFactory.CreateTileWideText01();
// Tile property values omitted for brevity...
tileWideUpdate.SquareContent = tileSquareUpdate;

// Set the time when the update needs to occur as 1 hour from now.
var deliveryTime = DateTimeOffset.Now.AddHours(1);

// Schedule the update.
var scheduledTileNotification =
                  new ScheduledTileNotification(tileWideUpdate.GetXml(), deliveryTime);
updater.AddToSchedule(scheduledTileNotification);
```

Future updates can be cleared by retrieving the scheduled notification from the **`TileUpdater`** instance and calling **`RemoveFromSchedule`**. To help with this, it is possible to specify an Id value for the **`ScheduledTileNotification`** instance, which can then be examined later in the list of pending updates.

```
// Get the list of scheduled notifications.
var scheduledUpdates = updater.GetScheduledTileNotifications();

// Try to find scheduled notifications with a matching Id.
foreach(var scheduledUpdate in scheduledUpdates.Where(x => x.Id == "SomeId"))
{
    updater.RemoveFromSchedule(scheduledUpdate);
}
```

Note: By default, scheduled updates are set to expire in three days in order to help prevent the display of stale content to users. The expiration value can be changed, or set to null to never expire. If queuing is enabled, scheduled updates are added to the end of the queue, and if that results in more than five tiles, the first item is removed from the queue. As noted before, if an existing tile update entry contains a `Tag` attribute, a replacement update with the same `Tag` value will overwrite the existing tile entry.

Periodic Updates

A tile can also be set to be updated at a fixed, repeating time period. For this type of update, instead of supplying XML for the tile, a URI to an HTTP or HTTPS endpoint is specified, which Windows will poll at the indicated interval for the XML to use for the tile's content. Available polling intervals include 30 minutes, one hour, six hours, 12 hours, and daily. The periodic update can be set up with the **`StartPeriodicUpdate`** method on the application's **`TileUpdater`** instance. It is also possible to specify a specific time for when the polling should begin, otherwise the first request will occur immediately. A tile can only have one periodic update interval, though multiple URIs—up to five—can be provided which will be called at the polling interval to provide multiple tiles for display via queuing by using the **`StartPeriodicUpdateBatch`** method. The server can provide a **`Tag`** value for batched tiles by setting the **`X-WNS-Tag`** HTTP response header in the value that is returned from the specified endpoints.

```
// Set up for polling every 30 minutes.
updater.StartPeriodicUpdate(pollingUri, PeriodicUpdateRecurrence.HalfHour);

// Delay the initial request by 1 minute.
var offsetTime = DateTimeOffset.Now.AddMinutes(1);
updater.StartPeriodicUpdate(pollingUri, offsetTime, PeriodicUpdateRecurrence.HalfHour);

// Provide a list of URIs to call.
var batchUris = new []{pollingUri1, pollingUri2, pollingUri3};
// Note that Notification Queuing must be enabled.
updater.EnableNotificationQueue(true);

updater.StartPeriodicUpdateBatch(batchUris, PeriodicUpdateRecurrence.HalfHour);
```

The periodic update can be stopped by calling the **`StopPeriodicUpdate`** method on the application's **`TileUpdater`** instance.

```
updater.StopPeriodicUpdate();
```

Note: Like scheduled updates, periodic updates are initially configured to expire every three days. Expiration values are set in the `X-WNS-Expires` HTTP response header in the value returned from the specified endpoints.

Clearing Tile Contents

The last tile update to be discussed is the ability to clear any tile updates that have occurred and reset the tile to the default tile layout specified in the application manifest file. That can simply be accomplished by calling the **`Clear`** method on the application's **`TileUpdater`** instance.

```
updater.Clear();
```

Badges

On top of containing text and image content, live tiles also host a small piece of status information in the corner opposite the tile's branding (lower right for left-to-right languages). This information is known as a badge, and can either be a number (1–99) or one of a set of provided glyphs. Information conveyed through badges often includes the number of pending items that require the user's attention, such as unread email messages, or perhaps some status information like a problem alert or unavailable network destination.

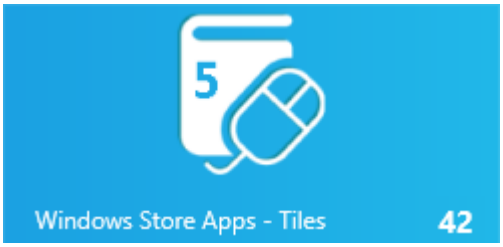


Figure 22: Live Tile Showing a Numeric Badge

Like the live tile content, the badge content is defined through specific XML content. As was previously mentioned, a numeric badge can have a number from 1 to 99, and there are 12 available badge values that can be set, including **none**. The available badge values are defined in the badge image catalog available at http://msdn.microsoft.com/en-us/library/windows/apps/hh761458.aspx. To update a badge value for an application's tile, a **BadgeUpdater** instance for the application is obtained from the **BadgeUpdateManager** class, and the desired badge XML is passed to the **Update** method provided by the **BadgeUpdater** instance.

```
var updater = BadgeUpdateManager.CreateBadgeUpdaterForApplication();

// Prepare a numeric notification and pass the updated badge number to the updater.
var content = new BadgeNumericNotificationContent(42);
var notification = content.CreateNotification();
updater.Update(notification);

// Prepare a glyph notification and pass the updated glyph to the updater.
var content = new BadgeGlyphNotificationContent(GlyphValue.Away);
var notification = content.CreateNotification();
updater.Update(notification);
```

Note: As with the tile content, the NotificationsExtensions library simplifies the process of specifying the badge value without needing to directly work with the XML DOM.

While scheduled updates cannot be set for badges, periodic updates can be configured in a manner that is nearly identical to periodic tile updates, with the exception that there is no provision for batching since there's also no notion of queued badges. Otherwise, the syntax for the call to the application's **`BadgeUpdater`** instance is identical to that of the **`TileUpdater`**.

Likewise, the tile's badge content is independent from the tile contents, so it is cleared independently of the tile contents, though the same **`Clear`** call is used on the **`BadgeUpdater`** instance as is used on the **`TileUpdater`**.

Secondary Tiles

Apps can optionally create additional live tiles known as secondary tiles that can be used to launch the application with parameters for presenting users with a specific set of information. For example, a weather application could create a specific tile that, when used to launch the application, takes users to a display of the weather for a specific city they are interested in. Likewise, Internet Explorer's pinning feature creates secondary tiles that instruct the browser to navigate to specific websites.

Working with Secondary Tiles

The process of adding a secondary tile is known as pinning, and must be initiated programmatically. It results in a system-defined dialog being shown to users for them to approve the addition of the tile. Users can remove a secondary tile at any time directly from the Start screen, and an app can also remove a secondary tile programmatically, though users will be presented with a dialog to confirm the removal. A secondary tile can be created with the following parameters:

- An **`Id`** value to identify the secondary tile.
- A short name to be displayed directly on the tile.
- A display name to be displayed with the tile for tooltips, in the All Programs list, and in Control Panel applications.
- The arguments to be provided to the application when it is activated via this tile.
- An options value to indicate whether the name should be displayed on the square or wide tiles, as well as whether the secondary tile will be shared through the cloud if the app is installed by the user (identified by his or her Microsoft ID) on a different machine.
- URIs to the images that will be placed on the tile (the wide tile logo can be omitted if a wide tile is not desired.)

In addition to the constructor properties, a secondary tile can also be given its own background color.

To pin a new secondary tile, a new **`SecondaryTile`** instance should be created, its properties set, and the new tile's **`RequestCreateAsync`** value should be called.

```
var secondaryTile = new SecondaryTile(
    "TileId",
    "Secondary Tile Sample",
    "Secondary Tile Sample Display Name",
```

```
    "Secondary Tile Activation Args",
    TileOptions.ShowNameOnLogo | TileOptions.ShowNameOnWideLogo,
    new Uri("ms-appx:///Assets/Logo.png"),
    new Uri("ms-appx:///Assets/LogoWide.png"))
    {
        BackgroundColor = Colors.ForestGreen
    };
await secondaryTile.RequestCreateAsync();
```

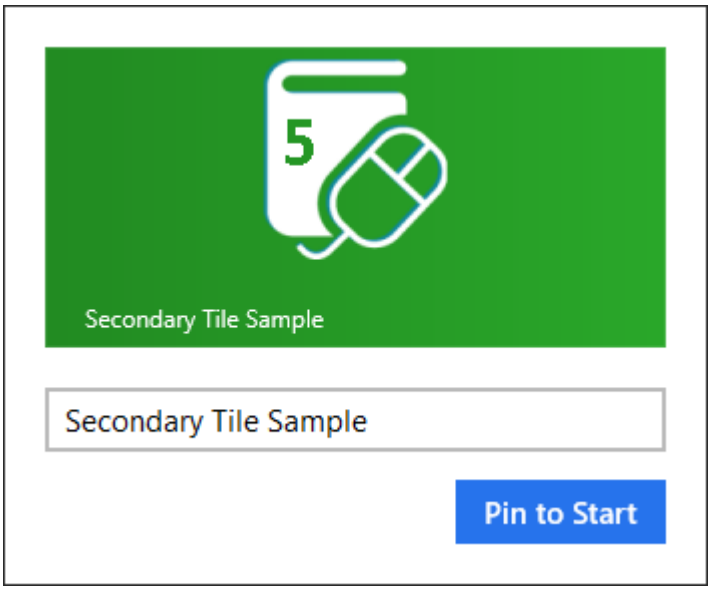


Figure 23: User Dialog Presented when Adding a Secondary Tile

Secondary tiles can be updated in all of the same ways as primary tiles. Instead of obtaining a **`TileUpdater`** or **`BadgeUpdater`** reference by calling the respective **`TileUpdateManager`** **`CreateTileUpdaterForApplication`** and **`BadgeUpdateManager`** **`CreateBadgeUpdaterForApplication`** methods, the **`CreateTileUpdaterForSecondaryTile`** and **`CreateBadgeUpdaterForSecondaryTile`** methods are called with the **`Id`** for the secondary tile to be updated.

Removing secondary tiles programmatically involves locating the tile by its **`Id`** or creating a new **`SecondaryTile`** instance with the same **`Id`**, and then calling the tile's **`RequestDeleteAsync`** method.

```
var matchingTile = new SecondaryTile(SecondaryTileId);
await matchingTile.RequestDeleteAsync();
```

Responding to Secondary Tile Activation

When a user selects a secondary tile from the Start screen, the application goes through the same launch sequence as was outlined in Chapter 3. The application is activated via the Launch contract, which will result in the **`OnLaunched`** method override being called in the application object. An app being launched via one of its secondary tiles can be distinguished from a launch via the primary tile by the value in the launch arguments' **`TileId`** parameter, which will match the value set when the secondary tile was created. Furthermore, the method's arguments will contain a value for the **`Arguments`** property. It is up to the app implementation to decide how the **`TileId`** value and the content of the **`Argument`** parameter should affect application startup, such as by presenting the application's initial page with certain data loaded, or perhaps starting the application in a different page.

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
     // Initialization content omitted for brevity...
     // Handle special launch for the secondary tile.
     if (args.TileId == "TileId")
     {
          if (!rootFrame.Navigate(typeof(SecondaryTilePage), args.Arguments))
          {
               throw new Exception("Failed to create secondary tile page");
          }
     }
}
```

Toast Notifications

Toast notifications are small message pop-ups that can be triggered to appear in the upper corner (upper right for left-to-right languages) of the Windows Start screen over any running applications. They give the application responsible for the toast notification the opportunity to alert users to some event regardless of what app is currently running, and are the only mechanism available for one Windows Store app to interrupt another. Users can react to a toast notification in one of three ways:

- They can ignore the notification, and after a period of time the notification will disappear.
- They can explicitly close the notification by dragging it off the screen or tapping the **X** icon it provides.
- They can use the notification to launch the corresponding app by tapping or clicking within the body of the notification window.

To be able to display toast notifications, the app has to indicate it is **`Toast Capable`** in the manifest file.

Note: Whereas most external activities that can launch an app require an entry in the declarations section of the manifest, toast notifications are enabled in the application UI section.

In addition to the app itself enabling toast notifications, users can disable notifications system-wide for a period of time (one hour, three hours, or eight hours) through the **Notifications** icon in the **Settings** panel. This is quite useful for situations like presentations where unexpected notifications could be annoying, embarrassing, or otherwise problematic. Users can also elect to disable notifications system-wide or for specific apps from within the **Notifications** section of the **PC settings** app.

Raising Toast Notifications

The content that appears in a toast notification is defined in XML in a manner very similar to the way content is defined for a live tile. There are eight available templates—four text-only and four that combine images and text. These templates are defined in the toast template catalog available at http://msdn.microsoft.com/en-us/library/windows/apps/hh761494.aspx. To display a toast notification, a **`ToastNotifier`** instance is obtained from the **`ToastNotificationManager`** class, and the desired toast XML is wrapped in a **`ToastNotification`** instance and passed to the **`Show`** method provided by the **`ToastNotifier`** instance.

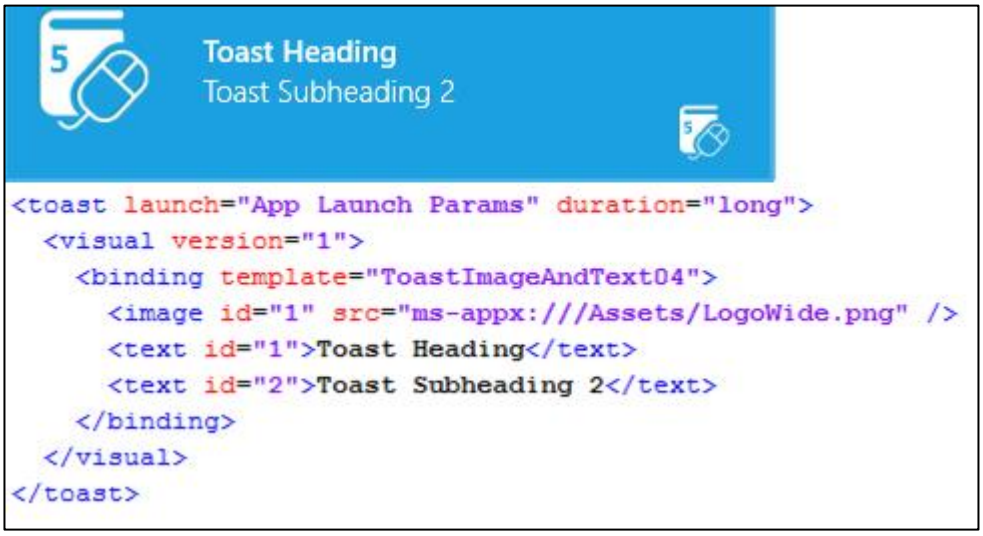


Figure 24: A Toast Notification and Its Corresponding XML

Showing Immediate Notifications

The following code shows the process involved in showing a toast notification:

```
// Ensure updates are enabled.
var notifier = ToastNotificationManager.CreateToastNotifier();
if (notifier.Setting == NotificationSetting.Enabled)
{
    // Build the content of the toast notification.
    var content = ToastContentFactory.CreateToastImageAndText04();
    content.TextHeading.Text = "Toast Heading";
    content.TextBody1.Text = "Toast Subheading 1";
    content.TextBody1.Text = "Toast Subheading 2";
```

```
    content.Image.Src = "ms-appx:///Assets/LogoWide.png";

    // Specify parameters to be passed to the app on launch.
    content.Launch = "App Launch Params";

    // Show the notification.
    var notification = content.CreateNotification();
    notifier.Show(notification);
}
```

Note: For toast notifications that show images for Windows Store apps, the image source must be a URI that uses the http, https, ms-appx, or ms-appdata schemes, and if ms-appdata is used, it must refer to a location in local application data storage (ms-appdata:///local/).

A toast notification can be configured to be shown for either a short duration (seven seconds) or a long duration (25 seconds). Toast notifications displayed for short durations include either the system's default audible alert, or can be set to use one of several alternate system-provided brief alert sounds. Notifications displayed for long durations can optionally use any of these sounds, as well as may opt to play one of several additional system-provided looping sounds that play for the duration of the toast's display. In both cases, an option exists to omit playing any sounds at all. The following code shows a toast notification configured for long duration with a looping sound:

```
// Build the content of the toast notification.
var content = ToastContentFactory.CreateToastImageAndText04();
// Toast contents properties omitted for brevity...

// Specify long duration and a looping sound.
content.Duration = ToastDuration.Long;
content.Audio.Content = ToastAudioContent.LoopingCall2;
content.Audio.Loop = true;

// Show the notification.
var notification = content.CreateNotification();
notifier.Show(notification);
```

Scheduling Toast Notifications

In practice, toast notifications triggered for immediate display like the ones shown so far will not be used directly from application code. For the most part, the application will be interrupting itself, and in these cases, perhaps a message dialog would be a more appropriate way to get users' attention. Instead, these immediate notifications will normally be used from background tasks, which will be discussed in more detail in the next chapter.

An alternative to immediate invocation of a toast notification is to schedule the notification to appear at a pre-scheduled time, perhaps as a reminder for users to return to the app to perform some task. Important notifications can be scheduled to repeat at a predetermined "snooze" interval, so if users dismiss the toast without invoking the application, the toast will reappear some time later. When a snooze interval is set, the maximum number of snoozes must also be set. The following code shows how a scheduled notification can be configured:

```
// Build the content of the toast notification.
var content = ToastContentFactory.CreateToastImageAndText04();
// Toast contents properties omitted for brevity...

// Set the time at which the toast should be triggered.
var deliveryTime = DateTimeOffset.Now.AddMinutes(1);

// Configure a one-time scheduled toast.
var scheduledNotification = new ScheduledToastNotification(
                                    content.GetXml(),
                                    deliveryTime);

// Configure a repeating scheduled toast.
var snoozeInterval = TimeSpan.FromMinutes(5);   // Must be between 60 secs and 60 mins.
var maximumSnoozeCount = (UInt32)5; // Can be 1-5 times.
var scheduledNotification = new ScheduledToastNotification(
                                    content.GetXml(),
                                    deliveryTime,
                                    snoozeInterval,
                                    maximumSnoozeCount);

notifier.AddToSchedule(scheduledNotification);
```

Scheduled updates can be cleared by retrieving the scheduled notification from the **`ToastNotifier`** instance and calling **`RemoveFromSchedule`**. To help with this, it is possible to specify an **`Id`** value for the **`ScheduledToastNotification`** instance, which can then be examined later in the list of pending updates.

```
// Retrieve the scheduled notifications.
var notifier = ToastNotificationManager.CreateToastNotifier();
var notifications = notifier.GetScheduledToastNotifications();

// Alternatively, find the notifications that have a specific Id.
//var notification = new ScheduledToastNotification(content.GetXml(), deliveryTime)
                                        {Id = "ScheduledId "};
//var notifications = notifier.GetScheduledToastNotifications()
                                       .FirstOrDefault(x => x.Id == "ScheduledId");

// Remove the indicated notifications.
foreach (var notification in notifications)
{
    notifier.RemoveFromSchedule(notification);
}
```

Responding to Toast Notification Activations

When users choose to activate the app by tapping or clicking within the body of the toast notification, the application will go through its launch sequence, including running the application's **`OnLaunched`** method. Like launching the app from a secondary tile, this sequence will occur regardless of whether the app is already running. If launch parameters are set on the toast notification, they will be included in the **`Arguments`** parameter of the method's arguments. It is up to the application implementation to decide how these arguments will be used to direct the user interface.

In addition to the activation and launch execution, a running application can subscribe to activation and dismissal events on an immediate notification. Both events return an instance of the notification that is triggering the event, and the dismissed event includes a parameter within its arguments that indicates how the notification was dismissed. Code for subscribing to these events follows:

```
// Build the content of the toast notification.
var content = ToastContentFactory.CreateToastImageAndText04();
// Toast contents properties omitted for brevity...

// Create the notification instance.
var notification = content.CreateNotification();

// Subscribe to the activation event.
notification.Activated += (activatedNotification, args) =>
{
    // Handle the activated event.
};

// Subscribe to the dismissal event.
notification.Dismissed += (dismissedNotification, args) =>
{
    // Handle the dismissed event.
};

// Show the notification.
notifier.Show(notification);
```

Note: These events are not available on scheduled notifications, so it is not currently possible to start an app, immediately query the list of outstanding scheduled notifications, and set up in-app event handlers for them.

Push Notifications

Up to this point, this chapter has focused on live tile updates and toast notifications that are initiated by the app itself, either immediately or through the use of some of the available scheduling options. In addition to these approaches, Windows Store apps can also make use of a mechanism known as a push notification to allow events that occur outside of the machine to to display tile updates or toasts through services managed by Windows. These push notifications extend the ability for an app to provide additional interaction with users, regardless of whether the app is running.

Note: As mentioned previously, background tasks can also offer some options for tile and toast updates to be updated when the app itself is not running. In some cases, push notifications offer a few advantages over using polling within background tasks since the action is server-initiated instead of polled, and execution time constraints are not a concern with push notifications. Nonetheless, background tasks do offer some interesting options for extending an app's functionality. Background tasks will be discussed further in the next chapter.

Push notifications are powered by a combination of a local service managed by Windows and a Microsoft-provided, cloud-based service known as the Windows Notification Service (WNS). The following diagram illustrates the sequence for working with push notifications:

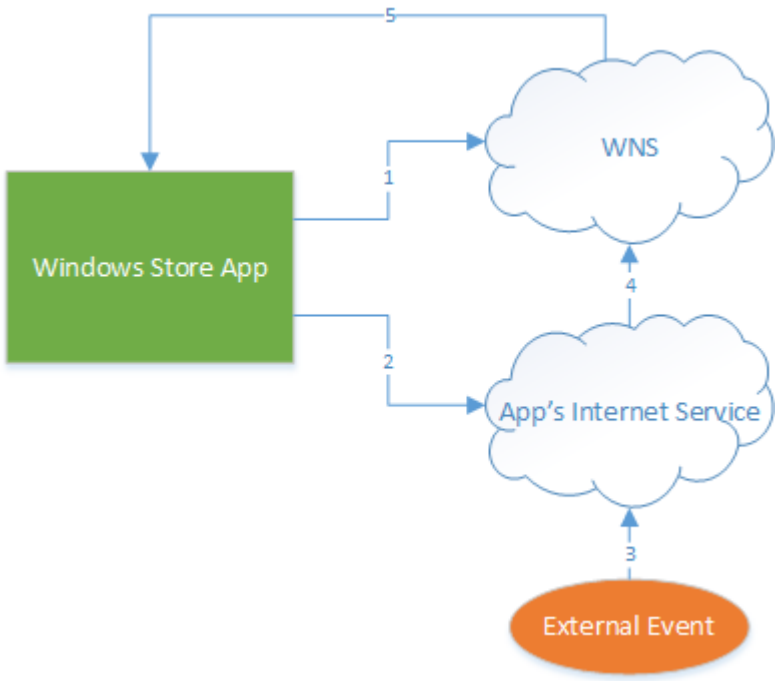


Figure 25: Push Notification Life Cycle

This diagram illustrates the following steps:

1. The Windows Store app requests and is issued a push notification channel.

2. The Windows Store app provides this channel to the Internet service that will be sending the notification, which stores the channel information along with any other relevant information for later use.
3. Some event occurs which causes the Internet service to decide it is appropriate to send a push notification to its subscribers.
4. The Internet service uses some criteria to identify which subscribers should receive a notification, gathers the list of channels for those subscribers, and sends the notification to the Windows Notification Service in the form of an XML payload by calling each of the channel URIs.
5. If the channel is still valid, the Windows Notification Service routes the notification to the appropriate device, where the notification payload and device settings determine how the notification should be displayed.

To provide some context, if a sports app were to provide users with the ability to receive push notifications for news related to their favorite teams, the app would first obtain a channel. The app would then upload that channel URI plus some information to describe the user's favorite teams to a sports Internet service related to that sports app, which would store that information internally. The sports service would then receive score and news updates in some manner. As each notification-worthy update arrived, the sports service would first build the appropriate notification XML. It would then query the notification information it had stored in order to identify channels where the favorite teams matched the incoming news. The sports service would then issue the notification XML to the WNS by using each one of these channels, and WNS in turn would provide the notification to the corresponding device, potentially displaying a toast message or a tile update indicating a change in the game score.

Note: The functionality offered by push notifications and the related services for Windows Store apps is very similar to the push notification functionality offered for Microsoft's Windows Phone. While there are some small differences between the two, an understanding of one will help form an understanding of the other.

There are four kinds of push notifications available that can be sent to Windows Store apps. Tile, badge, and toast notifications are the same as the app-initiated notifications that have been discussed so far. In addition to these, push notifications include a raw notification option. A raw notification simply includes custom text content that can be sent to the Windows Store application—it is up to the application to determine how best to process the text. In order to interact with a raw notification, either the app has to be running, or a background task has to be configured to handle the notification on the app's behalf. Background tasks will be discussed in the next chapter.

Now that the background of push notifications has been presented, the specific details for each of the steps involved will be discussed.

Configuring an App for Push Notifications

In order for a Windows Store app to participate in push notifications, it must first be registered with the Windows Store. Information related to this registration process will be presented here, while a more complete discussion of registering an app with the Windows Store will be included in the subsequent chapter concerning app deployment.

The simplest way to associate an app with the Windows Store is to select the **Associate App with the Store** menu option from the project's **Store** context menu command in Visual Studio:

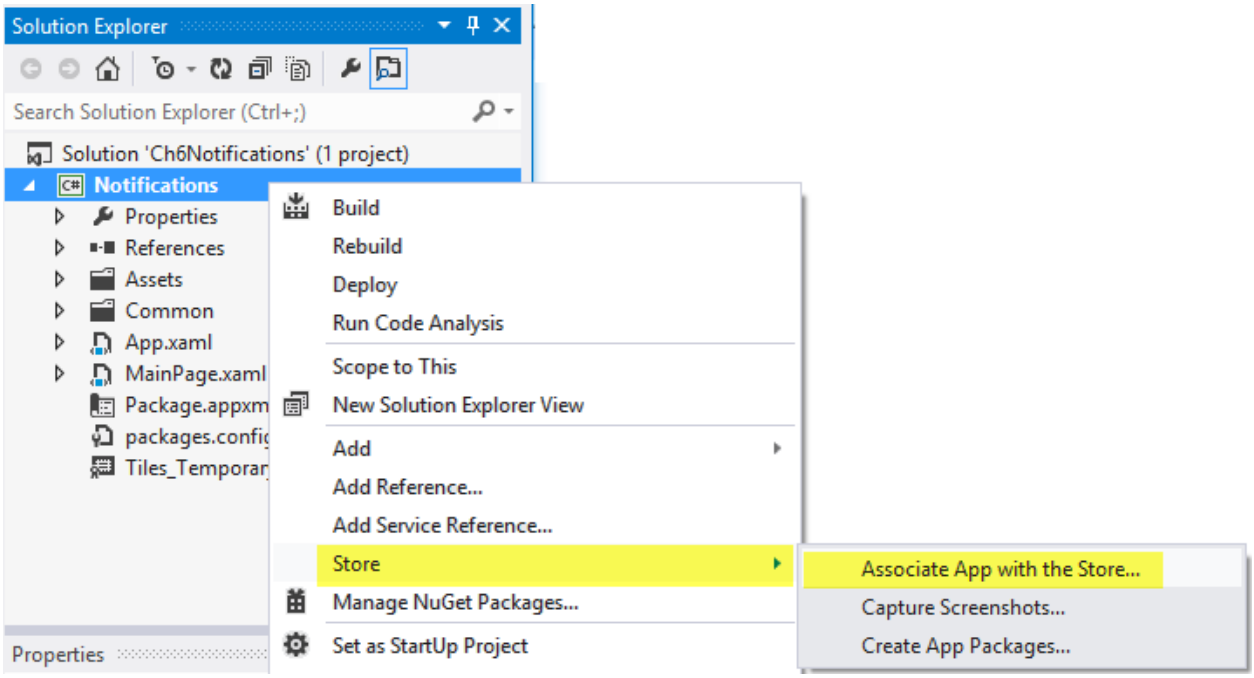


Figure 26: Associating an App with the Windows Store from Visual Studio

This will start the **Associate Your App with the Windows Store** wizard. The wizard will prompt for Windows Store credentials (discussed later), and then list the apps that have been registered in the Windows Store with those credentials. Once an app has been selected, the final page of the wizard will list changes that will be made to the app as a result of completing the wizard. These changes typically include:

- Updating the package name to a store-determined unique name.
- Updating the package's digital signature file to a new file that includes a store-determined Publisher ID value.

Once the app's manifest has been configured to be associated with an app registration in the Windows Store, the app can request a notification channel. The notification channel includes a URI uniquely associated with an installation of the app for a user on the machine. This channel is returned by a call to the **`CreatePushNotificationChannelForApplicationAsync`** method of the **`PushNotificationChannelManager`** class. A notification channel URI has the format **`https://xxx.notify.windows.com/?token=<token>`**, where the **`xxx`** value is determined by the WNS service, and the token is also provided by the WNS service.

The notification channel includes properties for both the channel URI and its expiration. It is important to note that the channels provided do expire after a period of time—currently 30 days—so an app must periodically update the Internet service issuing push notifications. To do this, it is a good idea to request a new channel each time the app is invoked, as well as to cache the channel URI value to see if the value returned when the app is invoked has already been sent to the Internet service. This latter step helps to reduce the amount of unnecessary calls to the Internet service by only making the update call when a change has been detected. The following code shows an implementation for this process using the **`ApplicationData LocalSettings`** storage:

```
// Request the channel.
var channel = await
     PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync();
//var secondaryTileChannel = await PushNotificationChannelManager
//                      .CreatePushNotificationChannelForSecondaryTileAsync("Tile Id");

// Retrieve the previous published channel Uri (if any) from settings.
var oldChannelUri = String.Empty;
var localSettings = ApplicationData.Current.LocalSettings;
if (localSettings.Values.ContainsKey("previousChannelUri"))
{
    var oldChannelUriValue = localSettings.Values["previousChannelUri"];
    oldChannelUri = oldChannelUriValue == null
        ? String.Empty
        : oldChannelUriValue.ToString();
}

// Send an updated channel Uri to the service, if necessary.
if (!oldChannelUri.Equals(channel.Uri, StringComparison.CurrentCultureIgnoreCase))
{
    // The Channel has changed from the previous value (if any). Publish to service.
    PublishChannelToService(channel.Uri);

    // Cache the sent Uri value.
    localSettings.Values["previousChannelUri"] = channel.Uri;
}
```

Note: The notification channel is actually associated with a particular application tile. It is possible to obtain a notification channel for a secondary tile by calling `CreatePushNotificationChannelForSecondaryTileAsync` instead of `CreatePushNotificationChannelForApplicationAsync`. With such a channel, a tile or badge update notification can be sent which targets a particular tile.

Once a Windows Store app has obtained a notification channel URI and shared it with the Internet service, perhaps along with other relevant data that helps the service decide when it is time to send a notification to a particular channel, the service is ready to start sending push notifications.

Sending Push Notifications

When it is time for the Internet service to send a notification to a user, the service will issue an HTTP POST request to the Windows Notification Service using the notification channel URI supplied by the Windows Store app. The content of the POST request includes some authentication information, as well as the XML payload that describes the toast, tile, badge, or raw notification.

The Windows Notification Service requires that calls to issue push notifications include an access token that must be obtained by authenticating with the WNS. This authentication requires that the caller supply the application's Package Security Identifier (Package SID) and client secret values available from the Windows Store entry for the app. These values are available by selecting the app in the **Dashboard**, navigating to the **App Name** > **Advanced Features** page, and then the **Push Notifications and Live Connect Services Info** link. Note that one access token can be used for multiple notification calls; it is not necessary to obtain a new one for every channel URI being called—though the tokens do expire, so it may be necessary to periodically refresh them.

Once the access token is available, the process of sending a notification includes preparing the XML payload, which is identical to the client-side XML for notifications that has been discussed so far, followed by simply issuing the request. Once the notification has been issued, the response codes should be examined so that expired or invalid notification channel URIs can be removed or otherwise marked to ensure they are not called again. A service that repeatedly attempts to issue notifications to bad endpoints may be throttled to prevent it from issuing notifications. Furthermore, a check should be made prior to sending a notification to ensure that the URI is actually a notification URI, and not a malicious attempt to intercept data destined for a user by providing an alternate URI to the Internet service. All valid notification channel URIs will use the domain notify.windows.com.

Note: Since the information exchange with the WNS is accomplished using Internet-standard protocols, the Internet service that triggers the notification does not have to be implemented in .NET. For .NET implementations, the Microsoft DPE team has released a helper library similar to the previously discussed NotificationsExtensions library that provides a strongly typed object model for building and sending tile, badge, toast, and raw notifications (and also provides IntelliSense and compile-time support as well). This library is available as a NuGet package that can be obtained at http://www.nuget.org/packages/WnsRecipe (for more information about including NuGet packages, please refer to http://docs.nuget.org/docs/start-here/overview). The remaining discussions and examples in this chapter will make use of this component for preparing and sending push notifications. For information on building and issuing notifications by hand, please refer to the MSDN documentation at http://msdn.microsoft.com/library/windows/apps/xaml/Hh868244.aspx.

A typical sequence for sending notifications to endpoints that match some particular criteria from a .NET service (perhaps a WCF service) is shown in the following sample. Note that only one tile, toast, badge, or raw notification can be sent at a time. The creation of multiple notifications in this sample is simply to illustrate the various factory classes that can be used.

```
// Set up the token provider to provide the access token for notification requests.
```

```
var tokenProvider = new WnsAccessTokenProvider(CliendSID, ClientSecret);

var matchingSubscriptions = Storage.GetMatchingEndpoints(dataOfInterest);
foreach(var matchingSubscription in matchingSubscriptions)
{
    // Note: Toast, tile, badge and raw notification configurations are being shown
    // with the same 'notification' variable for illustration purposes only.

    // Toast
    var notification = ToastContentFactory.CreateToastText01();
    // Detailed configuration of Toast contents omitted for brevity...

    // Tile
    var notification = TileContentFactory.CreateTileWideText01();
    // Detailed configuration of Tile contents omitted for brevity...

    // Badge
    var notification = BadgeContentFactory.CreateBadgeNumeric();
    // Detailed configuration of Badge contents omitted for brevity...

    // Raw
    var notification = RawContentFactory.CreateRaw();
    notification.Content = "Some raw content";

    // To prevent data hijacking, ensure that the host contains "notify.windows.com".
    var uri = new Uri(matchingSubscription.SubscriptionUri);
    if (!uri.Host.ToLowerInvariant().Contains("notify.windows.com"))
    {
        // Bad URI - Possible data hijack attempt. Remove the subscription.
        Storage.RemoveBadSubscription(matchingSubscription);
        continue;
    }

    // Send the notification and examine the results.
    var result = notification.Send(uri, tokenProvider);
    switch (result.StatusCode)
    {
        case HttpStatusCode.Gone:
            // The Channel URI has expired. Don't send to it anymore.
            Storage.RemoveBadSubscription(matchingSubscription);
            break;
        case HttpStatusCode.NotFound:
            // The Channel URI is not a valid URI. Don't send to it anymore.
            Storage.RemoveBadSubscription(matchingSubscription);
            break;
        case HttpStatusCode.NotAcceptable:
            // The channel throttle limit has been exceeded.
            // Stop sending notifications for a while.
            break;
        //case HttpStatusCode.Unauthorized:
        //    // The access token has expired. Renew it.
        // NOTE: Not needed - handled internally by helper library.
        //    tokenProvider = new WnsAccessTokenProvider(CliendSID, ClientSecret);
        //    break;
    }
    // TODO - Based on StatusCode, etc. log issues for troubleshooting & support
}
```

If a user is offline when a toast message is sent, the notification is dropped. The most recent tile and badge notifications will be cached by the service, with the number of tile notifications that make it to the client depending on whether queuing is enabled. By default, raw notifications are not cached.

Note: Microsoft has recently announced the Windows Azure Mobile Services (WAMS). Currently available in preview form, the service promises to provide the opportunity for mobile developers to quickly stand up cost-effective back-end support services for their apps, featuring "No hassles, no deployments, and no fear." As of the time of this writing, WAMS includes built-in support for cloud-based data access, user authentication, and building and sending push notifications to Windows Store apps. Additional information about WAMS, including a tutorial that includes configuring push notifications, can be found at http://www.windowsazure.com/en-us/develop/mobile/.

Interacting with Push Notifications from the App

The push notification channel provides an event that can be used to notify the app when a notification occurs on that channel. There are two general scenarios this event can be used for. First, it provides an app a way to be notified that a raw notification has been received in order for the app to process that notification as appropriate. Second, the app has the opportunity to cancel Windows' default processing of a notification (for example, to suppress displaying a toast notification that is intended to call attention to new data arriving for an app if the new data will automatically be displayed in the running app's user interface.)

The following code shows this event being used with a channel when the channel is obtained during app startup:

```
// Subscribe to the notification being received while the app is running.
channel.PushNotificationReceived += (o, e) =>
{
    if (e.NotificationType == PushNotificationType.Raw && e.RawNotification != null)
    {
        // TODO - Process the supplied raw notification contents
        DoSomethingWithRawNotificationContent(e.RawNotification.Content);
    }
    else
    {
        // Illustrates cancelling Windows' default handling of the notification.
        e.Cancel = true;
    }
};
```

Recap

This chapter examined the use of tiles and toasts to add additional dimensions to the ways in which a Windows Store app can appear to interact with users, even when the app itself isn't actively running. This included mechanisms for working with both primary and secondary live tiles, showing toast notifications to call a user's attention to one app while working in another, and using push notifications to allow actions occurring outside of the user's machine to work with these tiles and toasts.

The next chapter will introduce the facilities exposed to allow Windows Store apps to perform tasks in the background when the main app itself isn't activated through the Background Transfer and Background Task mechanisms.

Chapter 6 Hardware and Sensors

While the chapter on Contracts and Extensions discussed using a printer from a Windows Store app, this is only one of several different opportunities available for a Windows Store app to take advantage of the increasing wealth of hardware resources typically connected to today's tablet, laptop, and desktop computers. Several APIs are available that can be used by Windows Store apps to provide easy access to many of the hardware resources connected or installed in the machine on which the apps are running. These tools help enhance Windows Store apps by allowing them to provide richer, more immersive experiences where the apps can obtain information from or even be designed to react to information in the physical environment in which they're being used.

This chapter will present several common ways that Windows Store apps can incorporate hardware integration, starting with obtaining data from sensors that measure the various forces acting on the device running the apps. This will be followed by a discussion of the facilities for obtaining a device's geographic location. The chapter will conclude with a discussion of multimedia integration focused on integrating with cameras and microphones.

Note: Not all devices will have access to all sensors—many laptops may lack orientation sensors, and desktops with GPS sensors may be understandably rare. While some of the sensors have fallback implementations that use alternate means to obtain information, such as location sensors using network information to calculate a device's position, availability is ultimately up to the combination of the functional elements installed in the device by the manufacturer or those that are added onto the device after the fact. Because Windows Store apps are intended to run on a variety of devices and form factors, well-built apps should take into account whether the desired devices are available and provide alternate functionality when possible and when it makes sense to do so.

Interacting with Sensors

Sensors are used to measure the external forces exerted on a device. The WinRT API provides Windows Store apps with the ability to interact with several different kinds of sensors that measure forces such as movement in 3-D space, angular velocity, compass direction, pitch, roll, and yaw, and also the intensity of light that the device is being exposed to. This support is provided by the classes in the **`Windows.Devices.Sensors`** namespace.

The following list summarizes the sensors that are available and what they measure:

- **`SimpleOrientationSensor`**: Returns basic device orientation, expressed via an enumeration that indicates to which edge the device has been rotated, or whether the device is simply lying face-up or face-down where it isn't sitting on any particular vertical edge.

- **`Accelerometer`**: Measures the forces acting on the device in the x, y, and z directions, expressed in terms of Gs. The accelerometer also can raise an event indicating if the device is being shaken.
- **`Compass`**: Indicates the orientation of the device in degrees relative to both magnetic and true north.
- **`Gyrometer`**: Indicates the angular (rotational) velocity of the device in degrees per second about the x, y, and z axes.
- **`Inclinometer`**: Indicates the pitch, roll, and yaw of the device in degrees.
- **`LightSensor`**: Indicates the amount of light that the surface of the device is exposed to, expressed in lux.
- **`OrientationSensor`**: Sophisticated sensor that returns quaternion and rotation matrix data for the device.

These sensors are all accessed in very similar manners; the main difference in the interaction is the precise contents of the measured forces that are returned. A reference to the sensor (if present) is obtained through a **`GetDefault`** method. From there, the sensor reference can be interrogated directly to obtain its present value, or the app code can subscribe to a **`ReadingChanged`** event. All of the sensors except the simple orientation sensor expose **`ReportInterval`** and **`MinimumReportInterval`** properties that indicate in milliseconds how often the event will be raised (reported)—the **`ReportInterval`** property must be set to at least the read-only **`MinimumReportInterval`** value that is provided. If zero is provided as the **`ReportInterval`**, the device driver default value is used as the interval.

```
// Values will be null if no such sensor is available.
// Null checks omitted for brevity.
var simpleOrientation = Windows.Devices.Sensors.SimpleOrientationSensor.GetDefault();
var accelerometer = Windows.Devices.Sensors.Accelerometer.GetDefault();
var compass = Windows.Devices.Sensors.Compass.GetDefault();
var gyrometer = Windows.Devices.Sensors.Gyrometer.GetDefault();
var inclinometer = Windows.Devices.Sensors.Inclinometer.GetDefault();
var orientationSensor = Windows.Devices.Sensors.OrientationSensor.GetDefault();
var lightSensor = Windows.Devices.Sensors.LightSensor.GetDefault();


// Obtaining immediate readings for the sensor values.
var orientation = simpleOrientation.GetCurrentOrientation();
var accelerometerReading = accelerometer.GetCurrentReading();
var compassReading = compass.GetCurrentReading();
var gyrometerReading = gyrometer.GetCurrentReading();
var inclinometerReading = inclinometer.GetCurrentReading();
var orientationReading = orientationSensor.GetCurrentReading();
var lightSendorReading = lightSensor.GetCurrentReading();


// Subscribing to sensor value change events.
simpleOrientation.OrientationChanged += HandleSimpleOrientationReadingChanged;

//_accelerometer.ReportInterval
//    = Math.Max(_accelerometer.MinimumReportInterval, desiredReportInterval);
_accelerometer.ReadingChanged += HandleAccelerometerReadingChanged;
// ...etc...
```

```
// Special "shaken" event exposed by the accelerometer.
_accelerometer.Shaken += HandleAccelerometerShaken;
```

When using sensor events, it is important to be mindful of the frequency at which these events are being raised, as there is a cost in battery life and CPU load related to handling these events. When an application is suspended, the event handlers are dormant until the application is resumed—the events are not queued up while the app is suspended. Finally, it is important to note that the events are not raised on the application's UI thread, so if user interface elements are to be updated as a result of these events, the calls must be properly marshalled, which is accomplished with the **`Dispatcher`** object.

```
private async void  HandleSimpleOrientationReadingChanged (
                            SimpleOrientationSensor sender,
                            SimpleOrientationSensorOrientationChangedEventArgs args)
{
    // Marshal the call to display sensor content to the UI thread with the Dispatcher.
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
    {
        DisplaySimpleOrientationReading(args.Orientation, args.Timestamp);
    });
}

private async void HandleAccelerometerReadingChanged(
                                    Accelerometer sender,
                                    AccelerometerReadingChangedEventArgs args)
{
    // Marshal the call to display sensor content to the UI thread with the Dispatcher.
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
    {
        DisplayAccelerometerReading(
            args.Reading.AccelerationX,
            args.Reading.AccelerationY,
            args.Reading.AccelerationZ,
            args.Reading.Timestamp
            );
    });
}
```

Determining a Device's Location

In addition to determining the forces acting on a device, it is also possible for Windows Store apps to obtain the device's geographic position, as well as to receive updates as the position changes. Internally, Windows makes use of the Windows Location Provider, which coordinates a combination of triangulation of known Wi-Fi hotspots and IP address lookup. The WinRT Location APIs internally examine the Windows Location Provider and any installed GPS hardware to determine which is currently capable of supplying the most accurate data (usually GPS, though when indoors a GPS sensor may not be able to obtain satellite data). For app developers, this process is relatively transparent.

Protecting Users' Privacy

It is important to note that location information is considered to be personally identifiable information. As a result, there are several mechanisms and policies in place that exist to help protect app users' privacy and keep users in control over how such information is gathered and exchanged. Apps that intend to use the Location APIs are required to declare the **`Location`** capability in the application manifest. When an app attempts to access location information through these APIs for the first time following installation, users will be prompted as to whether access to location information should be allowed. This setting will be reflected within a **Location** entry in the **Permissions** panel that is automatically added to apps obtained from the Windows Store, and can be accessed through the app's **Settings** charm. Because a user is prompted through a message box, the initial access of the Location APIs must occur on the application's UI thread and cannot be part of a background task. Users may use this setting to disable or re-enable the location setting for an app at any time. Finally, within the **Privacy** section of the Windows PC settings, users may elect to completely disable all apps' access to location information.

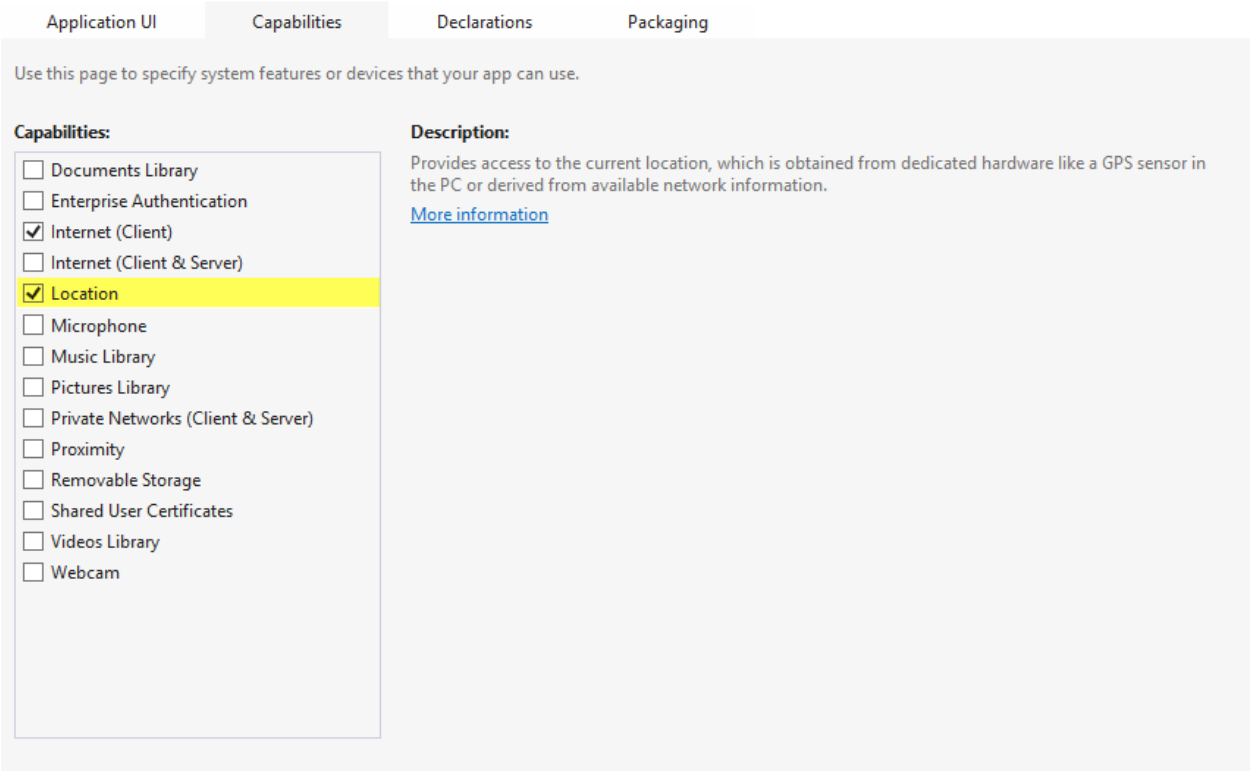


Figure 27: Application Capabilities Screen with the Location Capability Selected

Obtaining Location Information

Location information is accessed through the **`Geolocator`** class in the **`Windows.Devices.Geolocation`** namespace. The **`GetGeopositionAsync`** method asynchronously returns a **`Geoposition`** value, which includes a **`Coordinate`** property that returns a **`Geocoordinate`** component. The **`Geocoordinate`** value includes:

- The latitude and longitude in degrees.
- The altitude in meters (if available).
- The speed in meters per second and heading in degrees from true north (if available).

The following code illustrates directly requesting position information from the **Geolocator**. Note that the call to obtain the information is actually deferred until after it is certain the locator is ready for interaction by waiting for the proper status to be returned from the **StatusChanged** event. It is also important to note that if the **Geolocator** is in a **Disabled** state due to users disabling location information either at the app or system levels, a call to **GetPositionAsync** will result in an **AccessDeniedException** being thrown.

```
// Obtain a reference to the Geolocator.
var locator = new Windows.Devices.Geolocation.Geolocator();
// Wait for the locator to achieve a "ready" state before requesting information.
locator.StatusChanged += async (o, args) =>
{
    if (args.Status == Windows.Devices.Geolocation.PositionStatus.Ready)
    {
        // Obtain the position information.
        var position = await locator.GetGeopositionAsync();

        var latitude = position.Coordinate.Latitude;
        var longitude = position.Coordinate.Longitude;
        var altitude = position.Coordinate.Altitude;
        var speed = position.Coordinate.Speed;
        var heading = position.Coordinate.Heading;

        // Use the position information to update a map, geotag a photo, etc.
    }
};
```

It is also possible to register to receive **PositionChanged** events from the **Geolocator** when its location value changes. The **Geolocator** uses two settings to help determine when to raise these events. The first setting is defined in the **MovementThreshold** property and indicates in meters the minimum distance change that must occur before the event is raised. The second property is set via the **ReportInterval** property, and identifies the minimum number of milliseconds that need to elapse between event notifications.

Note: In order to conserve resources, the `Geolocator` defaults to a mode in which position information is not obtained using GPS hardware. This mode provides potentially less accurate data, but also optimizes power consumption. If the Windows Store app requesting the location information requires more accurate data, the app should set the `DesiredAccuracy` value for the `Geolocator` to `High`. This value will potentially engage any available GPS hardware to provide higher-accuracy data. Apps that are listening for `PositionChanged` events will likely be interested in this higher-accuracy data, and should consider using the `High DesiredAccuracy` setting.

The following code shows subscribing and reacting to the **PositionChanged** event. Note that this event is not guaranteed to return on the application's UI thread, so if UI updates are being made as a reaction to this event, it will be necessary to asynchronously marshal the UI update call to the appropriate thread.

```
// Set minimum threshold in meters.
locator.MovementThreshold = 20.0;
// Set minimum interval between updates in milliseconds.
locator.ReportInterval = 3000;

// Subscribe to the PositionChanged event.
locator.PositionChanged += async (o, args) =>
{
    // Marshal UI calls using the Dispatcher.
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
        () =>
        {
            var latitude = args.Position.Coordinate.Latitude;
            var longitude = args.Position.Coordinate.Longitude;
            var altitude = args.Position.Coordinate.Altitude;
            var speed = args.Position.Coordinate.Speed;
            var heading = args.Position.Coordinate.Heading;

            // Use the position information to update a map, geotag a photo, etc.
        });
};
```

Note: Apps will not receive `PositionChanged` events while suspended. When an app resumes from suspension, it will only receive new events—`PositionChanged` events are not queued while the app is in a suspended state.

Tip: For map-based apps, the Bing Maps SDK for Windows Store apps includes a map control that can be used in XAML applications and offers a tremendous amount of functionality. Along with some documented configuration steps required to use this control, developers must register to obtain a key at the Bing Maps Account Center, and the key must be provided to the map control through its `Credentials` property. Instructions for obtaining this SDK and configuring an app to use this control can be found at http://msdn.microsoft.com/en-us/library/hh846481.aspx.

Using the Simulator to Emulate Position Changes

The debugging simulator that was introduced in Chapter 1 includes functionality for simulating updates in position information. Tapping the **Set Location** button (it looks like a globe) in the simulator's margin will bring up the **Simulated Location** dialog. When **Use simulated location** is selected, it is possible to enter latitude, longitude, and altitude information that will be sent to the location API as a position update when the **Set Location** button is tapped. This allows easier testing and debugging of location-aware applications.

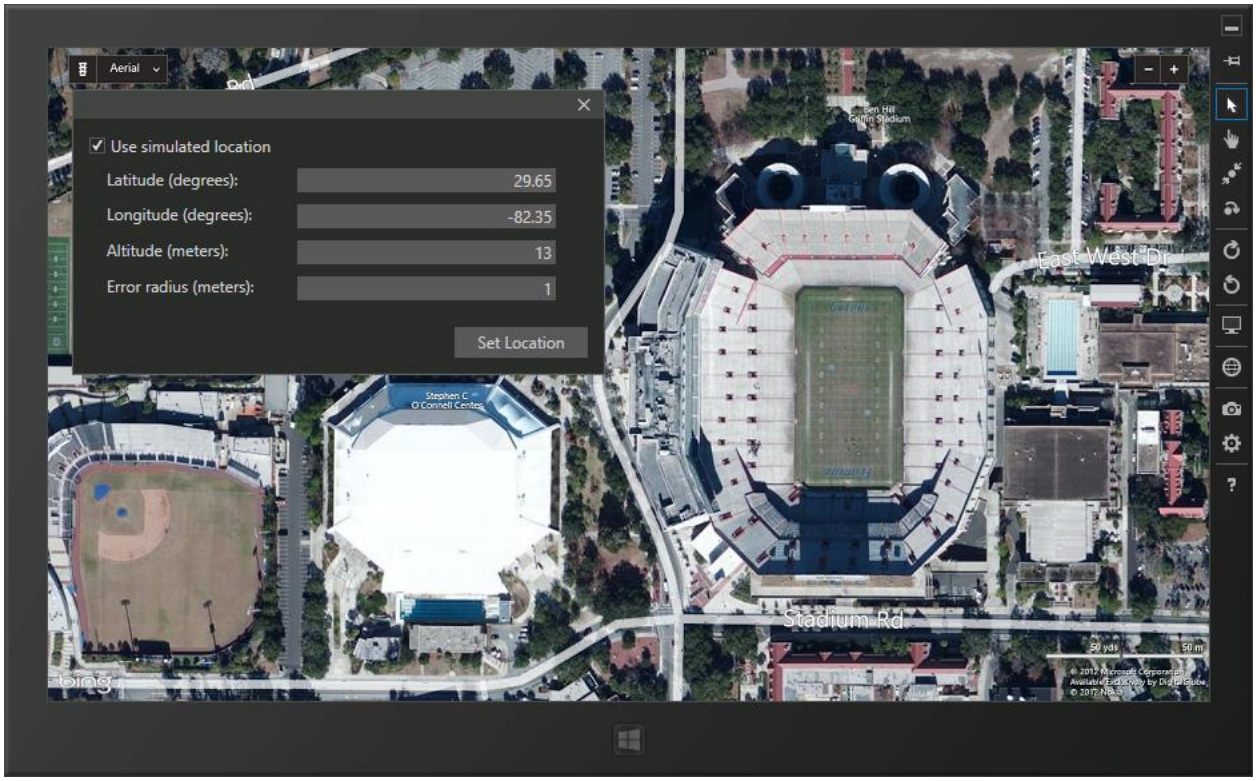


Figure 28: Using the Debugging Simulator's Location Simulation Tools with a Map Application

Multimedia Integration with Cameras and Microphones

Windows Store apps have access to a rich set of media APIs. While this section will briefly discuss playback as a side effect of a broader discussion of multimedia capture, the available set of APIs include tools for:

- Consuming (playing back) multimedia content, including working with the playback stream to manipulate levels or add effects.
- Acquiring multimedia content.
- Accessing and interacting with multimedia hardware devices.
- Creating and manipulating playlists to provide sequenced multimedia playback.
- Streaming multimedia content to Play To-enabled devices.
- Interacting with multimedia content protected with digital rights management (DRM) tools.
- Converting multimedia content between different formats.

Note: Beyond these high-level concepts, there are many advanced scenarios that can be developed through integration with DirectX and with the Media Foundation Pipeline that are well outside the scope of this book. Additional information on advanced multimedia integration with Windows Store apps can be found in the MSDN documentation.

Protecting Users' Privacy

Access to a user's camera and microphone is considered to be access to "sensitive" devices—those that may provide access to a user's personal data. Like location sensors, any app that makes use of the APIs that interact with cameras and microphones must first declare the appropriate capabilities in the application manifest file, where each webcam and microphone are listed as available capabilities. When either or both of these capabilities is indicated, the first time these devices are accessed users will be prompted to confirm that they are comfortable allowing such access (note that this includes threading concerns similar to those discussed in relation to user location information). Likewise, the **Permissions** panel in the application's **Settings** panel will include an entry for **Webcam**, **Microphone**, or **Webcam and microphone** that users may change at any time to disable access to the corresponding hardware.

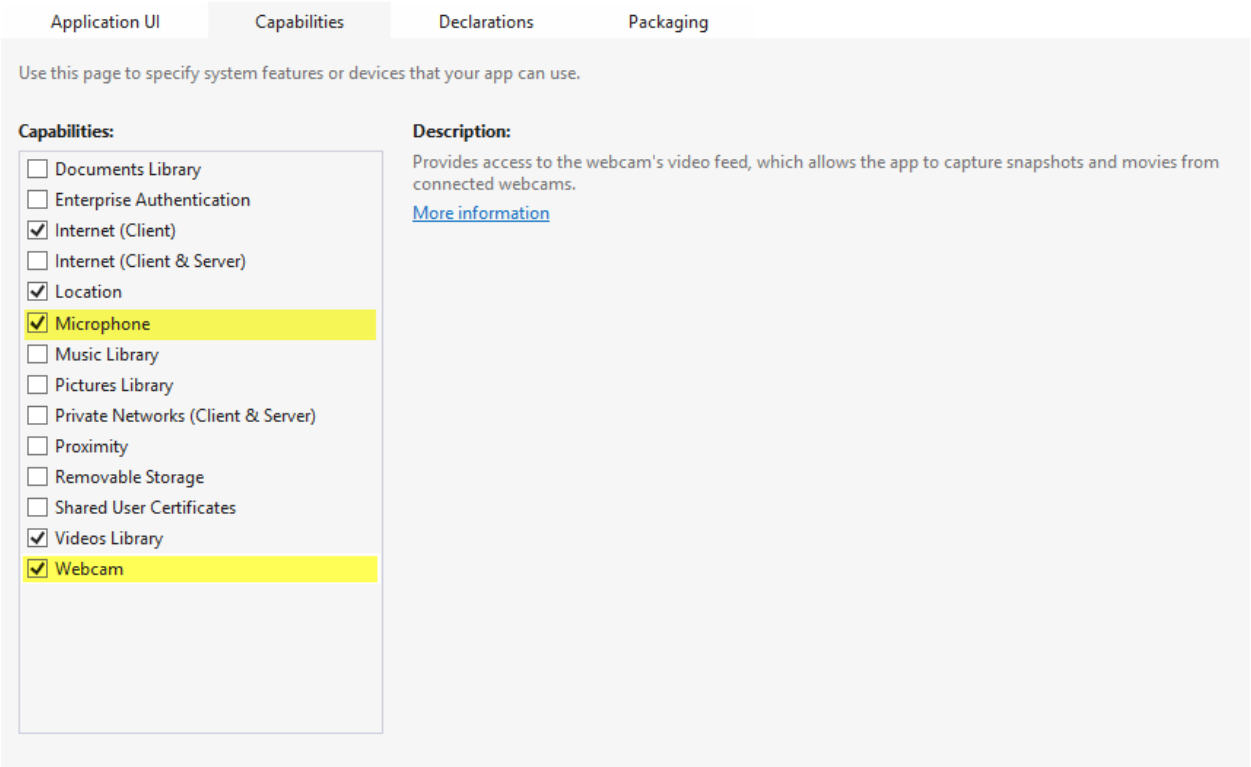


Figure 29: Application Capabilities Screen with the Microphone and Webcam Capabilities Indicated

Capturing Video with the CameraCaptureUI

The simplest way for a Windows Store app to obtain pictures, audio, and videos from a user's webcam and microphone is through the use of the **`CameraCaptureUI`** class. This class provides the ability to invoke a pre-built user interface for capturing pictures and videos with support for configuration options, including:

- Customization of the resulting picture file's format (JPG, PNG, JPG-XR) and max resolution.
- Support for setting properties for the built-in, on-screen cropping interface.
- Selection of the resulting video file format (MP4, WMV).
- Setting the maximum video capture duration.
- Setting the maximum supported video resolution (high definition, standard definition, low definition, and highest available.)
- Support for built-in, on-screen video trimming.

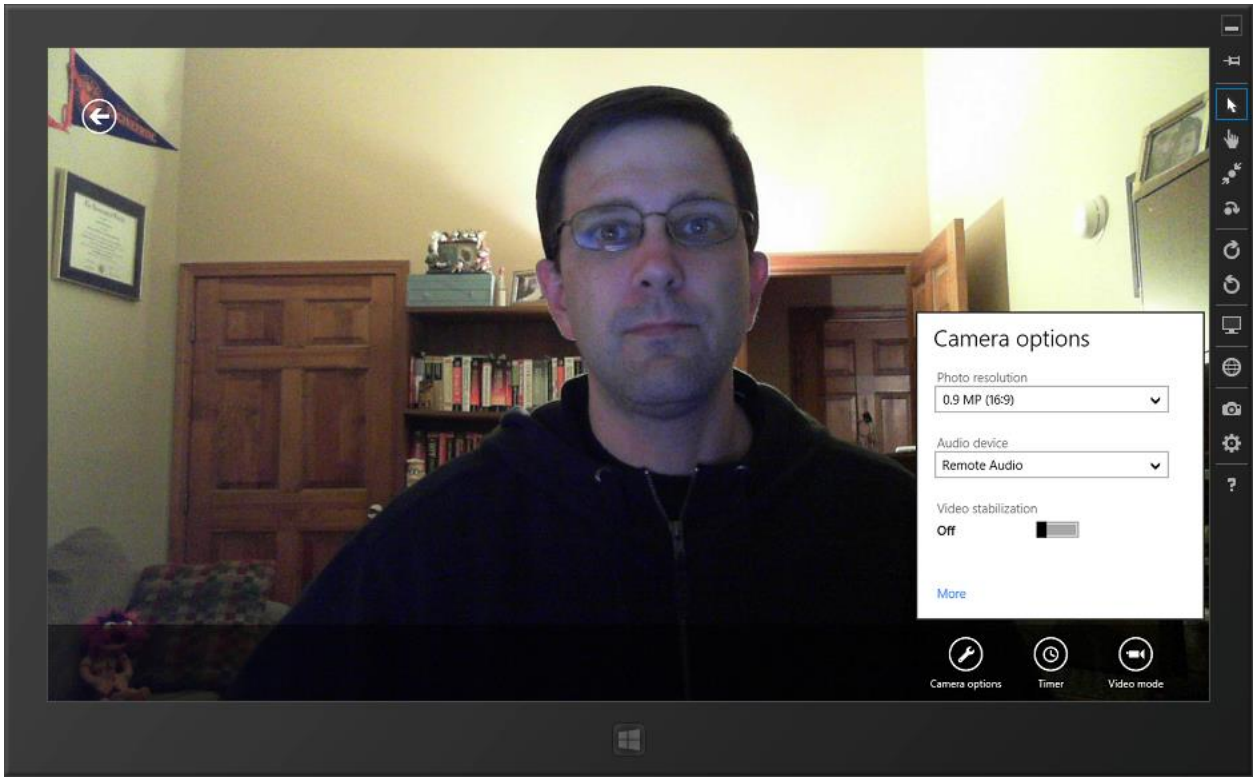


Figure 30: The CameraCaptureUI Image Capture Screen

The **`CameraCaptureUI`** user interface is invoked through a call to **`CaptureFileAsync`**, where a selection can be made as to whether the resulting UI will support photo, video, or user selection of either mode. If users select content to be returned through the UI, the content is returned as a **`StorageFile`**, which can then be stored on disk, potentially with some user interaction such as with a **`FileSavePicker`**. This sequence is shown in the following code:

```
// Create and invoke the CameraCaptureUI.
var cameraCaptureUI = new Windows.Media.Capture.CameraCaptureUI();
var capture = await cameraCaptureUI
    .CaptureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.PhotoOrVideo);

// Process the returned content.
if (capture != null)
{
    // Figure out the start folder based on whether image or video content was returned.
    var saveStartLocation =
        capture.ContentType.StartsWith("image", StringComparison.OrdinalIgnoreCase) ?
            Windows.Storage.Pickers.PickerLocationId.PicturesLibrary :
            Windows.Storage.Pickers.PickerLocationId.VideosLibrary;

    // Display a FileSavePicker to allow users to save the file.
    var savePicker = new Windows.Storage.Pickers.FileSavePicker();
    savePicker.FileTypeChoices.Add(capture.DisplayType,
                                       new List<String> { capture.FileType });
    savePicker.SuggestedFileName = capture.Name;
    savePicker.SuggestedStartLocation = saveStartLocation;
    var saveFile = await savePicker.PickSaveFileAsync();
    if (saveFile != null)
    {
        // Copy in the file.
        await capture.CopyAndReplaceAsync(saveFile);
    }
}
```

The **CameraCaptureUI** display takes care of handling when there is no video hardware connected or users have disabled video access by presenting the appropriate text on the screen.

Obtaining Finer Control over Multimedia Capture

Whereas the **CameraCaptureUI** offers a lot of functionality, there may be scenarios where Windows Store apps can benefit from more direct control of the media capture process. To achieve this, the **DeviceInformation** and **MediaCapture** classes can be used.

The **DeviceInformation** class provides the ability to enumerate hardware devices that meet specific criteria like audio or video capture devices. These values can be used to give users the ability to select which audio and video devices to use.

```
// Locate the available video capture devices.
var videoDeviceClass = Windows.Devices.Enumeration.DeviceClass.VideoCapture;
var videoDevices =
    await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync(videoDeviceClass);

// Locate the available audio capture devices.
var audioDeviceClass = Windows.Devices.Enumeration.DeviceClass.VideoCapture;
var audioDevices =
    await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync(audioDeviceClass);
```

Media interaction is coordinated through the **MediaCapture** class. This class allows several configuration values, including:

- Specifying the capture mode to be audio, video, or both. Note that the **CameraCaptureUI** does not allow any mechanism for only capturing audio.
- Setting the audio and video device **Id**s to be used. These **Id** values are obtained from the **DeviceInformation** query that was previously discussed.

The Media Capture Manager is first initialized with these setting values. It is then possible to set the **MediaCapture** instance to be the **Source** item for a **CaptureElement** control, which will allow the media content to be displayed in the application UI. The **StartPreviewAsync** method can be used to start the flow of data through the multimedia stream.

```
// Create and initialize the Media Capture Manager.
var captureMode = Windows.Media.Capture.StreamingCaptureMode.AudioAndVideo;
var settings = new Windows.Media.Capture.MediaCaptureInitializationSettings
{
    StreamingCaptureMode = captureMode,
    VideoDeviceId = selectedVideoDevice.Id,
    AudioDeviceId = selectedAudioDevice.Id,
};
var captureManager = new Windows.Media.Capture.MediaCapture();
await captureManager.InitializeAsync(settings);

// Connect the UI CaptureElement control to the Media Capture Manager.
// Note that this control is defined in XAML as
// <CaptureElement x:Name="VideoCapturePreviewRegion"/>.
VideoCapturePreviewRegion.Source = _captureManager;

// Start data flowing through the stream.
await captureManager.StartPreviewAsync();
```

To capture a picture from the **MediaCapture** class, the **CapturePhotoToStorageFileAsync** and **CapturePhotoToStreamAnsyc** methods are provided. An **ImageEncodingProperties** provides the ability to configure the type of image that will be emitted, including the width, height, and image type.

```
// Configure a File Save Picker and get the file to save into.
var savePicker = new Windows.Storage.Pickers.FileSavePicker
{
    SuggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.PicturesLibrary,
    SuggestedFileName = "Webcam Picture"
};
savePicker.FileTypeChoices.Add("PNG File (PNG)", new List<String> { ".png" });

var saveFile = await savePicker.PickSaveFileAsync();
if (saveFile != null)
{
    // Set the properties for the picture to capture.
    var pictureProperties = new Windows.Media.MediaProperties.ImageEncodingProperties
    {
        Width = 320,
        Height = 240,
```

```
        Subtype = "PNG",
    };

    // Write the picture into the file.
    await captureManager.CapturePhotoToStorageFileAsync(pictureProperties, saveFile);
}
```

Capturing audio and video is similar, except that an encoding profile is used to define the desired output content, and rather than capture being a one-shot operation as it is when working with pictures, the process needs to be started and then stopped when complete. There are several different kinds of audio and video output profiles that can be created, including M4A, MP3, MP4, WMA, and WMV files. These encoding profiles can be built programmatically or by supplying existing media files or streams whose settings will be used to construct a matching profile.

```
// Set the media encoding profile properties.
var desiredEncodingQuality = Windows.Media.MediaProperties.VideoEncodingQuality.Ntsc;
var videoEncodingProfile =
  Windows.Media.MediaProperties.MediaEncodingProfile.CreateWmv(desiredEncodingQuality);

// Start capturing into the selected file.
await captureManager.StartRecordToStorageFileAsync(videoEncodingProfile, saveFile);

// Wait for capturing to be stopped by the user.
// ...do other tasks...

// Stop the capture.
await captureManager.StopRecordAsync();
```

It is also possible to show a video settings user interface to allow adjusting properties of the multimedia data by calling the static **`CameraOptionsUI.Show`** method and passing it a copy of the current **`MediaCapture`** instance.

There are several other properties defined for the **`MediaCapture`** class, including the ability to interact with the multimedia pipeline by adding effects and setting more specific audio and video device properties. More information about these advanced various facilities provided by the **`MediaCapture`** class can be found in the MSDN documentation at http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.mediacapture.aspx.

Recap

This chapter discussed several of the ways that Windows Store apps can interact with system hardware. This included retrieving data about the system's environment through sensors, obtaining the device's location, and using the device's camera and microphone to obtain static images, audio, and video. This treatment was certainly not all-inclusive, as there are several other hardware interactions that are supported for Windows Store apps, including ways to retrieve detailed information about touch, manipulation, and gestures, the ability to interact with SMS messaging hardware, and access to proximity (NFC) sensors, just to name a few.

The next chapter will conclude this discussion of developing Windows Store apps by examining what options are available and what requirements need to be met in order to successfully deploy Windows Store apps.

Chapter 7 Deployment

Windows desktop applications have traditionally been deployed to end users with a combination of dedicated installer programs, file-copy mechanisms, and command-line batch files. For Windows Store apps, the options and mechanisms available for distributing applications are a little different and may be new to traditional desktop application developers.

As the name implies, the most common way users will obtain Windows Store apps is through the Windows Store itself. Developers can publish either free apps or apps that are sold for a fixed fee, and apps that are for sale can opt to include support for preview or trial modes. In addition to the revenue that can be realized through an app's initial purchase, developers can also make money throughout the lifetime of the app through both in-app purchases as well by integrating with one of several advertising frameworks. As has been discussed throughout the book, any apps submitted to the Windows Store must first go through a standard approval process before they are made available to the public, ensuring that the apps in the store are not malicious, comply with Microsoft's published requirements, and offer end users all of the necessary privacy and system-state protections.

Developers aren't restricted to only deploying through the public Windows Store, which is particularly interesting and valuable for line-of-business applications. There are mechanisms available to deploy Windows Store applications directly to end-user devices focused on several scenarios. However, these mechanisms do involve more than just simple file copying, as will be discussed later in this chapter.

Note: Visual Studio Express 2012 for Windows 8 includes a convenient Store menu with commands for several common actions related to working with Windows Store accounts and application deployments, whereas the higher-level Visual Studio editions do not include this menu. Some of the commands available in this menu can also be found elsewhere in all of the Visual Studio editions that support creating Windows Store apps, including a Store entry in the context menu that appears when right-clicking on a project in the Solution Explorer. This context menu includes commands for associating an app with a store entry, capturing app screenshots, and creating app packages for upload.

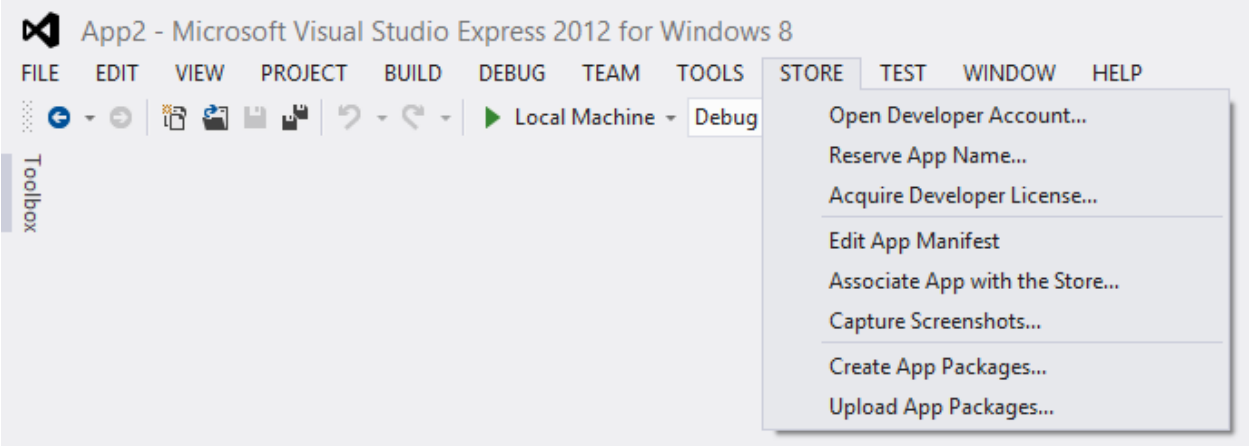


Figure 31: The Store Menu in Visual Studio Express 2012 for Windows 8

The Windows Store

The Windows Store offers several opportunities for developers. Apps published through the Windows Store are visible in a single central location and can reach more than 200 markets across the globe in over 100 different languages. The store itself implements the Search contract that was discussed previously, allowing users to find their way to apps of interest both via the store layout and through searching for key terms. Furthermore, descriptions of apps published in the Windows Store can be viewed over the web, and the content provided to describe the apps is exposed to search engines, making it easier for end users to discover apps they may be interested in.

As mentioned previously, there are several ways that Windows Store apps can be sold. The apps themselves can be assigned a wide array of prices within a predefined set of tiers. These include (prices in US dollars):

- Free
- $.50 intervals from $1.49 through $4.99.
- $1.00 intervals from $4.99 through $49.99.
- $5.00 intervals from $49.99 through $99.99.
- $10.00 intervals from $99.99 through $299.99.
- $50.00 intervals from $299.99 through $999.99.

For apps sold through the Windows Store, Microsoft's current model is that 70% of the revenue is passed along to the app owner until the app achieves $25,000 in revenue, after which the share percentage jumps to 80%. Furthermore, as has been mentioned and will be explored more thoroughly throughout this chapter, apps can include trial modes, built-in purchases through either the Windows Store or other means, and advertising.

The process for submitting an app to the Windows Store is not overly complex and typically involves:

1. Obtaining a Windows Store developer account.
2. Reserving an app name and providing some app settings.
3. Building, testing, and uploading the app.
4. Providing publishing information about the app for end users.
5. Submitting the app to the store for certification.

These steps will be discussed in the following sections.

Windows Store Developer Accounts

Before starting the process of submitting apps to the Windows Store, it is necessary to register for and obtain a Windows Store developer account that must be associated with a particular Microsoft Account, which is the name now used to describe what were formerly Hotmail or Windows Live accounts. There are two Windows Store developer account types: individual accounts and company accounts. Company accounts require additional verification to ensure that the applicant is authorized to create an account on behalf of the company, and in the US require an Employee Identification Number, also known as a Federal Tax Identification Number issued by the Internal Revenue Service. Individual accounts in the U.S. require a Social Security Number. Only apps submitted through company accounts can include the Enterprise Authentication, Shared User Certificate, and Documents Library Access capabilities. Because of these additional capabilities, apps submitted through company accounts may sometimes go through a more rigorous certification review than those submitted by individual accounts.

Regardless of the specific account type, the process for creating a Windows Store developer account can be started through the **Open Developer Account** command in the **Store** menu in Visual Studio Express, or by navigating to https://appdev.microsoft.com/StorePortals/Account/Signup/Start.

Note: Company accounts can also be used to submit desktop applications through the Windows Store. Publishing and obtaining desktop applications through the Windows Store is beyond the scope of this book. More information about deploying desktop apps through the Windows Store can be found at http://blogs.msdn.com/b/windowsstore/archive/2012/06/08/listing-your-desktop-app-in-the-store.aspx.

As of this writing, the standard fee charged for an individual developer account is $49 per year, and $99 per year for a company account. A table describing fees for other countries is available at http://msdn.microsoft.com/en-us/library/windows/apps/hh694064.aspx.

Tip: There are programs that periodically include offers that can help cover some or all of the cost of a Windows Store developer account subscription. Such programs potentially

include MSDN subscriptions and membership in Microsoft's DreamSpark and BizSpark programs. It is advisable to check with the program benefits resources and coordinators to understand what offers may be available.

Registering and Submitting an App

Once a Windows Store developer account is created, an app can be submitted to the Windows Store. Apps are submitted and managed through the Windows Store Dashboard, which is available online at https://appdev.microsoft.com/StorePortals. The Dashboard provides a list of existing app registrations and links for managing and viewing account information.

Reserving an App Name and Pre-Upload Settings

From within the Dashboard, the app submission process is broken down into several discrete steps that focus on providing different app information and content. The first step in this process is to create and reserve a name for the app, which must be completed before any of the other steps can be started. The app's name must be unique across all of the apps in the store, and while an app can have language-specific names, each of the names must be unique across all of the languages in the store. Furthermore, the app name that is selected in the Windows Store registration process must be used as the `DisplayName` value in the application's manifest file. An actual app must be submitted for a reserved name within a year of the reservation, or the reservation will be lost. An app name can be reserved by either selecting the **Reserve App Name** command from the **Store** menu in Visual Studio Express, or selecting the **Submit an App** link from the Dashboard. In either case, a webpage will be displayed that lists the steps necessary to submit an app—selecting the link on the **App Name** step will navigate the browser to a page where the app name can be provided.

Once an app name has been selected, Visual Studio can be used to associate an app with the store. Selecting the **Associate App with the Store** command from the **Store** menu in Visual Studio Express (or from the **Store** entry in the **Project** context menu in other Visual Studio editions) will bring up the **Associate Your App with the Windows Store** wizard. This wizard will ask for the Microsoft account credentials associated with a Windows Store developer account, and list the application names registered with that account. Selecting one of the applications and clicking **Associate** will update the `Package` entries in the application's manifest file:

- The package name will be updated to match the unique name being used in the Windows Store.
- The digital certificate file that is used to sign the application during development will be updated with published information that corresponds to the Windows Store developer account. Note that this is not the final certificate that will be used to sign the application when it is available in the Windows Store.
- This combination of updates will create the **Package family name** entry that uniquely identifies the application that is consistent with the one that will be made available when the application is added to the Windows Store.

Beyond the ability to associate an app with the store, once an app name is selected, the other pre-upload application registration steps can be completed. These steps include:

- **Selling details**: This is where a price is selected for an app or an indication is made if the app is to be available free of charge. If a price is selected for the app, the app can be configured to also offer a trial mode which will allow the app to be downloaded for free and potentially later upgraded. Trial modes can be set to either never expire or to expire after a particular duration of time. This screen also offers the ability to indicate the countries to which the app should be made available, the earliest app release date if one is desired, and the category and subcategory in which the app should appear within the Windows Store. Finally, some minimum system DirectX and RAM requirements may optionally be specified, as well as whether the app is designed to meet accessibility guidelines. More information about application accessibility can be found at http://www.microsoft.com/enable/.
- **Advanced features**: This is where settings related to providing push notifications may be obtained. Settings related to integrating with authentication provided by the Live Connect API can be both set and obtained here. Furthermore, this step includes the ability to register product IDs for in-app offers to be sold through the Windows Store. In-app purchases will be discussed later in this chapter.
- **Age rating and rating certificates**: This is where an app's age rating can be identified. The page includes descriptions and guidelines for each rating category. Note that unless an app is specifically geared for kids, in most cases 12+ will be the age category to use. In addition to selecting a target age group, certain countries have mandatory and optional requirements for games and other apps to include certificates issued by certain ratings boards, which can be uploaded through this page.
- **Cryptography**: Apps must identify whether they use any kind of cryptography due to regulations concerning the export of technology that makes use of certain kinds of encryption. For certain uses of encryption within an app, an Export Commodity Classification Number (ECCN) must be provided.

Uploading an App Package

Once the pre-upload information has been completed for an app, an app package can be created and uploaded. Selecting **Create App Packages** from the **Store** menu or context menu will invoke the Create App Packages wizard which can be used to create the **appxupload** packages to be uploaded to the store. Upon completion of the creation of the package file, the final screen of this wizard will include a link to the folder where the file has been created, as well as the option to run the app through the Windows Application Certification Kit.

Windows Application Certification Kit

The tools installed for developing Windows Store apps include the Windows Application Certification Kit (ACK). Windows ACK is a tool that automates the process of testing several characteristics of Windows Store applications (among other application types). The tool will run the application multiple times while checking for functional and security issues. When the tool finishes, a report indicating any problems that must be addressed for the app to pass certification will be shown.

More information about using Windows ACK can be found at http://msdn.microsoft.com/en-us/library/windows/apps/hh694081.aspx. Microsoft has also published a white paper discussing the Windows ACK that can be downloaded from http://www.microsoft.com/en-us/download/details.aspx?id=27414.

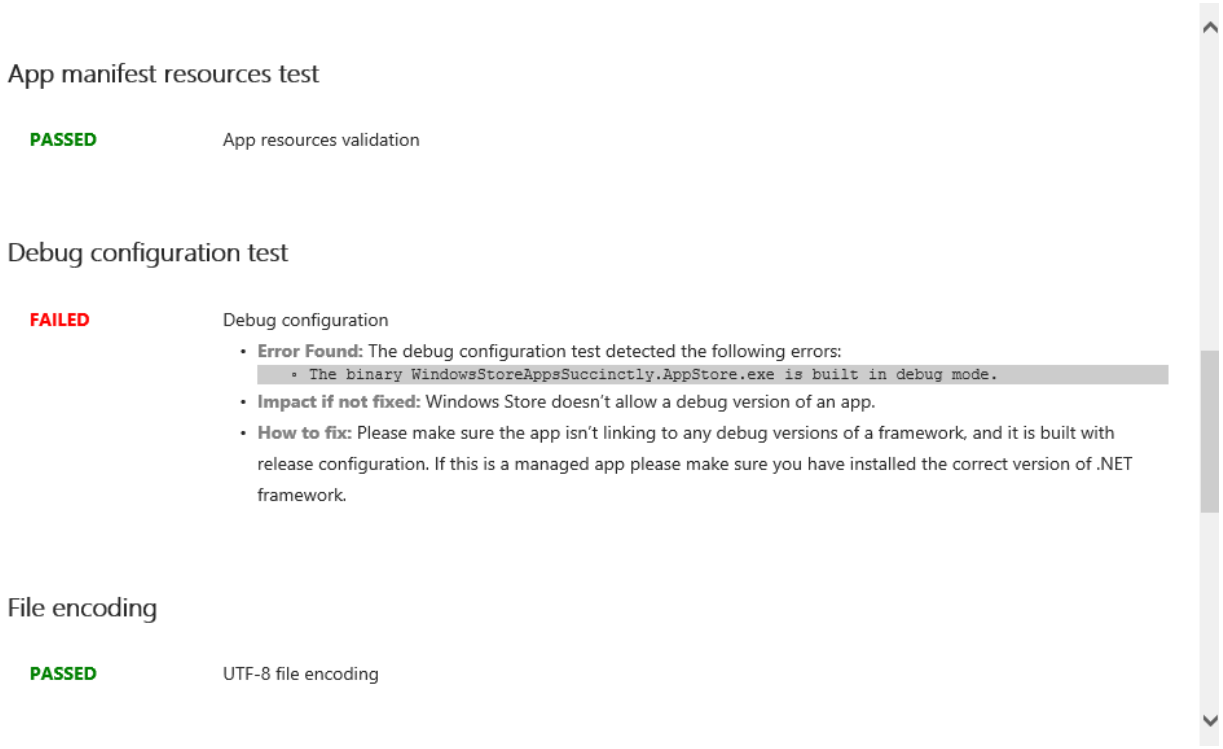


Figure 32: A Certification Issue Identified in a Windows ACK Report

Post-Upload Content

Once the app package has been successfully uploaded, the last two pieces of information to be provided include the information that will be displayed to customers in the app's Windows Store entry, and any notes that may help certification testers successfully navigate the app (e.g., test account usernames and passwords, descriptions of how to navigate to features whose access may not be obvious, etc.).

The application description information that can be provided includes:

- The app's description text (mandatory).
- A listing of the app's features (optional).
- Keywords that can be matched to help end users searching for an app (optional).
- Information about update contents (optional).
- The copyright and trademark information to display to the end user (mandatory).
- A description of any additional license terms related to the use of the application (optional).

- Application screenshots and related captions (at least one is mandatory). Note that the simulator discussed in previous chapters provides built-in facilities for taking app screenshots.
- Promotional images to be used if the app is selected to be featured in the Store (optional).
- An indication of any hardware recommendations (optional).
- A URL to a website related to the app (optional).
- An email address or URL that users can use to obtain app support (mandatory).
- A URL link to the app's privacy policy text (mandatory if the app collects end users' personal information or connects to online services).
- If the app includes in-app purchases through the Windows Store, a brief description of each in-app offer (mandatory if any offer entries have been provided).

The Certification Process

As each information gathering step is completed, its displayed status within the Dashboard will switch from **Incomplete** to **Complete**. Once all of the steps have been completed, the app can officially be submitted for certification and, if it passes the certification process, inclusion in the Windows Store.

The certification process is meant to be fairly transparent, and includes several steps:

- Pre-processing inspection: Ensures that the appropriate details needed to publish the app are in place, including ensuring that the developer account is in good order and that payout information has been set up correctly.
- Security tests: The app is examined for viruses and other malicious code.
- Technical compliance tests: The Windows Application Certification Kit is used to automatically test the app.
- Content compliance testing: The app's content is examined by a human tester.
- Release date: If the app has indicated a particular release date is desired, this step will hide the app until the target release date arrives.
- Signing and publishing: The app is digitally signed to ensure it is not tampered with once approved and released, and then is published to the Windows Store.

In order to help provide transparency into the process, the Dashboard will indicate an app's progress through each of these certification steps, along with information about how long each of these steps typically takes. Furthermore, whether or not an app fails certification, a certification report will be provided. The report can be used in case of failures to determine what went wrong, provide corrections, and resubmit the app.

The Windows Store app certification requirements revolve around a key set of tenets. In order to pass certification, Windows Store apps:

- Must provide value to the customer.
- May display ads, but must be more than just ads or websites-within-an-app.

- Must behave predictably.
- Must put the customer in control.
- Must be appropriate for a global audience.
- Must be easily identified and understood.

The specific guidance for how these criteria are to be met is communicated, but is expected to evolve. The latest version of these requirements along with revision history information can be found at http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh694083.aspx.

Including Trial Modes

Information gathered from Microsoft's experiences with Windows Phone app sales provides interesting insights into the use of trial modes in apps.[1] At the time it was published, its analysis showed:

- Paid apps that offered trial modes were 70 times more likely to be downloaded than those that did not.
- The rate of conversion from a trial app to a paid app was nearly 10%.
- Apps that offered trial modes generated 10 times more revenue on average than paid apps that did not include trial modes.

As mentioned in the application pre-upload setting step, paid apps offered through the Windows Store can be configured to offer trial modes. WinRT APIs can be used within an app to determine if it is currently running in a trial mode, when the trial mode is set to expire, or if a trial mode has already expired.

Note: There is no functionality built into the trial mode APIs that directly affects the app's runtime behavior; the APIs merely provide information about the current state of the app's license. It is up to the app developer to determine how to interpret this information and adjust functionality accordingly, such as by disabling features or reducing functionality, providing messages to the end user, etc.

The static **`CurrentApp`** class within the **`Windows.ApplicationModel.Store`** class is used to obtain information about an app's current trial mode state. Among other functionality, this class offers members that can be used to retrieve information about an app's listing in the Windows Store, the current state of the installed app's license, and to request that a trial mode app be upgraded. The following code shows how the information related to trial mode for an application can be obtained.

```
// Obtain the current application's licensing state.
var licenseInformation = Windows.ApplicationModel.Store.CurrentApp.LicenseInformation;
```

[1] Source: http://windowsteamblog.com/windows_phone/b/wpdev/archive/2011/03/08/an-update-on-windows-phone-marketplace-new-tips-policies-and-regional-access-program.aspx

```
// The application is currently in trial mode (though it may have expired).
var isTrial = licenseInformation.IsTrial;

// The application is in full mode, or is in trial mode and not yet expired.
var isActive = licenseInformation.IsActive;

// The date the application's trial mode will expire (relative to the system clock).
// DateTimeOffset.MaxValue for non-trial mode or non-expiring trials.
// DateTimeOffset.MinValue for expired trials.
var expirationDate = licenseInformation.ExpirationDate;

// Event raised when license information changes as the result of an upgrade request.
// Note that an app crossing its trial expiration time will not raise this event.
licenseInformation.LicenseChanged += HandleLicenseInformationLicenseChanged;
```

The final aspect of interacting with trial mode apps is to provide the ability for a trial mode app to be upgraded to full functionality. This is provided by the **RequestAppPurchaseAsync** method on the **CurrentApp** class. Since this method needs to interact with the Windows Store, calling it requires the app to have access to an available Internet connection.

```
// Request an app upgrade purchase.
// The parameter indicates whether the method should return an XML receipt.
await Windows.ApplicationModel.Store.CurrentApp.RequestAppPurchaseAsync(false);
```

While there are circumstances that can result in this method throwing an exception, cancellation or other circumstances where the purchase is not completed can result in the method returning without error. As a result, it is important to either listen to the **LicenseChanged** event, or retrieve and inspect **LicenseInformation** at the conclusion of a call to **RequestPurchaseAsync**.

Debugging Trial Mode Applications

In order to develop, test, and debug applications that support trial modes, a mechanism is necessary to simulate the calls that **Windows.ApplicationModel.Store.CurrentApp** would otherwise make to the Windows Store. To help with this, the WinRT APIs include the **Windows.ApplicationModel.Store.CurrentAppSimulator** class. This class offers the same methods and properties as the **CurrentApp** class, plus one additional call. Because of this similarity, the **CurrentAppSimulator** class can be substituted for the **CurrentApp** class during development to validate the implementation of an app's trial mode functionality.

In the most common use cases, the **CurrentAppSimulator** works by loading app information from a proxy XML file titled **WindowsStoreProxy.xml** that it expects to find inside the **Microsoft\Windows Store\ApiData** folder within the app's local storage folder. The schema describing the values contained within the proxy file can be found in the MSDN documentation at http://msdn.microsoft.com/library/windows/apps/windows.applicationmodel.store.currentappsimulator.aspx.

The proxy file contains information describing basic app store information, including a collection of products offered for in-app purchase (this will be discussed in more detail later). It also includes the ability to configure the current state of the app's license information that describes the trial status of an application and any purchased products, as well as settings that can be used to either specify HRESULT error codes that should be returned by **CurrentAppSimulator** method calls or instructions to bring up a dialog that allows those error codes to be set interactively.

It is important to note that updates made while an app is running occur in-memory and are not saved in the WindowsStoreProxy file, but rather exist only in memory. Relaunching the app will result in the file being read from disk again and any licensing changes that were previously made will have been lost.

Finally, any **CurrentAppSimulator** calls must be removed prior to submitting an app to the Windows Store. If an app includes simulator calls, it will fail certification. The following code makes use of the **#if** preprocessor directive to create a **CurrentAppAccesor** alias that allows swapping out the **CurrentAppSimulator** class for the **CurrentApp** class when the app is built in release mode:

```
// This #if approach allows swapping out to the "real" app when deploying to the store.
#if DEBUG
    using CurrentAppAccessor = Windows.ApplicationModel.Store.CurrentAppSimulator;
#else
    using CurrentAppAccessor = Windows.ApplicationModel.Store.CurrentApp;
#endif
```

In-App Purchases

In-app purchases allow end users to enhance an app by making purchases for discrete units of functionality above and beyond what is included in the basic application itself, such as additional game levels or perhaps seasonal content for a kids' coloring app. Both free and paid applications can include in-app purchases, though it is important to note that an app that is not useful to end users without any in-app purchases will most likely fail certification. In-app purchases need to enhance application functionality in some way. While the terms for an app being included in the Windows Store do not restrict the mechanism used to safely and securely offer such purchases, the Windows Store does include a mechanism to conveniently integrate such functionality.

The Windows Store mechanism for in-app purchases uses the term "product" to indicate a discrete piece of functionality that is offered for in-app purchasing. Products are defined in the **Advanced Features** section of the application registration process, and must include:

- A product ID, which is an internal code used to identify the feature.
- A price, which can be any of the pricing values discussed previously for app pricing.
- A lifetime indicating how long the feature is active following the user's purchase (includes **Forever** as an option).

A user-readable description of each product must also be provided, but this occurs later in the registration process when the application description information is being configured.

As with trial mode support, the in-app purchase information defined in the Windows Store registration process is paired with code designed to interact with it within the application.

The first task is to obtain the list of available in-app purchase items to be presented to users in whatever way is appropriate to the application. The list of defined packages can be obtained through the **CurrentApp** (and **CurrentAppSimulator**) object. The **LoadListingInformationAsync** method obtains information related to the app's registration with the Windows Store, including the app's name, description, age rating, current market, price, and product listings. The product listings include the **Name**, **ProductId**, and **Price** information for each registered package, which can then be shown to users.

The **CurrentApp** class includes a **RequestProductPurchaseAsync** method that functions much like the previously discussed **RequestAppPurchaseAsync** method, except that it allows the ID of the feature to be purchased to be specified. If the purchase is successful, the **LicenseChanged** event will be raised, and the **CurrentApp LicenseInformation** property will be updated.

```
// Obtain the app's Windows Store listing information.
var listing = await
              Windows.ApplicationModel.Store.CurrentApp.LoadListingInformationAsync();
var productListings = listing.ProductListings;

// Purchase a specific feature from the store.
await Windows.ApplicationModel.Store.CurrentApp.RequestProductPurchaseAsync(featureId,
                                                                            false);

// Check the existing licenses to obtain activity and expiration information.
var licenses = licenseInformation.ProductLicenses;
var featureLicense = licenses[featureId];
var featureIsActive = featureLicense.IsActive;
var featureExpirationDate = featureLicense.ExpirationDate;
```

When using the **CurrentAppSimulator** class, the XML includes the ability to define products within the application listing markup, as well as existing product purchases within the license information markup.

Adding Ads

Another means of realizing revenue from Windows Store apps is through the use of embedded ads. Like in-app purchases, the terms for an app being included in the Windows Store do not impose restrictions as to which specific ad display tool must be used, but the Microsoft Advertising SDK for Windows 8 provides a convenient mechanism for doing so that integrates with Microsoft's pubCenter advertising platform. Regardless of the tool chosen, the basic certification requirements must always be met, especially those related to user privacy concerns.

Note: While this section will provide basic information about using the Microsoft Advertising SDK for Windows 8, additional and more in-depth information can be found at http://msdn.microsoft.com/en-us/library/hh506371.aspx.

Configuring pubCenter Content

To include Advertising SDK ads in a Windows 8 application, application and related "ad unit" information needs to be provided in pubCenter which can be accessed at https://pubcenter.microsoft.com. To access pubCenter, an account must be set up which will be used to manage application registrations as well as to configure payment information.

From a pubCenter account, the first step in preparing to provide advertising is to register an application. Registering an application merely requires identifying that it is a Windows 8 application and providing a name for the application so that pubCenter can generate a unique application ID. For each application, multiple ad units can be created. Ad units are uniquely associated with an application. Creating an ad unit requires defining a size and a category for the item to be presented to users. There are several ad sizes currently available for Windows Store apps; selecting the right one is dependent on the layout of the application and the space in which the ad is intended to fit. Ads should be placed where they can be seen by end users, but preferably in a way that does not interfere with users' ability to successfully use the app.

Ad units also require an ad category to be selected. The category (and potentially subcategory) value helps determine what ads will be shown in an effort to ensure that presented ads are relevant to the audience using the app. This is especially important because of the revenue model for advertisements. Ad revenue is realized based on a combination of impressions (the number of ads served to individual users' screens) and click-throughs (the number of times an ad is clicked). Ads that are relevant to the interests of the end user are more likely to be clicked, potentially increasing revenue.

The final configuration option for pubCenter is the ability to establish ad exclusions. Ad exclusions are used to grant the ability to exclude ads that link to certain URLs. The most obvious use for these is to prevent ads that link to competitors' websites from being displayed within an application.

It is important to take note of both the Application ID and Demo Unit IDs that are created. They will be used when the advertising control is included in the Windows Store app.

Using the Advertising SDK

Before the ad control can be included in a Windows Store app, the Microsoft Advertising SDK for Windows 8 must first be downloaded and installed. Instructions for obtaining the SDK can be found at http://msdn.microsoft.com/en-us/library/hh506342.aspx. Also, including the ad control in a Windows 8 app requires the Internet (client) capability to be defined in the app's manifest file.

Note: This requirement has the fortunate side effect of implying that this SDK cannot be used to place ads in apps that are rated for young children since apps rated below 12+ cannot include access to online services or else they will fail the certification process.

The ad control can be included on a page manually or by dragging it from the toolbox in Visual Studio. Dragging it will add the appropriate project reference and XAML namespace declaration.

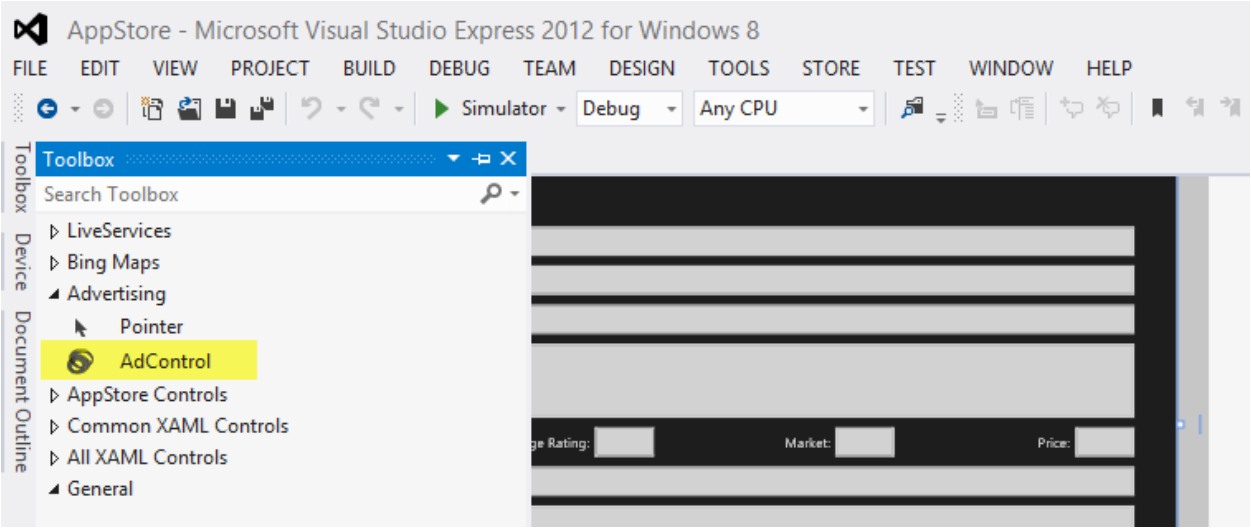


Figure 33: The Ad Control in the Visual Studio XAML Designer Toolbox

To manually include the control, the project will need a reference added to the **Microsoft Advertising SDK for Windows 8 (Xaml)** assembly. The page or control on which the ad control is being placed will need to include a namespace declaration for **`Microsoft.Advertising.WinRT.UI`**. The ad control can then be placed in the appropriate location and sized according to the dimensions of the ad unit being displayed.

```
Namespace declaration for the ad control:

xmlns:UI="using:Microsoft.Advertising.WinRT.UI"

Ad control element declared in XAML:

<UI:AdControl x:Name="AdControl" Grid.ColumnSpan="99" Grid.Row="1"
              ApplicationId="d25517cb-12d4-4699-8bdc-52040c712cab"
              AdUnitId="10042999"
              Width="728" Height="90"
              HorizontalAlignment="Center"
              Margin="0,10,0,0"/>
```

Whether added manually or via drag-and-drop from the toolbox, the ad control **`ApplicationId`** and **`AdUnitId`** values need to be set based on the values noted previously in pubCenter.

Tip: During development and testing, the emulator is not able to show real, live ads. However, test values for ApplicationId and AdUnitId are available that allow the ad control to simulate it is displaying actual ads. The available values that can be used are defined at http://msdn.microsoft.com/en-us/library/advertising-windows-test-mode-values(v=msads.10).aspx.

Note: When the Microsoft Advertising SDK for Windows 8 is installed, it includes a EULA document that contains provisions that may need to be included in the privacy policy, terms of use for the Windows Store app, or both. It is important to review this content prior to distributing an app that includes the ad control and ensure the necessary conditions are being met. Information pertaining to the EULA can be found at http://msdn.microsoft.com/en-us/library/jj157026.aspx.

There are several other settings available for the ad control—one such setting is the ability to provide latitude and longitude values to the ad control to ensure that the ads shown are contextual to the geographic region indicated by the coordinates. However, providing this piece of additional information will require the appropriate capability to be declared, users made aware that the information is being shared, and controls to allow users to opt out of providing such information.

Other Ways to Distribute Windows Store Apps

The primary focus of this chapter has related to distributing apps through the Windows Store. This is certainly the most common way these apps will be obtained by end users, and the only way to tap into the Windows Store mechanisms for trial modes and in-app purchases. However, there are several scenarios where this kind of deployment is not ideal, such as for an app outside of a development machine or for sharing line-of-business apps within an enterprise. For these and similar cases, there are some alternate deployment approaches available.

The first deployment option is through the use of developer licenses on target machines. This approach is most useful for situations where it is desirable to test an app on various non-developer machines. The app is secured with the self-signed certificate that is generated by Visual Studio, and is only able to run on machines that have a valid developer license available (discussed in Chapter 1). These developer licenses are only valid for a limited amount of time. The precise duration depends on whether they are obtained with a Microsoft account linked to a Windows Store account, though they can be renewed upon expiration. Apps running under the auspices of a developer account that has expired will appear with an X on their Start menu tile, indicating there is something wrong that prevents the app from running.

The easiest way to deploy an application using a developer license is to use the content emitted when Visual Studio creates an app package. The process for creating such a package was described earlier in this chapter, though if the intent is to create such a package without uploading it to the Windows Store, **No** can be selected in the question about building packages for upload in the first page of the Create App Packages wizard in order to skip the association steps.

A folder with a name of **<App Package Name>_<Version>_<Processor Architecture>_Test** will be added to the target directory where the package is created. This folder includes the content necessary to do a local app deployment, including a PowerShell script titled **Add-AppDevPackage.ps1**. This folder can be copied to the machine where the app is to be run, and the PowerShell script can be run to install the Windows Store app. The PowerShell script will also register the self-signed certificate and check to see if a developer license is available on that machine—if not, the user will be prompted to obtain one.

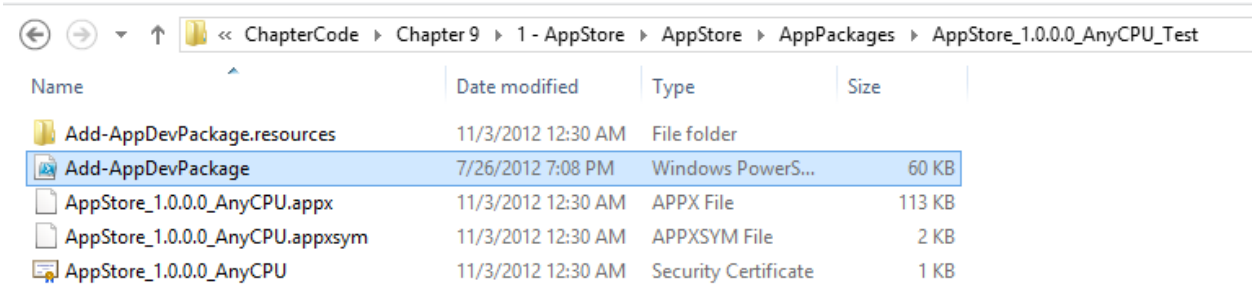


Figure 34: Contents of a Test Deployment Folder

The second deployment option is known as sideloading. In general terms, sideloading is often used to refer to the process of directly applying software to a device while bypassing a normal managed approach of doing so. However, for Windows Store apps, the term has been associated with one very specific mechanism for doing so. For an app to be able to be sideloaded on a device running Windows 8 or Windows RT, the following conditions must be met:

- The machine must be running Windows RT, Windows 8 Enterprise, Windows 8 Professional, or Windows Server 2012.
- If it is running Windows 8 Enterprise or Windows Sever 2012, the OS must be joined to an active directory domain that has the **Allow all trusted applications to install** group policy enabled.
- Alternatively, all Windows 8 Professional, Windows 8 Enterprise, or Windows Server 2012 systems that are not joined to an active directory domain may participate if they have a special sideloading product key applied and the **Allow all trusted applications to install** group policy setting has been enabled locally.
- Windows RT devices must have a special sideloading product key applied in order to participate.
- The application must be digitally signed with a certificate that is chained to a trusted root certificate on the target machine.

Note: The sideloading product keys are obtained through Microsoft's Volume License program. A specific discussion of the details of obtaining and applying these product keys is beyond the scope of this book, but additional information can be found in the Windows 8 and Windows RT Volume Licensing Guide available at http://download.microsoft.com/download/9/4/3/9439A928-A0D1-44C2-A099-26A59AE0543B/Windows_8_Licensing_Guide.pdf.

Sideloading is being positioned as a feature for enterprise operations to use to deploy applications in-house, and requires a certain amount of infrastructure and knowledge of Microsoft licensing and enterprise system management in order to be properly orchestrated. More specific guidance for the requirements described previously and how to manage sideloaded apps is available at http://technet.microsoft.com/en-us/library/hh852635.aspx

Recap

In this chapter, the basic concepts related to publishing an app to the Windows Store were presented, including the need to obtain a developer account and the process involved in submitting an app for certification. It also presented additional options available for realizing revenue from these apps, including offering trial modes, providing in-app purchases, and enabling ads in Windows Store apps. Finally, some additional options were discussed for how to deploy apps without involving the Windows Store.

This chapter concludes what has hopefully been a succinct yet useful discussion of many of the parts involved and available in the creation of Windows Store applications using XAML and C#. Certainly there is much more that can be covered, and hopefully the content that has been presented provides a solid foundation from which additional areas of this platform can be explored.