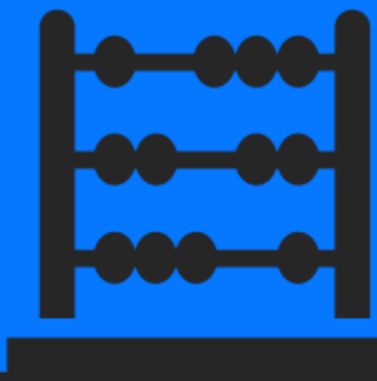

Windows Phone 8 Development Succinctly

by Matteo Pagani

Syncfusion | Technology Resource Portal

Windows Phone 8 Development Succinctly

By
Matteo Pagani

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Rajen Kishna
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, 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

Matteo Pagani is a developer with a strong passion for mobile development, particularly within the Windows Phone platform and Microsoft technologies.

He graduated in Computer Science in 2007 and became a web developer. In subsequent years he started to show great interest in mobile development, especially in the Windows Phone platform. He started to share his passion with the technical community by launching two blogs (in Italian and in English), where he regularly wrote articles and technical posts about his experience with the Windows Phone and Windows 8 platforms.

Pagani is a regular writer for many technical websites and wrote the first Italian book about Windows Phone 8 development, published by FAG Editore. A professional speaker, he has joined many communities and official conferences like WhyMCA, WPC, and Community Days, and is a member of the Italian community DotNetLombardia.

Since January 2011, Pagani has been a Microsoft MVP in the Windows Phone Development category and, since October 2012, he's been awarded as a Nokia Developer Champion.

He currently works in Funambol as a developer on the Windows team, where he focuses on Windows Phone and Windows 8 projects.

Chapter 1 Introduction

There's a new kid in town

Smartphones and tablets are, without any doubt, the kings of the consumer market. Traditional computers won't go away anytime soon, especially in the business world, but in many scenarios mobile devices have replaced the old "mouse and keyboard" approach with a more modern and intuitive one, based on touch and natural interfaces. For sure, the iPhone by Apple was the product that forced all the other companies to rethink the mobile experience: first, with a new phone concept, and second, with the idea of applications and app stores. These days, one of the key considerations when choosing a new phone is the availability and quality of the apps, rather than just the features offered by the platform. Developers play an important role in this.

Microsoft might have joined the party a little late, but it did so with a fresh and new approach. Microsoft was developing Windows Mobile 7 when it realized that the phone wouldn't be an appealing product for consumers who were starting to get used to iPhone or Android devices. So its developers dropped the project and started from scratch to build a totally new platform: Windows Phone 7. The result was really different from the other competitors: a new user interface, based on a flat design style called Microsoft Design style (once known as Metro); and deep integration with social networks and all the Microsoft services, like Office, SkyDrive, and Xbox.

The current version of the platform (which will be covered in this book) is Windows Phone 8; in the middle, Microsoft released an update called Windows Phone 7.5 that added many new consumer features but, most of all, improved the developer experience by adding many new APIs.

Windows Phone 8 is a fresh start for the platform: Microsoft has abandoned the old stack of technologies used in Windows Phone 7 (the Windows Mobile kernel, Silverlight, XNA) to embrace the new features introduced in Windows 8, like the new kernel, the Windows Runtime, and the native code (C++) support.

For this reason, Windows Phone 8 isn't available as an update for old Windows Phone 7 devices. To help users in the transition, Microsoft has released an intermediate update called Windows Phone 7.8, which has ported some of the new Windows Phone 8 features (like the new Tile formats) to the old devices.

Since the release of Windows Phone 8, Microsoft has released three updates:

- Update 1 (or GDR1), which added some improvements in Internet Explorer, Wi-Fi connectivity, and messaging experience.
- Update 2 (or GDR2), which improved support for Google accounts, Xbox Music, and Skype, added FM radio support, and expanded the availability of the Data Sense application to keep track of the data traffic.
- Update 3 (or GDR3), which added support for a new resolution (1080p), driving mode, screen lock orientation, and better storage management, and improved the Bluetooth and Wi-Fi stack.

The hardware

Talking about the hardware is important because it's strictly connected to the developement experience and the features you can use while developing an application. With Windows Phone, Microsoft has introduced an approach that is a hybrid between those of Apple's and Google's.

Like Android, Windows Phone can run on a wide range of devices, with different form factors and hardware capabilities. However, Microsoft has defined a set of hardware guidelines that all manufacturers need to follow to build a Windows Phone device. In addition, vendors can't customize the user interface or the operating system; all the phones, regardless of the producer, offer the same familiar user experience.

This way, Windows Phone can take the best from both worlds: a wide range of devices means more opportunities, because Windows Phone can run well on cheap and small devices in the same way it works well on high-resolution, powerful phones. A more controlled hardware, instead, makes the lives of developers much easier, because they can always count on features like sensors or GPS.

Here are the key features of a Windows Phone 8 device:

- Multi-core processor support (dual core and quad core processors).
- At least 512 MB of RAM (usually 1 GB or 2 GB on high-end devices).
- At least 4 GB of storage (that can be expanded with a Micro SD).
- Camera.
- Motion sensors (accelerometer, gyroscope, compass), optional.
- Proximity sensor, optional.
- Wi-Fi and 3G connection.
- GPS.
- Four supported resolutions: **WVGA** (480 × 800), **WXGA** (768 × 1280), **720p** (720 × 1280), and **1080p** (1080 × 1920).
- Three hardware buttons: Back, Start, and Search.

The Windows Runtime

The Windows Runtime is the new API layer that Microsoft introduced in Windows 8, and it's the foundation of a new and more modern approach to developing applications. In fact, unlike the .NET framework, it's a native layer, which means better performance. Plus, it supports a wide range of APIs that cover many of the new scenarios that have been introduced in recent years: geolocation, movement sensors, NFC, and much more. In the end, it's well suited for asynchronous and multi-threading scenarios that are one of the key requirements of mobile applications; the user interface needs to be always responsive, no matter which operation the application is performing.

Under the hood of the operating system, Microsoft has introduced the **Windows Phone Runtime**. Compared to the original Windows Runtime, it lacks some features (specifically, all the APIs that don't make much sense on a phone, like printing APIs), but it adds several new ones specific to the platform (like hub integration, contacts and appointments access, etc.).

Microsoft faced a challenge during the Windows Phone 8 development: there was already a great number of applications published on the Windows Phone Store that were based on the "old" technologies like Silverlight and XNA. To avoid forcing developers to write their apps from scratch, Microsoft introduced three features:

- The XAML stack has been ported directly from Windows Phone 7 instead of from Windows 8. This means that the XAML is still managed and not native, but it's completely aligned with the previous one so that, for example, features like behaviors, for which support has been added only in Windows 8.1, are still available). This way, you'll be able to reuse all the XAML written for Windows Phone 7 applications without having to change it or fix it.
- Thanks to a feature called **quirks mode**, applications written for Windows Phone 7 are able to run on Windows Phone 8 devices without having to apply any change in most cases. This mode is able to translate Windows Phone 7 API calls to the related Windows Runtime ones.
- The Windows Phone Runtime includes a layer called **.NET for Windows Phone**, which is the subset of APIs that were available in Windows Phone 7. Thanks to this layer, you'll be able to use the old APIs in a Windows Phone 8 application, even if they've been replaced by new APIs in the Windows Runtime. This way, you'll be able to migrate your old applications to the new platform without having to rewrite all the code.

Like the full Windows Runtime, Windows Phone 8 has added support for **C++** as a programming language, while the **WinJS layer**, which is a library that allows developers to create Windows Store apps using HTML and JavaScript, is missing. If you want to develop Windows Phone applications using web technologies, you'll have to rely on the `WebBrowser` control (which embeds a web view in the application) and make use of features provided by frameworks like PhoneGap.

This book will cover the development using C# as a programming language and XAML as a user interface language. We won't talk about C++ or VB.NET (the available APIs are the same, so it will be easy to reuse the knowledge acquired by reading this book).

Plus, since this book is about Windows Phone 8, I will cover just the Windows Runtime APIs. In the areas where APIs are duplicated (meaning that there are both Windows Runtime and .NET for Windows Phone APIs to accomplish the same task, like storage or sensors), I will cover only the Windows Runtime ones.

The development tools

The official platform to develop Windows Phone applications is **Visual Studio 2012**, although support has also been added to the Visual Studio 2013 commercial versions. The major difference is that while Visual Studio 2012 still allows you to open and create Windows Phone 7 projects, Visual Studio 2013 can only be used to develop Windows Phone 8 applications.

There are no differences between the two versions when we talk about Windows Phone development: since the SDK is the same, you'll get the same features in both environments, so we'll use Visual Studio 2012 as a reference for this book.

To start, you'll need to download the Windows Phone 8 SDK from the official developer portal at http://dev.windowsphone.com. This download is suitable for both new developers and Microsoft developers who already have a commercial version of Visual Studio 2012. If you don't already have Visual Studio installed, the setup will install the free Express version; otherwise, it will simply install the SDK and the emulator and add them to your existing Visual Studio installation.

The setup will also install **Blend for Windows Phone**, a tool made by Microsoft specifically for designers. It's a XAML visual editor that makes it easier to create a user interface for a Windows Phone application. If you're a developer, you'll probably spend most of the time manually writing XAML in the Visual Studio editor, but it can be a valid companion when it comes to more complex things like creating animations or managing the visual states of a control.

To install the Windows Phone 8 SDK you'll need a computer with **Windows 8 Pro** or **Windows 8 Enterprise 64-bit**. This is required since the emulator is based on **Hyper-V**, which is the Microsoft virtualization technology that is available only in professional versions of Windows. In addition, there's a hardware requirement: your CPU needs to support the Second Level Address Translation (**SLAT**), which is a CPU feature needed for Hyper-V to properly run. If you have a newer computer, you don't have to worry; basically all architectures from Intel i5 and further support it. Otherwise, you'll still be able to install and the use the SDK, but you'll need a real device for testing and debugging.

You can use a free tool called Machine SLAT Status Check to find out if your CPU satisfies the SLAT requirement.

The emulator

Testing and debugging a Windows Phone app on a device before submitting it to the Windows Phone Store is a requirement; only on a real phone will you be able to test the true performance of the application. During daily development, using the device can slow you down. This is when the emulator is useful, especially because you'll easily be able to test different conditions (like different resolutions, the loss of connectivity, etc.).

The emulator is a virtual machine powered by Hyper-V that is able to interact with the hardware of your computer. If you have a touch monitor, you can simulate the phone touch screen; if you have a microphone, you can simulate the phone microphone, etc. In addition, the emulator comes with a set of additional tools that are helpful for testing some scenarios that would require a physical device, like using the accelerometer or the GPS sensor.

You'll be able to launch the emulator directly from Visual Studio. There are different versions of the emulator to match the different resolutions and memory sizes available on the market.

The developer experience

Windows Phone applications are published on the Windows Phone Store, which is the primary way for developers to distribute their applications. However, there are two exceptions: enterprise companies and developers for testing purposes.

To start publishing applications, you'll need a developer account, which can be purchased from the official portal at http://dev.windowsphone.com. The fee is $19 per year and allows you to publish an unlimited number of paid apps and a maximum of 100 free apps. Recently, Microsoft has merged the developer experience for its two main platforms. This means that with the developer account, you'll also be able to publish Windows Store apps for Windows 8 on the Windows Store.

The developer account is also connected to testing. In fact, by default, phones are locked and the only way to install third-party apps is through the Windows Phone Store. All developers can unlock phones for free, even if they don't have a paid account; the difference is that with a free account, only one phone can be unlocked and only two applications can be loaded on the phone. With a paid account, developers are able to unlock up to three phones and can load up to 10 applications on each.

The app deployment can be performed directly from Visual Studio or by using a tool installed with the SDK called **Application Deployment**.

To unlock your phone, you'll need another tool that comes with the SDK called **Windows Phone Developer Registration**. You'll have to connect your phone to the computer and sign in with the same Microsoft account you've used to register the developer account.

The application to be published on the Windows Phone Store needs to be certified. The certification process (in which both automatic and manual tests are executed) makes sure that your app is acceptable from a technical (the app doesn't crash, the user experience isn't confusing, etc.) and content (no pornography, no excessive violence) point of view.

We'll cover more details about the publishing process in the last chapter of the book.

Preview for developers

Microsoft has introduced a new program for developers to grant early access to new Windows Phone updates. This way, developers are able to test their apps against the latest OS releases before they are distributed to users.

To subscribe to the program you have to:

- Own a developer unlocked phone.
- Have a paid developer account or a free account on App Studio (http://apps.windowsstore.com), the web tool created by Microsoft for easily creating Windows Phone apps without programming skills.

Once you've met these requirements, you can download the Preview for Developers application from the Store at: https://go.microsoft.com/fwlink/p/?LinkId=324357.

After installing it, you'll have to run it and enable the preview program by accepting the terms and conditions. Once you've completed the process, preview updates will be delivered like regular updates: you'll have to go to the **Updates** section of the **Settings** page and check for new updates. At the time of writing, Microsoft is delivering GDR3 with this preview program.

Just keep in mind that depending on the manufacturer, installing a preview version can break the phone's warranty. It's a safe procedure, but if something goes wrong, your manufacturer may not be able to support you.

The first project

The starting point of every Windows Phone application is Visual Studio 2012. Let's see how to create a project and how it is structured.

The first step is to open Visual Studio 2012 and click on **New project**. In the available installed templates, you'll find the **Windows Phone** section that contains all the templates related to Windows Phone development.

We're going to use the simplest one, **Windows Phone app**, which contains only a starting page. Give it a name and click **OK**. You'll be asked which platform you're going to support. Since we're covering Windows Phone 8 development, choose **Windows Phone 8.0**. The designer will automatically load the starting page and, in the **Solution Explorer** window, you'll see the structure of the project. Let's look at it in detail:

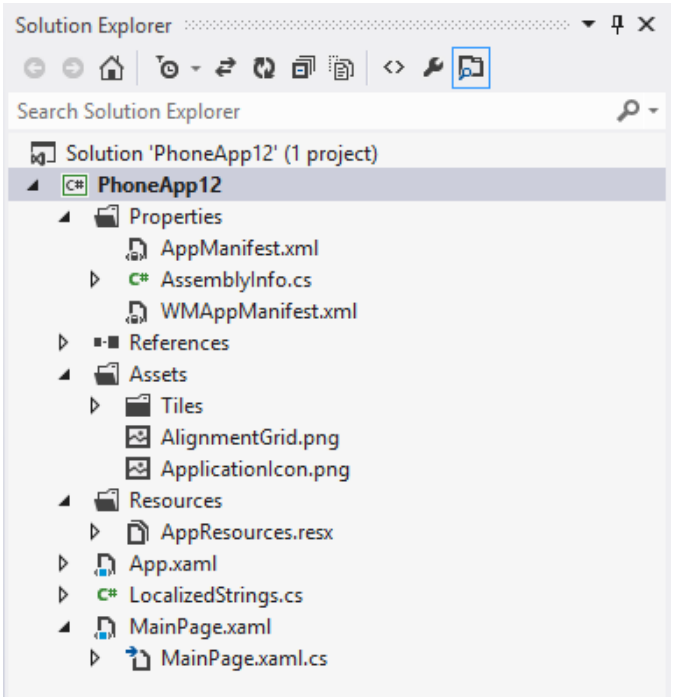

Figure 1: Structure of a Standard Windows Phone Project

App.xaml

The **App.xaml** file is the starting point of every Windows Phone application: it takes care of initializing the application and the frame that will contain the pages. Plus, since an instance of the **App** class (which is defined in the **App.xaml.cs** file) is always kept alive during the application execution, it's also used to define all the global settings. For example, you can intercept the life-cycle event we'll cover in Chapter 3, or you can define global XAML styles and resources that will be used across the application.

MainPage.xaml

This is the main page of the application that is launched after the app is initialized. It's the default one included in a project, but you can add as many pages as you want in your project. Every page has the same structure: it's composed of a file with the extension **.xaml**, which defines the user interface, and a file with extension **.cs**, which is the **code behind** that defines the logic for interacting with the user interface. Every page inherits from a class called **PhoneApplicationPage** that offers built-in support for properties and events that are connected to the page life cycle, such as navigation events, orientation, system tray management, and more.

Assets

The **Assets** folder contains the graphic assets of the application. The standard project includes some default icons and images for the various Tile sizes and templates.

Resources

The **Resources** folder contains all the files needed to manage localization. By default, you'll find just one file called **AppResources.resx**, which contains the base language (usually English, but you can select another). Every other language will be supported by adding a new AppResources file. The **LocalizedStrings.cs** file is added by default in the project's root, and it is the class that we will use to manage localization. We'll cover this more deeply in Chapter 10.

The manifest file

Inside the **Properties** folder you'll find a file called **WMAppManifest.xml**. This is a very important file. It is called manifest because it's used to declare all the capabilities and features of the application. Its role is crucial during the certification process; thanks to this file, the automatic process is able to extract all the needed information about the application, like its title, supported resolutions, features that are used, etc.

Visual Studio 2012 provides a visual editor for the manifest file; simply double-click on the file to open it. It's important to note that not all use-case scenarios are supported by the visual editor. Sometimes we'll have to manually edit the XML to extend our application.

The editor is split into four different sections:

- **Application UI**: Features all the information about the look of the application once it is installed on the phone, like the title, supported resolutions, template, and standard images to use as the main Tile.
- **Capabilities**: Lists all the hardware and software features the application can use, like the camera, geolocalization services, networking, etc. In most cases, using a feature for which the specific capabilities have not been declared will lead to an exception when the application is executed. In this book I will note every time we use a feature that requires a specific capability.
- **Requirements**: Lists specific hardware features that can be used by your app, like the camera or NFC. If you set requirements, people that own phones without these specific hardware features won't be able to download the app.
- **Packaging**: Defines the features of the package that will be published in the store, like the author, publisher, default language, and supported languages.

The splash screen

If you've already developed apps for Windows Phone 7, you should be familiar with the splash screen. It's a static image that is immediately displayed when the app is opened and disappears when the app is fully loaded and ready to be used.

The splash screen was part of the standard Visual Studio template for Windows Phone 7 projects, but it has been removed in Windows Phone 8 projects. Due to the performance improvements introduced in the Windows Runtime, apps now start much faster, so typically a splash screen is not needed.

If you do need a splash screen, you'll have to manually add it to the project following a specific naming convention: it has to be in JPG format, the file name has to be **SplashScreenImage.jpg**, and the required resolution is **768 × 1280**. This way, the image will be automatically adapted to the resolution of the device. If you want full control over the device's resolution, you can add three different images, one for each supported resolution. In this case, the naming convention to follow is:

- **SplashScreenImage.screen-WVGA.jpg** for 480 × 800 devices.
- **SplashScreenImage.screen-WXGA.jpg** for 768 × 1280 devices.
- **SplashScreenImage.screen-720p.jpg** for 720 × 1280 devices.

The 1080p resolution, from an aspect ratio point of view, behaves like 720p: if your application is launched on a 1080p device, the 720p splash screen will be used if one exists.

Testing the application

When you're ready to test your application, you can deploy it on a device or in the emulator directly from Visual Studio. In the toolbar area you will find what looks like a play button, together with a description of a deployment target. You can choose between five different targets: a real device, two versions of the WVGA emulator (one with 512 MB of RAM and one with 1 GB of RAM), a WXGA emulator, and a 720p emulator. From time to time, the list may be longer because Microsoft periodically releases SDK updates to add new emulators that match the new release of the operating system. For example, Microsoft has already released an SDK update that adds the emulator images aligned with the GDR2 release.

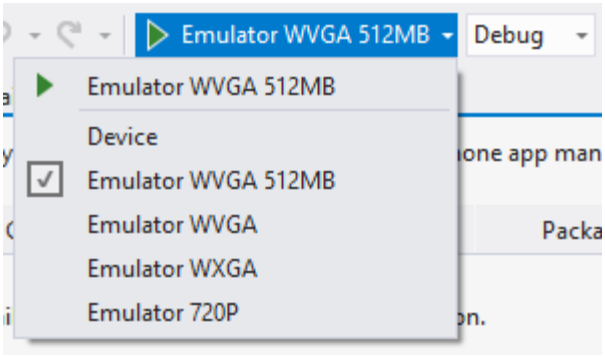

Figure 2: Options for Debugging a Windows Phone Application with the Default SDK

When the application is running in debug mode, two features are automatically enabled to help you work:

- The right side of the screen displays performance counters like the number of frames per second or the memory usage. They will help you identify potential performance issues with your app (see Figure 3 for more detailed information).
- The phone or emulator will never be suspended—the "auto lock" feature that turns the screen off after not being used is usually enabled on devices, but in debug mode, this feature is disabled.

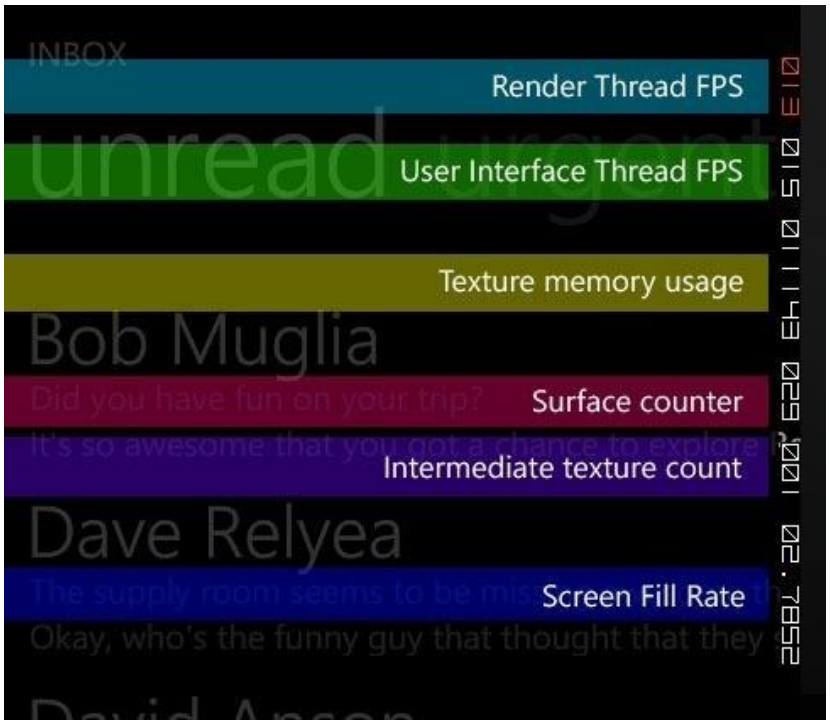

Figure 3: Various Counters Available in Debug Mode

A quick recap

In this first chapter we've started to get comfortable with some important concepts that every Windows Phone developer should be familiar with:

- We've learned the main software and hardware features of the platform that developers can take advantage of.
- We discussed the Windows Runtime that is the core of the newest Microsoft technologies, like Windows 8 and Windows Phone 8.
- We've seen how to start working with Windows Phone: which tools to download and install, how to create the first project, and the structure of a Windows Phone application.

Chapter 2 The User Interface: Basic XAML Concepts

What is XAML?

XAML is the acronym for Extensible Application Markup Language. It's a markup language based on XML, and its purpose and philosophy are very similar to HTML. Every control that can be placed on a page, whether a button, text box, or custom controls, is identified by a specific XML tag. Like XML, the structure is hierarchical; you can place tags inside other tags. For example, this hierarchical structure is how you can define the layout of a page, thanks to some controls that act as a container for other controls, like **`Grid`** or **`StackPanel`**.

The following is a sample of the XAML that defines a Windows Phone page:

```
<phone:PhoneApplicationPage
    x:Class="FirstApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <StackPanel>
                <TextBlock Text="This is a page" />
            </StackPanel>
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

PhoneApplicationPage is the base class of a Windows Phone page. As you can see, every other control is placed inside it. Notice also the **x:Class** attribute; it identifies which is the code-behind class that is connected to this page. In this sample, the code that is able to interact with the page will be stored in a class called **MainPage** that is part of the **FirstApp** namespace. We can see this class simply by clicking on the black arrow near the XAML file in Solution Explorer. You'll see another file with the same name of the XAML one plus the **.cs** extension.

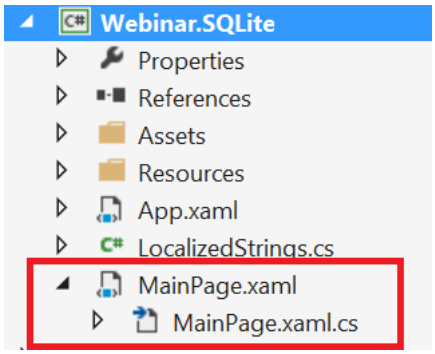

Figure 4: Visual Representation of a Page with Its Code-Behind File

Let's begin analyzing this simple XAML to introduce some key concepts, like namespaces and resources.

Namespaces

You should already be familiar with namespaces; they're a way to structure your code by assigning a logical path to your class.

By default, Visual Studio assigns namespaces using the same folder structure of the project. This means that if, for example, you have a class called **MyClass** stored inside a file in the **Classes** folder, the default full namespace of your class will be **Classes.MyClass**.

Namespaces in XAML work exactly the same way. The XAML controls are, in the end, classes that are part of your project, so you have to tell the page where it can find them. In the standard page you can see many examples of namespace declarations:

```
xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
```

Every namespace starts with the **xmlns** prefix, which is the standard XML namespace, followed by a custom prefix (in this sample it's **phone**). This prefix is very important, because it's the one that we're going to use in the rest of the page to add the controls. Then, we define the full namespace that contains the controls. If the class is part of our project, it's enough to specify just the namespace; otherwise, we also need to define which assembly (that is the DLL's name) contains the class.

In the previous example, we want to include controls and resources in our page that are defined inside the **Microsoft.Phone.Controls** namespace, which is included in the **Microsoft.Phone.dll** library.

The **PhoneApplicationPage** class gives you an example of how to use a namespace. Since the **PhoneApplicationPage** class is part of the **Microsoft.Phone.Controls** namespace, we have to add the prefix **phone** to the tag to use it:

```
<phone:PhoneApplicationPage />
```

It's very important to understand how namespaces in XAML work, because we'll need to declare them every time we use third-party controls (that we created on our own or are part of an external library) or resources, like converters.

Properties and events

Every control can be customized in two ways: by setting properties and actions. Both are identified by attributes of the XAML tag, but they have two different purposes.

Properties are used to change the look or the behavior of the control. Usually, a property is simply set by assigning a value to the specific attribute. For example, if we want to assign a value to the **Text** property of a **TextBlock** control, we can do it in the following way:

```
<TextBlock Text="This is a text block" />
```

There's also an extended syntax that can be used in the case of a complex property that can't be defined with a plain string. For example, if we need to set an image as a control's background, we need to use the following code:

```
<Grid>
    <Grid.Background>
        <ImageBrush ImageSource="/Assets/Background.png" />
    </Grid.Background>
</Grid>
```

Complex properties are set by using a nested tag with the name of the control plus the name of the property, separated by a dot (to set the **Background** property of the **Grid** control, we use the **Grid.Background** syntax).

One important property that is shared by every control is **x:Name**, which is a string that univocally identifies the control in the page. You can't have two controls with the same name in a single page. Setting this property is very important if you need to interact with the control in the code behind—you'll be able to refer to it by using its name.

Events are a way to manage user interactions with the control. One of the most used is **Tap**, which is triggered when users tap the control.

```
<Button Tap="OnButtonClicked" />
```

When you define an action, Visual Studio will automatically prompt you to create an **event handler**, which is the method (declared in the code behind) that is executed when the event is triggered.

```
private void OnButtonClicked(object sender, GestureEventArgs e)
{
    MessageBox.Show("Hello world");
}
```

In the previous example, we display the classic “Hello world” message to users when the button is pressed.

Resources

As in HTML, we are able to define CSS styles that can be reused in different parts of the website or the page. XAML has introduced the concept of **resources** that can be applied to different controls in an application.

Basically every XAML control supports the **Resources** tag: thanks to the hierarchy structure, every other nested control will be able to use it. In the real world, there are two common places to define a resource: at page and application level.

Page resources are defined in a single page and are available to all the controls that are part of that page. They are placed in a specific property called **Resources** of the **PhoneApplicationPage** class.

```
<phone:PhoneApplicationPage.Resources>
    <!-- you can place resources here -->
</phone:PhoneApplicationPage.Resources>
```

Application resources, instead, are globally available and they can be used inside any page of the application. They are defined in the **App.xaml** file, and the standard template already includes the needed definition.

```
<Application.Resources>
    <!-- here you can place global resources -->
</Application.Resources>
```

Every resource is univocally identified by a name that is assigned using the **x:Key** property. To apply a resource to a control, we need to introduce the concept of **markup extensions**. These are special extensions that allow us to apply different behaviors that would otherwise need some code to properly work. There are many markup extensions in the XAML world, and the one needed to apply a resource is called **StaticResource**. Here is an example of how to use it:

```
<TextBlock Text="MY APPLICATION" Style="{StaticResource
PhoneTextNormalStyle}" />
```

In this sample, the resource is applied to the **Style** property by including the **StaticResource** keyword inside braces, followed by the resource name which is the value of the **x:Key** property.

Resources can be also defined in an external file called **ResourceDictionary** if you want to better organize your project. To do this, right-click on your project in Visual Studio, click **Add** > **New Item**, and choose **XML file**. Give the file a name that ends with the .xaml extension and include the following definition:

```
<ResourceDictionary
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <!-- put resources here -->

</ResourceDictionary>
```

Now you can add it to your project by declaring it in the **App.xaml**:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Assets/Resources/Styles.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Notice the **MergedDictionaries** property: all the external resource files should be declared here. This way, they will be automatically merged and every resource declared in each external file will be available to every page, as if they were declared inline.

Let's see now, in detail, which are the most important kind of resources available.

Styles

XAML styles work the same way as CSS styles: you can define the values of different properties together in a single style that can be applied to multiple controls so that they all use the same layout. Here is how a style definition looks:

```
<Style x:Key="CustomText" TargetType="TextBlock">
    <Setter Property="FontSize" Value="24" />
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

A style is defined by a **`Style`** tag, which has two important attributes: **`x:Key`**, the name of the style, and **`TargetType`**, the type of controls that will be suitable for this style.

Inside the **`Style`** tag you can place as many **`Setter`** tags as you want. Each one identifies a control's property you want to change. Every **`Setter`** tag needs two attributes: **`Property`** is the control's property you want to change, and **`Value`** is the value you want to assign to the property.

The style defined in the previous example can be applied to any **`TextBlock`** control. Its purpose is to change the size of the font to 24 and to apply a bold style to the text.

There are also special types of styles called **implicit styles**. They are defined in the same way as the styles in the earlier example, except that the **`x:Key`** attribute is missing. In this case, the style is automatically applied to every control that is compliant with the value of the **`TargetType`** property, according to the scope where the style has been defined. If the style is set as a page resource, it will be applied only to the controls of the page; if the style is set as an application resource, it will be applied to every control in the application.

Data Templates

Data templates are a special type of resource that can be applied to a control to define its appearance. Data templates are often used with controls that are able to display collections of elements, like **`ListBox`** or **`LongListSelector`**.

The following sample shows a data template:

```
<DataTemplate x:Key="PeopleTemplate">
    <StackPanel>
        <TextBlock Text="Name" />
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock Text="Surname" />
        <TextBlock Text="{Binding Path=Surname}" />
    </StackPanel>
</DataTemplate>
```

A data template simply contains the XAML that will be used to render the specific item. If, for example, we apply this data template to the **ItemTemplate** property of a **ListBox** control, the result will be that the defined XAML will be repeated for every item in the collection (for the moment, just ignore the **Binding** markup extension; we'll deal with it later when we talk about data binding).

As for every other resource, data templates can be assigned to a property using the **StaticResource** markup extension.

```
<ListBox ItemTemplate="{StaticResource PeopleTemplate}" />
```

Animations

XAML is a powerful language because it allows us to do more than just create the layout of applications. One of the most interesting features is the animation feature, which can be created using the **Storyboard** control.

The **Storyboard** control can be used to define different types of animations:

- **DoubleAnimation**, which can be used to change the numeric value of a property (for example, **Width** or **FontSize**).
- **ColorAnimation**, which can be used to interact with properties that define a color (like inside a **SolidColorBrush**).
- **PointAnimation**, which can be applied to properties that define a point coordinate.

The following sample code defines an animation:

```
<Storyboard x:Name="Animation">
    <DoubleAnimation Storyboard.TargetName="RectangleElement"
                     Storyboard.TargetProperty="Width"
                     From="200"
                     To="400"
                     Duration="00:00:04" />
</Storyboard>
```

The first two properties are inherited from the **Storyboard** control: **Storyboard.TargetName** is used to set the name of the control we're going to animate, while **Storyboard.TargetProperty** is the property whose value we're going to change during the animation.

Next, we define the animation's behavior: the initial value (the **From** property), the ending value (the **To** property) and the duration (the **Duration** property). The behavior defined in the previous sample animates a **Rectangle** control by increasing its width from 200 to 400 over 4 seconds.

We can also control the animation more deeply by using the **UsingKeyFrames** variant available for every animation type:

```
<Storyboard x:Name="Animation">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="RectangleElement"
                                   Storyboard.TargetProperty="Width">
        <LinearDoubleKeyFrame KeyTime="00:00:00" Value="200" />
        <LinearDoubleKeyFrame KeyTime="00:00:02" Value="250" />
        <LinearDoubleKeyFrame KeyTime="00:00:04" Value="500" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

This way you're able to control the animation's timing. In the previous sample, the animation's type is the same (it's a **DoubleAnimation**), but we're able to set, for a specific time, which is the value to apply using the **LinearDoubleKeyFrame** tag.

In the previous sample, the **Width** of the **Rectangle** control is set to 200 at the beginning. Then, after two seconds, it increases to 250 and after four seconds, it is set to 500.

Easing animations

Another way to create animations is to use mathematical formulas that are able to apply a realistic behavior to an object, like bouncing, or acceleration and deceleration. You could achieve the same result by using key frames, but it would require a lot of work. For this reason, the animation framework offers a set of predefined easing functions that can be easily applied to an animation.

To add an easing function, you just need to set the **EasingFunction** property of an animation, as shown in the following sample:

```
<Storyboard x:Name="EasingAnimation">
    <PointAnimation From="0,0" To="0, 200" Duration="00:00:3"
                                   Storyboard.TargetName="Circle"
                                   Storyboard.TargetProperty="Center">
        <PointAnimation.EasingFunction>
            <BounceEase Bounces="2" EasingMode="EaseOut"  />
        </PointAnimation.EasingFunction>
    </PointAnimation>
</Storyboard>
```

After you've defined a regular animation (in the example, it's a **PointAnimation** that moves an **Ellipse** object from the coordinates (0, 0) to (0, 200)), you can set the **EasingFunction** property with one of the available easing functions. This example shows how to use the **BounceEase** function, which can be used to apply a bouncing effect to the object (the number of bounces performed is specified with the **Bounces** property).

Other available easing functions are:

- **BackEase**, which retracts the motion of the animation slightly before it starts.
- **CircleEase**, which applies a circular function to the acceleration animation.
- **ElasticEase**, which creates an animation that resembles an oscillating spring.

The official MSDN documentation features a complete list of the available easing functions.

How to control animations

Animations are treated like resources. They can be defined as local resources, page resources, or application resources. Unlike traditional resources, **Storyboard** controls are identified by the **x:Name** property, like a regular control.

The following sample shows an animation that is set as a page resource:

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="Animation">
        <DoubleAnimation Storyboard.TargetName="RectangleElement"
                         Storyboard.TargetProperty="Width"
                         From="200"
                         To="400"
                         Duration="00:00:04" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Thanks to the unique identifier, you'll be able to control the animation in the code behind. Every **Storyboard** object offers many methods to control it, like **Begin()**, **Stop()**, or **Resume()**. In the following code you can see the event handlers assigned to two buttons that are used to start and stop the animation:

```
private void OnStartClicked(object sender, GestureEventArgs e)
{
    Animation.Begin();
}

private void OnStopClicked(object sender, GestureEventArgs e)
{
    Animation.Stop();
}
```

Data binding

Data binding is one of the most powerful features provided by XAML. With data binding, you'll be able to create a communication channel between a UI element and various data sources, which can be another control or a property in one of your classes. Moreover, data binding is heavily connected to the XAML notification system (which we'll detail later) so that every time you change something in your object, the control displaying it will be automatically updated to reflect the changes and display the new value.

When you create a communication channel using data binding, you define a **source** (which contains the data to display) and a **target** (which takes care of displaying the value). By default, the binding channel is set to **`OneWay`** mode. This means that when the **source** changes**,** the **target** is updated to display the new value, but not vice versa. If we need to create a two-way communication channel (for example, because the target is a **`TextBox`** control and we need to intercept a new value inserted by the user), we can set the **`Mode`** property of the binding to **`TwoWay`**.

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

Almost every control in the XAML can participate in data binding. Most of the properties available for a control, in fact, are **dependency properties**. Beyond offering basic read and write capabilities, these are special properties that support notifications, so that they can notify the other side of the channel that something has changed.

The following example shows how data binding can be used to create a channel between two XAML controls:

```
<StackPanel>
    <Slider x:Name="Volume" />
    <TextBlock x:Name="SliderValue" Text="{Binding ElementName=Volume,
Path=Value}" />
</StackPanel>
```

The first thing to notice is that to apply binding, we need to use another **markup extension**, called **`Binding`**. With this expression, we connect the **`Text`** property of a **`TextBlock`** control (the target) to the **`Value`** property of a **`Slider`** control named **`Volume`** (the source).

Since both **`Text`** and **`Value`** are dependent properties, every time you move the slider, the selected value will be automatically displayed on the screen in the **`TextBlock`** control.

Data binding with objects

One of the most powerful data binding features is the ability to connect controls with objects that are part of your code. However, first, we need to introduce the **DataContext** concept. **DataContext** is a property that is available for almost every control and can be used to define its binding context, which is also automatically inherited by every nested control. When you define an object as **DataContext**, the control and all its children will have access to all its properties.

Let's see an example that will help you better understand how it works. Let's say that you have a class that represents a person:

```
public class Person
{
    public string Name { get; set; }
    public string Surname { get; set; }
}
```

Our goal is to display information about a person using this class. Here is how we can do it using data binding. First, let's take a look at the code behind:

```
public MainPage()
{
    InitializeComponent();
    Person person = new Person();
    person.Name = "Matteo";
    person.Surname = "Pagani";
    Author.DataContext = person;
}
```

When the page is initialized, we create a new **Person** object and set a value for the **Name** and **Surname** properties. Then, we set this new object as **DataContext** of the **Author** control. Let's see in the XAML page the **Author** control and how the **Name** and **Surname** properties are displayed:

```
<StackPanel x:Name="Author">
    <TextBlock Text="Name" />
    <TextBlock Text="{Binding Path=Name}" />
    <TextBlock Text="Surname" />
    <TextBlock Text="{Binding Path=Surname}" />
</StackPanel>
```

Author is the name assigned to a **StackPanel** control, which is the container we've placed inside different **TextBlock** controls. In the previous sample we can see the **Binding** markup extension in action again, this time with a different attribute: **Path**. We're using it to tell the XAML which property of the current **DataContext** to display. Since the **DataContext** is inherited from the **StackPanel** control, every **TextBlock** has access to the properties of the **Person** object we've created in the code behind. Notice that the **Path** attribute is optional. The two following statements are exactly the same:

```
<TextBlock Text="{Binding Path=Name}" />
<TextBlock Text="{Binding Name}" />
```

The INotifyPropertyChanged interface

The previous code has a flaw. Everything works fine, but if you change the value of one of the **Name** or **Surname** properties during the execution, the user interface won't be updated to display the new value. The reason is that **Name** and **Surname** are simple properties, so they aren't able to notify the user interface that something has changed, unlike dependency properties. For this scenario, the XAML framework has introduced the **INotifyPropertyChanged** interface that can be implemented by objects that need to satisfy this notification requirement. Here is how the **Person** class can be changed to implement this interface:

```
public class Person: INotifyPropertyChanged
{
    private string _name;
    private string _surname;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            OnPropertyChanged();
        }
    }


    public string Surname
    {
        get { return _surname; }
        set
        {
            _surname = value;
            OnPropertyChanged();
        }
    }
```

```
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null) handler(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

The class now implements the **INotifyPropertyChanged** interface, allowing us to support an event handler (called **PropertyChangedEventHandler**) that is triggered every time a property's value changes. The class also implements a method called **OnPropertyChanged()** that acts as a wrapper of the event handler and needs to be invoked when a property changes.

We also need a change in our properties. Every time the set of the property is called (meaning that a new value has been assigned) we raise the **OnPropertyChanged()** method. The result will be that every control bound with the property will be notified of the change and will update its visual status accordingly.

Data binding and collections

Data binding is especially helpful when you have to deal with collections of objects like arrays or lists (basically, every framework's collection types that implement the **IEnumerable** interface). Almost every control that supports collections, which inherit from the **ItemsControl** class (like **ListBox** or **LongListSelector**), has a property called **ItemsSource**, which can be directly assigned to a list.

You can control how every object of the collection will be rendered by using the **ItemTemplate** property. As we've seen when we talked about data templates, this property allows us to set which XAML to use to display the object.

Now that we've talked about data binding, there's another important piece to add. In the sample code we used to show data templates, we included some binding expressions to display the name and surname of a person.

```
<DataTemplate x:Key="PeopleTemplate">
    <StackPanel>
        <TextBlock Text="Name" />
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock Text="Surname" />
        <TextBlock Text="{Binding Path=Surname}" />
    </StackPanel>
</DataTemplate>
```

When you set a collection as **ItemSource**, every object that is part of it becomes the **DataContext** of the **ItemTemplate**. If, for example, the **ItemsSource** property of a **ListBox** is connected to a collection whose type is **List<Person>**, the controls included in the **ItemTemplate** will be able to access all the properties of the **Person** class.

This is the real meaning of the previous sample code: for every **Person** object that is part of the collection, we're going to display the values of the **Name** and **Surname** properties.

Another important piece of the puzzle when you deal with collections is the **ObservableCollection<T>** class. It acts like a regular collection, so you can easily add, remove, and move objects. Under the hood, it implements the **INotifyPropertyChanged** interface so that every time the collection is changed, the UI receives a notification. This way, every time we manipulate the collection (for example, we add a new item), the control that is connected to it will automatically be updated to reflect the changes.

Converters

Converters play an important role in data binding. Sometimes, in fact, you need to modify the source data before it is sent to the target. A common example is when you have to deal with **DateTime** properties. The **DateTime** class contains a full representation of a date, including hours, minutes, seconds, and milliseconds. Most of the time, however, you don't need to display the full representation—often the date is just enough.

This is where converters come in handy. You are able to change the data (or, as shown in the following example, apply different formatting) before sending it to the control that is going to display it using data binding.

To create a converter, you need to add a new class to your project (right-click in Visual Studio, choose **Add** > **Class**), and it has to inherit from the **IValueConverter** interface. The following is a converter sample:

```
public class DateTimeConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        if (value != null)
        {
            DateTime date = (DateTime)value;
            return date.ToShortDateString();
        }
        return string.Empty;
    }

    public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
    {
```

```
        if (value != null)
        {
            DateTime date = DateTime.Parse(value.ToString());
            return date;
        }
        return DateTime.Now;
    }
}
```

When you support the **IValueConverter** interface, you are forced to implement two methods:

- **Convert()** is the method invoked when data from the source is sent to the target.
- **ConvertBack()** does the opposite—it is invoked when data from the target is sent back to the source.

Most of the time it's enough to implement the **Convert()** method that is supported by every binding. The **ConvertBack()** method, instead, is supported only when you have **TwoWay** binding.

Both methods receive some important information as input parameters:

- The value returned from the binding source (which is the one you need to manipulate).
- The property that the binding has been applied to.
- An optional parameter that can be set in XAML using the **ConverterParameter** property. This parameter can be used to apply a different behavior in the converter logic.
- The current culture.

The previous code sample shows the **DateTime** example mentioned before. In the **Convert()** method we get the original value and, after we've converted it into a **DateTime** object, we return a string with the short formatting.

In the **ConvertBack()** method, we get the string returned from the control and convert it into a **DateTime** object before sending it back to the code.

Converters are treated like resources—you need to declare them and include them in your binding expression using the **StaticResource** keyword.

```
<phone:PhoneApplicationPage.Resources>
    <converters:DateTimeConverter x:Key="DateConverter" />
</phone:PhoneApplicationPage.Resources>
<TextBlock Text="{Binding Path=BirthDate, Converter={StaticResource
DateConverter}}" />
```

It's important to highlight that converters can have a negative impact on performance if you use them too heavily, since the binding operation needs to be reapplied every time the data changes. In this case, it's better to find a way to directly modify the source data or to add a new property in your class with the modified value.

Controls

The Windows Phone 8 SDK includes many built-in controls that can be used to define the user interface of the application. There are so many controls that it's almost impossible to analyze all of them in this book, so we'll take a closer look at just the most important ones.

Layout controls

Some controls simply act as containers for other controls and define the layout of the page. Let's discuss the most important ones.

StackPanel

The **`StackPanel`** control can be used to simply align the nested controls one below the other. It is able to automatically adapt to the size of the child controls.

```
<StackPanel>
    <TextBlock Text="First text" />
    <TextBlock Text="Second text" />
</StackPanel>
```

You can also use the **`StackPanel`** control to align controls horizontally, one next to the other, by setting the **`Orientation`** property to **`Horizontal`**.

```
<StackPanel Orientation="Horizontal">
    <TextBlock Text="First text" />
    <TextBlock Text="Second text" />
</StackPanel>
```

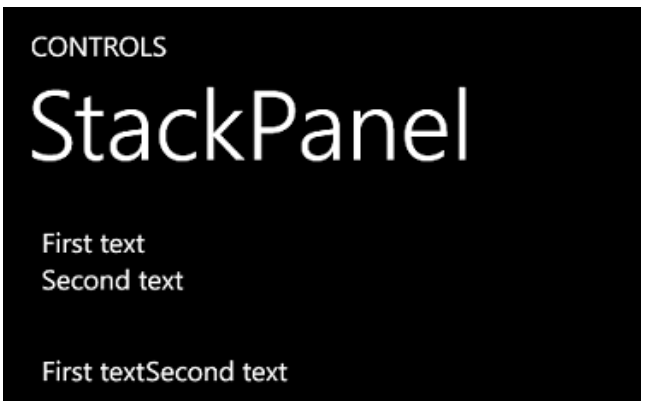

Figure 5: The StackPanel Control

Grid

The **Grid** control can be used to create table layouts, which fill the entire parent container's size. It supports rows and columns in which you can place the different controls. The following code sample demonstrates its use:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50" />
        <RowDefinition MaxHeight="100" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="200" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBlock Text="1° row - 2° column" Grid.Row="0" Grid.Column="1" />
</Grid>
```

You define the grid layout by using the **RowDefinitions** and **ColumnDefinitions** properties. You can add a **RowDefinition** tag for every row that the table will have, while the **ColumnDefinition** tag can be used to set the number of columns. For every row and column you can set the width and height, or you can simply omit them so that they automatically adapt to the nested controls.

To define a control's position inside the grid, we're going to use two attached properties, which are special dependency properties that are inherited from the **Grid** control and can be used with every control. With the **Grid.Row** property, we can set the row's number, and with **Grid.Column**, we can set the column's number.

The previous sample code is used to display a **TextBlock** in the cell that is placed in the first row, second column of the grid, as you can see in the following figure:

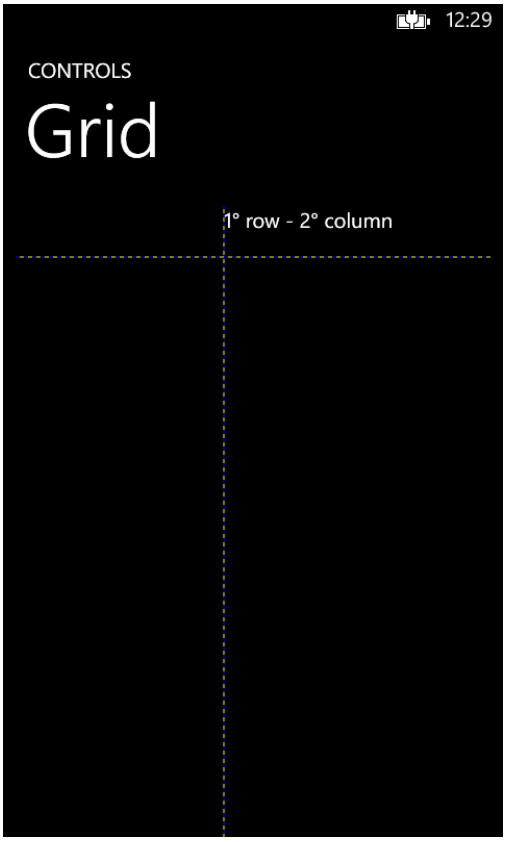

Figure 6: A Grid Control

ScrollViewer

The **`ScrollViewer`** control is a container, but it doesn't define a layout. If you want to arrange nested controls, you still have to use another container like a **`StackPanel`** or a **`Grid`**. The purpose of this control is to create a layout that is bigger than the size of the screen. This way, users will be able to scroll down to see the rest of the user interface.

This control is useful, for example, when you have to display text that doesn't fit the size of the page.

```
<ScrollViewer>
    <StackPanel>
        <TextBlock TextWrapping="Wrap" Text="This can be long text" />
    </StackPanel>
</ScrollViewer>
```

Border

The **`Border`** control's purpose is to display a border. This is useful because it's able to contain a child control that will be wrapped into the border.

The following sample shows how to wrap an image inside a red border:

```
<Border BorderThickness="5" BorderBrush="Red">
    <Image Source="/Assets/windows-phone-8-logo.png"/>
</Border>
```

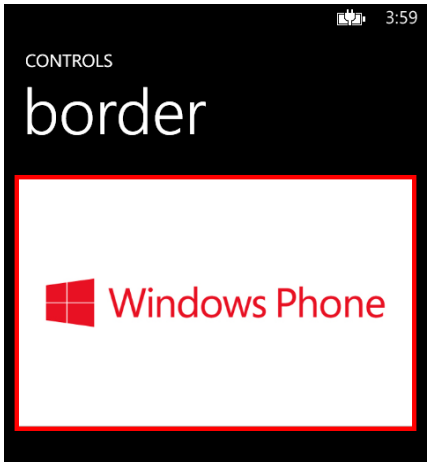

Figure 7: Border Control Used to Embed an Image

There are some key properties of the **`Border`** control. The first one is **`BorderThickness`**, which specifies the border's thickness. You can specify a single value, as we did in the previous sample. In this case, the same thickness is applied to every side. You can also specify multiple values to give every border a different size, as in the following sample:

```
<Border BorderThickness="5, 10, 15, 20" BorderBrush="Red">
    <Image Source="/Assets/windows-phone-8-logo.png"/>
</Border>
```

Figure 8: A Border Control with Different Border Thicknesses

The second important property is **BorderBrush**, which is used to set the brush that is applied to the border. It can use any of the available XAML brushes. By default, it accepts a **SolidColorBrush**, so you can simply specify the color you want to apply.

Another useful property is **Padding**, which can be used to specify the distance between the border and the child control, as shown in the following sample:

```
<Border BorderThickness="5" BorderBrush="Red" Padding="10">
    <Image Source="/Assets/windows-phone-8-logo.png"/>
</Border>
```

Figure 9: A Border Control with Padding

Output controls

The purpose of these controls is to display something to users, such as text, an image, etc.

TextBlock

TextBlock is one of the basic XAML controls and is used to display text on the screen. Its most important property is **Text**, which, of course, contains the text to display. You have many properties to choose from to modify the text's appearance, such as **FontSize**, **FontWeight**, and **FontStyle**, and you can automatically wrap the text in multiple lines in case the text is too long by setting the **TextWrapping** property to **true**.

```
<TextBlock Text="This is long and bold text" TextWrapping="Wrap"
FontWeight="Bold" />
```

You can also apply different formatting to the text without using multiple **TextBlock** controls by using the **Run** tag, which can be used to split the text as shown in the following sample:

```
<TextBlock>
    <Run Text="Standard text" />
    <LineBreak />
    <Run Text="Bold test" FontWeight="Bold" />
</TextBlock>
```

RichTextBlock

The **RichTextBlock** control is similar to **TextBlock**, but it offers more control over the formatting styles that can be applied to the text. Like the one offered by HTML, you can define paragraphs, apply different text styles, and more.

```
<RichTextBox>
    <Paragraph>
        <Bold>This is a paragraph in bold</Bold>
    </Paragraph>
    <Paragraph>
        <Italic>This is a paragraph in italics</Italic>
        <LineBreak />
    </Paragraph>
</RichTextBox>
```

Image

The **Image** control can be used to display images. You can set the **Source** property with a remote path (an image's URL published on the Internet) or a local path (a file that is part of your Visual Studio project). You can't assign a path that refers to an image stored in the local storage of the application. We'll see in Chapter 4 how to manage this limitation.

```
<Image Source="http://www.syncfusion.com/Content/en-
US/Home/Images/syncfusion-logo.png" />
```

You can also control how the image will be adapted to fill the control's size by using the **Stretch** property, which can have the following values:

- **Uniform**: The default value. The image is resized to fit the container while keeping the original aspect ratio so that the image won't look distorted. If the container's aspect ratio is different from the image's, the image will look smaller than the available space.
- **Fill**: The image is resized to fit the container, ignoring the aspect ratio. It will fill all the available space, but if the control's size doesn't have the same aspect ratio as the image, it will look distorted.
- **UniformToFill** is a mix of the previous values. If the image has a different aspect ratio than the container, the image is clipped so that it can keep the correct aspect ratio and, at the same time, fill all the available space.
- **None**: The image is displayed in its original size.

Input controls

These controls are used to get input from users.

TextBox

`TextBox` is another basic XAML control, and it's simply a way to collect text from users. The entered text will be stored in the **`Text`** property of the control. When users tap a **`TextBox`** control, the virtual keyboard is automatically opened. As a developer, you can control the keyboard's type that is displayed according to the data types you're collecting.

For example, you can display a numeric keyboard if users only need to input a number; or you can use an email keyboard (which provides easy access to symbols like @) if you're collecting an email address.

You can control the keyboard's type with the **`InputScope`** property. The list of supported values is very long and can be found in the MSDN documentation. Some of the most used are:

- **`Text`** for generic text input with dictionary support.
- **`Number`** for generic number input.
- **`TelephoneNumber`** for specific phone number input (it's the same keyboard that is displayed when you compose a number in the native Phone application).
- **`EmailNameOrAddress`** which adds quick access to symbols like @.
- **`Url`** which adds quick access to common domains like .com or .it (depending on the keyboard's language).
- **`Search`** which provides automatic suggestions.

```
<TextBox InputScope="TelephoneNumber" />
```

Tip: If the TextBox control is used to collect generic text, always remember to set the InputScope property to Text. This way, users will get support from the autocomplete and autocorrection tools.

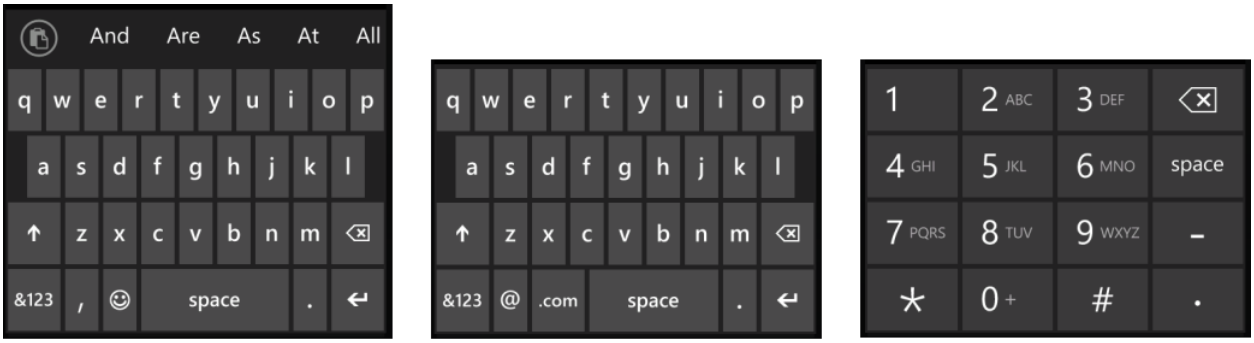

Figure 10: From Left to Right: Text, EmailNameOrAddress, and PhoneNumber Input Scopes

PasswordBox

PasswordBox works exactly like the **TextBox** control, except that the inserted characters are automatically converted into dots so that people near the user won't be able to read the text. As the name of the control implies, it's typically used to collect passwords.

Theme resources

One of a developer's goals should be to keep the user interface of her or his application as consistent as possible with the guidelines provided by the operating system. To help achieve this goal, the SDK offers many out-of-the-box resources that can be applied to controls to get the same look and feel of the native applications. These styles are typically used with controls like **TextBox**, **TextBlock**, or **RadioButton**, and they provide a standard set of visual features (like font size, color, opacity, etc.) that are consistent with the other applications.

Another good reason to use theme resources is that they are aware of the theme set by users. For example, you can use the **PhoneAccentBrush** style if you want to give a control the same color as the phone's accent color.

The many theme resources are split into different categories like brush resources, color resources, font names and styles, and text resources. You can find a complete list of the available styles in the MSDN documentation. They are simply applied using the **Style** property offered by every control, as shown in the following sample:

```
<TextBlock Text="App name" Style="{StaticResource PhoneTextNormalStyle}" />
```

Interacting with users

In this category, we can collect all the controls that we can use to interact with users, like **Button**, **CheckBox**, or **RadioButton**.

The text to display is set with the **Content** property, like in the following sample:

```
<Button Content="Tap me" Tap="OnClickMeClicked" />
```

The **Content** property can also be complex so that you can add other XAML controls. In the following sample we can see how to insert an image inside a button:

These controls offer many ways to interact with users. The most common events are **Click**, **Tap**, and **DoubleTap**.

```
<Button Tap="OnClickMeClicked">
    <Button.Content>
        <StackPanel>
            <Image Source="/Assets/logo.png" Height="200" />
        </StackPanel>
    </Button.Content>
</Button>
```

Note: Click and Tap are the same event—they are both triggered when users press the control. Tap has been introduced in Windows Phone 7.5 to be more consistent with the touch interface, but to avoid breaking old apps, the Click event is still supported.

The Windows Phone signature controls

Most of the controls we've seen so far are part of the XAML framework and are available on every other XAML-based technology, like Silverlight, WPF, and Windows Store apps.

However, there are some controls that are available only on the Windows Phone platform, since they are specific for the mobile experience. Let's take a look at them.

Panorama

The **`Panorama`** control is often used in Windows Phone applications since it's usually treated as a starting point. The control gets its name because an oversized image is used as the background of the page. Users are able to swipe to the left or to the right to see the other available pages. Since the image is bigger than the size of the page, the phone applies a parallax effect that is visually pleasing for users.

The other main feature of the **`Panorama`** control is that users can get a sneak peek of the next page. The current page doesn't occupy all of the available space because a glimpse of the next page is displayed on the right edge.

A **`Panorama`** control is typically used to provide an overview of the content that is available in an application. It's a starting point, not a data container. For example, it's not appropriate to use a panorama page to display all the news published on a blog. It's better, instead, to display only the latest news items and provide a button to redirect users to another page, where they will be able to see all of them.

From a developer's point of view, a **`Panorama`** control is composed of different pages: each one is a **`PanoramaItem`** control that contains the layout of the page.

The **`Panorama`** can have a generic title, like the application's title (which is assigned to the **`Title`** property), while every page can have its own specific title (which is assigned to the **`Header`** property).

```
<phone:Panorama Title="Panorama">
    <phone:PanoramaItem Header="First page">
        <StackPanel>
            <TextBlock Text="Page 1" />
        </StackPanel>
    </phone:PanoramaItem>
    <phone:PanoramaItem Header="Second page">
        <StackPanel>
            <TextBlock Text="Page 2" />
        </StackPanel>
    </phone:PanoramaItem>
</phone:Panorama>
```

Note: The Panorama control (like the Pivot control) is not available by default in a page. You'll have to declare the following namespace: ***`xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"`***

Pivot

The **`Pivot`** control, from a technical and user interaction perspective, works in a similar way to the **`Panorama`** control—users can swipe the screen left or right to view the other pages. The difference, from the users' point of view, is that the view will fit the screen's size. Users can see which page is next because its header is displayed next to the current page's header in gray.

Figure 11: The Pivot Control

However, the **Pivot** control is used for purposes not shared by the **Panorama** control:

- To display one kind of information applied to different contexts. The previous figure is a good example. The information on each page is the same (the weather forecast), but referred to in different contexts (the cities).
- To display different kinds of information that refer to the same context. A contact's details page in the People Hub on a Windows Phone is a good example of this—you have much information (the contact's details, social network updates, conversations, etc.), but it all belongs to the same context (the contact).

As previously anticipated, the XAML for the **Pivot** control works like the XAML for the **Panorama** control. The main control is called **Pivot**, while the nested controls that represent the pages are called **PivotItem**.

```
<phone:Pivot Title="Pivot">
    <phone:PivotItem Header="First page">
        <StackPanel>
            <TextBlock Text="Page 1" />
        </StackPanel>
    </phone:PivotItem>
    <phone:PivotItem Header="Second page">
        <StackPanel>
            <TextBlock Text="Page 2"/>
        </StackPanel>
    </phone:PivotItem>
</phone:Pivot>
```

The ApplicationBar

The **ApplicationBar** is a control placed at the bottom of the page and is used as a quick access for functions that are connected to the current view.

You can add two element types to an application bar:

- **Icons** are always displayed (unless the **ApplicationBar** is minimized), and up to four can be displayed at once.
- **Menu items** are simply text items that are displayed only when the application bar is opened. There's no limit to the number of items that can be included.

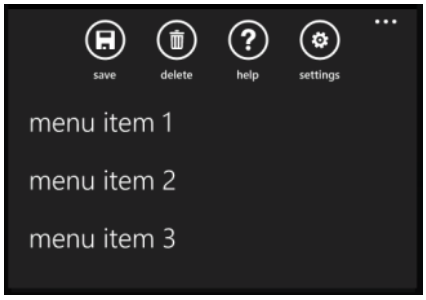

Figure 12: An Open ApplicationBar

The **ApplicationBar** does not behave like the other XAML controls. It is not part of the page—in fact, it's declared outside the main **Grid**, the one called **LayoutRoot**—and it doesn't inherit from the **FrameworkElement** class, like every other control. The biggest downside of this is that the control doesn't support binding; you'll have to rely on third-party libraries, like the implementation available in the Cimbalino Toolkit for Windows Phone.

Here is an **ApplicationBar** sample:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton IconUri="/Assets/Add.png"
Text="Add" Click="ApplicationBarIconButton_Click" />
        <shell:ApplicationBar.MenuItems>
            <shell:ApplicationBarMenuItem Text="update"
Click="ApplicationBarMenuItem_Click"/>
        </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

ApplicationBar is a property of the **PhoneApplicationPage** class, which contains the real **ApplicationBar** definition. It's not part of the standard XAML namespaces, but it's part of the following namespace, which should already be declared in every standard Windows Phone page:

```
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
```

Icon buttons are declared directly inside the **ApplicationBar** tag. The base class is **ApplicationBarIconButton**, and the most important properties are **Text** (the icon description) and **IconUri** (the path of the image used as the icon), while **Click** is the event handler that is invoked when a user taps the button.

Menu items, instead, are grouped inside a property of the **ApplicationBar** control called **MenuItems**. You can add as many **ApplicationBarMenuItem** controls as you want. They behave like the button icons except that, since they're just text, the **IconUri** property is missing.

Another important limitation of the **ApplicationBar** control is that we're not able to assign the **x:Name** property to an **ApplicationBarIconButton** or **ApplicationBarMenuItem** control. This means that if we need to change a property's value in the code behind, we can't simply use the notation *ControlName.PropertyName.*

The work-around is to use the code behind to directly access the collections that contain the buttons and menu items that are available because of the **ApplicationBar** object. They are called **Buttons** and **MenuItems**, as you can see in the following code sample:

```
ApplicationBarIconButton iconButton = this.ApplicationBar.Buttons[0] as
ApplicationBarIconButton;
iconButton.Text = "New text";
ApplicationBarMenuItem menuItem = this.ApplicationBar.MenuItems[0] as
ApplicationBarMenuItem;
menuItem.Text = "New text";
```

The purpose of this sample is to get access to the first icon button and first menu item inside the **ApplicationBar**, and to change the value of the **Text** property.

In the end, there are two other ways to customize the **ApplicationBar**. The first is to minimize it. This way, only the three dots at the right margin will be displayed; icons won't be visible. To achieve this, you have to set the **Mode** property to Minimized.

The other way is to change the opacity. You can set the **Opacity** property with a value between 0 (transparent) and 1 (opaque). The biggest difference is that when **ApplicationBar** is translucent, the page content will go below the bar and fit the entire size of the screen; when the bar is opaque, the content won't fit all of the available screen space since the bottom of the page will be reserved for the **ApplicationBar**.

Displaying collections of data with the LongListSelector

One of the most common requirements in an application is to display a collection of items which can be retrieved from a remote service or a local database. The Windows Phone SDK has included, since the beginning, some controls for this purpose, like **ItemsControl** and **ListBox**. In Windows Phone 8, Microsoft has introduced a new and more powerful control, which previously was available as part of the Windows Phone Toolkit (you'll find more details about this toolkit later in this chapter). This control is called **LongListSelector** and has many advantages over other similar controls:

- Better performance.
- Virtualization support to avoid loading all the data at the same time, instead loading data only when it is needed.
- Group support to turn the list into a **jump list**, so that data is grouped in categories and users can easily jump from one to another (for example, in the People Hub, contacts are grouped by the first letter).

Creating a flat list

The **LongListSelector** control can be used like a regular **ListBox** to display a flat list of items, without grouping. In this case, you just need to set the **IsGroupingEnabled** property to **false**. Except for this modification, you'll be able to use a **LongListSelector** like a standard **ItemsControl**. You'll define the **ItemTemplate** property with a **DataTemplate** to define the item layout, and you'll assign the collection you want to display to the **ItemsSource** property.

```
<phone:LongListSelector x:Name="List"
                        IsGroupingEnabled="False"
                        ItemTemplate="{StaticResource PeopleItemTemplate}"
/>
```

Creating a list grouped by letter

Creating a list grouped by the first letter of the items is a bit more complicated since we have to change our data source. It won't be a flat collection of data anymore. In addition, if we want to keep the user experience consistent with other applications, we'll need to create a jump list with all the letters of the alphabet. The groups that don't have any members will be disabled so that users can't tap them.

To achieve this result, Microsoft offers a class called **`AlphaKeyGroup<T>`**, which represents a letter of the alphabet and all the items that start with it. However, this class is not part of the Windows Phone SDK, and should be manually added to your project by right-clicking on your project in the Solution Explorer in Visual Studio, and choosing **Add new class**. The following code example is the full implementation.

```
public class AlphaKeyGroup<T> : List<T>
{
    /// <summary>
    /// The delegate that is used to get the key information.
    /// </summary>
    /// <param name="item">An object of type T.</param>
    /// <returns>The key value to use for this object.</returns>
    public delegate string GetKeyDelegate(T item);

    /// <summary>
    /// The key of this group.
    /// </summary>
    public string Key { get; private set; }

    /// <summary>
    /// Public constructor.
    /// </summary>
    /// <param name="key">The key for this group.</param>
    public AlphaKeyGroup(string key)
    {
        Key = key;
    }

    /// <summary>
    /// Create a list of AlphaGroup<T> with keys set by a
SortedLocaleGrouping.
    /// </summary>
    /// <param name="slg">The </param>
    /// <returns>The items source for a LongListSelector.</returns>
```

```
    private static List<AlphaKeyGroup<T>> CreateGroups(SortedLocaleGrouping
slg)
    {
        List<AlphaKeyGroup<T>> list = new List<AlphaKeyGroup<T>>();

        foreach (string key in slg.GroupDisplayNames)
        {
            list.Add(new AlphaKeyGroup<T>(key));
        }

        return list;
    }

    /// <summary>
    /// Create a list of AlphaGroup<T> with keys set by a
SortedLocaleGrouping.
    /// </summary>
    /// <param name="items">The items to place in the groups.</param>
    /// <param name="ci">The CultureInfo to group and sort by.</param>
    /// <param name="getKey">A delegate to get the key from an
item.</param>
    /// <param name="sort">Will sort the data if true.</param>
    /// <returns>An items source for a LongListSelector.</returns>
    public static List<AlphaKeyGroup<T>> CreateGroups(IEnumerable<T> items,
CultureInfo ci, GetKeyDelegate getKey, bool sort)
    {
        SortedLocaleGrouping slg = new SortedLocaleGrouping(ci);
        List<AlphaKeyGroup<T>> list = CreateGroups(slg);

        foreach (T item in items)
        {
            int index = 0;
            if (slg.SupportsPhonetics)
            {
             //Checks whether your database has the string yomi as an item.
             //If it does not, then generate Yomi or ask users for this
item.
             //index = slg.GetGroupIndex(getKey(Yomiof(item)));
            }
            else
            {
                index = slg.GetGroupIndex(getKey(item));
            }
            if (index >= 0 && index < list.Count)
            {
                list[index].Add(item);
            }
        }

        if (sort)
```

```
        {
            foreach (AlphaKeyGroup<T> group in list)
            {
                group.Sort((c0, c1) => { return
ci.CompareInfo.Compare(getKey(c0), getKey(c1)); });
            }
        }

        return list;
    }

}
```

The main features of this class are:

- It inherits from **`List<T>`**, so it represents a list of elements.
- It has a property called **`Key`**, which is the key that identifies the group (the letter of the alphabet).
- It uses a special collection type called **`SortedLocaleGroup`**, which is able to manage the cultural differences between one language and another.
- It offers a method called **`CreateGroups()`**, which is the one we're going to use to group our data.

To better explain how to use the **`AlphaKeyGroup<T>`** class, let's use a real example. Let's define a collection of people that we want to group by the first letters of their names, in a similar way the People Hub does. The first step is to create a class that represents a single person:

```
public class Person
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public string City { get; set; }
}
```

Then, when the application starts, we populate the list with a set of fake data, as shown in the following sample:

```
void LongListSelectorAlphabetic_Loaded(object sender, RoutedEventArgs e)
{
    List<Person> people = new List<Person>
    {
        new Person
        {
            Name = "John",
            Surname = "Doe",
            City = "Como"
        },
```

```
        new Person
        {
            Name = "Mark",
            Surname = "Whales",
            City = "Milan"
        },
        new Person
        {
            Name = "Ricky",
            Surname = "Pierce",
            City = "New York"
        }
    };
}
```

Now it's time to use the **`AlphaGroupKey<T>`** class to convert this flat list into a grouped list by calling the **`CreateGroups()`** method.

```
List<AlphaKeyGroup<Person>> list =
AlphaKeyGroup<Person>.CreateGroups(people,
    Thread.CurrentThread.CurrentUICulture,
    p => p.Name, true);
```

The method requires four parameters:

- The collection that we want to group: In the sample, it's the collection of **`Person`** objects we created.
- The culture to use to generate the alphabet letters: The standard practice is to use the value of the **`Thread.CurrentThread.CurrentUICulture`** property, which is the primary language set by the user for the phone.
- The object's property that will be used for grouping: This is specified using a lambda expression. In the sample, the list will be grouped by the first letter of the name.
- The last parameter, a **`Boolean`** type, is used to determine whether to apply ordering: If set to **`true`**, the collection will be alphabetically ordered.

What we get in return is a collection of **`AlphaKeyGroup<T>`** objects, one for each letter of the alphabet. This is the collection that we need to assign to the **`ItemsSource`** property of the **`LongListSelectorControl`**.

```
List<AlphaKeyGroup<Person>> list =
AlphaKeyGroup<Person>.CreateGroups(people,
Thread.CurrentThread.CurrentUICulture,
p => p.Name, true);

People.ItemsSource = list;
```

However, this code isn't enough—we also need to supply additional templates in XAML that are used to define the jump list layout.

The first property to set is called **GroupHeaderTemplate**, and it represents the header that is displayed in the list before every group. In the case of an alphabetic list, it's the letter itself. The following XAML code shows a sample template, which recreates the same look and feel of native apps:

```
<DataTemplate x:Key="PeopleGroupHeaderTemplate">
    <Border Background="Transparent" Padding="5">
        <Border Background="{StaticResource PhoneAccentBrush}"
BorderBrush="{StaticResource PhoneAccentBrush}" BorderThickness="2"
Width="62"
    Height="62" Margin="0,0,18,0" HorizontalAlignment="Left">
            <TextBlock Text="{Binding Key}" Foreground="{StaticResource
PhoneForegroundBrush}" FontSize="48" Padding="6"
    FontFamily="{StaticResource PhoneFontFamilySemiLight}"
HorizontalAlignment="Left" VerticalAlignment="Center"/>
        </Border>
    </Border>
</DataTemplate>
```

With this layout, the letter is placed inside a square with a background color the same as the accent color of the phone. Take note of two important things:

- Some properties of the **Border** control use the **PhoneAccentBrush** resource. It's one of the theme resources previously described, which identifies the accent color of the phone.
- The **Text** property of the **TextBlock** control is bound with the **Key** property of the **AlphaGroupKey<T>** class. This way, we are able to display the group's letter inside the square.

Figure 13: Typical Group Header Layout in a LongListSelector Control

The second property to define is called **JumpListStyle** and, unlike the previous property, it's not a template but a style. Its purpose is to define the look and feel of the jump list, which is the view that is displayed when users tap a letter. It displays all the letters of the alphabet so that users can tap one of them and quickly jump to that group.

Here is a sample definition which, again, recreates the look and feel of native applications—all the letters of the alphabet are displayed side by side in multiple rows.

```
<phone:JumpListItemBackgroundConverter x:Key="BackgroundConverter"/>
<phone:JumpListItemForegroundConverter x:Key="ForegroundConverter"/>
<Style x:Key="PeopleJumpListStyle" TargetType="phone:LongListSelector">
    <Setter Property="GridCellSize"  Value="113,113"/>
    <Setter Property="LayoutMode" Value="Grid" />
    <Setter Property="ItemTemplate">
        <Setter.Value>
            <DataTemplate>
                <Border Background="{Binding Converter={StaticResource
BackgroundConverter}}" Width="113" Height="113" Margin="6" >
                    <TextBlock Text="{Binding Key}"
FontFamily="{StaticResource PhoneFontFamilySemiBold}" FontSize="48"
Padding="6"
        Foreground="{Binding Converter={StaticResource
ForegroundConverter}}" VerticalAlignment="Center"/>
                </Border>
            </DataTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

This style uses two converters which are part of the Windows Phone Toolkit called **JumpListItemBackgroundConverter** and **JumpListItemForegroundConverter**. As you can see in the **ItemTemplate** that is applied to the control, these converters are used to define the text color and the **Border** background color. Their purpose is to manage empty groups. If the collection contains one or more items, it will be displayed in white with the phone's accent color as a background. If instead, the letter is associated to a group with no items, both the text and the background will be gray. This way, users can immediately see that the group is disabled, and tapping it won't produce any effect.

Figure 14: Typical Appearance of a LongListSelector Jump List

In the end, don't forget that as for every other control based on the **`ItemsControl`** class, you'll need to set the **`ItemTemplate`** property with a **`DataTemplate`** to define how a single item will look. The following sample shows a basic implementation, where name and surname are displayed one below the other:

```
<DataTemplate x:Key="PeopleItemTemplate">
    <StackPanel>
        <TextBlock Text="{Binding Name}" />
        <TextBlock Text="{Binding Surname}" />
    </StackPanel>
</DataTemplate>
```

Once you've gathered all the required styles and templates, you can apply them to the **`LongListSelector`** control, as shown in the following example:

```
<phone:LongListSelector
                    x:Name="People"
```

```
                    GroupHeaderTemplate="{StaticResource
PeopleGroupHeaderTemplate}"
                    ItemTemplate="{StaticResource PeopleItemTemplate}"
                    JumpListStyle="{StaticResource PeopleJumpListStyle}"
                    IsGroupingEnabled="True"
                    HideEmptyGroups="True" />
```

Notice the **HideEmptyGroups** property, which can be used to hide all the groups in the list that don't contain any items.

Creating a list grouped by category

In some cases, we might need to group items by a custom field instead of the first letter. Let's look at another example. We want to group **Person** objects by the **City** field; we need, then, a class to replace the **AlphaKeyGroup<T>**, since it's not suitable for our scenario.

Let's introduce the **Group<T>** class, which is much simpler:

```
public class Group<T> : List<T>
{
    public Group(string name, IEnumerable<T> items)
        : base(items)
    {
        this.Key = name;
    }

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

Under the hood, this class behaves like **AlphaKeyGroup<T>**. It inherits from **List<T>**, so it represents a collection of items, and defines a group which is identified by a key (which will be the category's name).

With this class, you'll be able to use LINQ to group the items that are included in a collection, as shown in the following sample:

```
private List<Group<T>> GetItemGroups<T>(IEnumerable<T> itemList, Func<T,
string> getKeyFunc)
{
    IEnumerable<Group<T>> groupList = from item in itemList
        group item by getKeyFunc(item)
        into g
        orderby g.Key
        select new Group<T>(g.Key, g);
```

```
    return groupList.ToList();
}
```

The previous method takes as input parameters:

- The flat collection to group.
- A function (expressed with a lambda expression) to set the object's property we want to use for grouping.

The following example is the code we can use to group the collection of **`Person`** objects by the **`City`** property:

```
List<Group<Person>> groups = GetItemGroups(people, x => x.City);
PeopleByCity.ItemsSource = groups;
```

The result returned by the **`GetItemGroups()`** method can be directly assigned to the **`ItemsSource`** property of the **`LongListSelector`** control.

Like we did in the alphabetical grouping scenario, we still need to define the layout of the group headers and the jump list. We can reuse the resources we've previously defined, but, if we want to achieve a better result, we can adapt them so that the background rectangle fills the size of the category's name, as shown in the following code example:

```
<phone:JumpListItemBackgroundConverter x:Key="BackgroundConverter"/>
<phone:JumpListItemForegroundConverter x:Key="ForegroundConverter"/>

<DataTemplate x:Key="PeopleCityGroupHeaderTemplate">
    <Border Background="Transparent" Padding="5">
        <Border Background="{StaticResource PhoneAccentBrush}"
BorderBrush="{StaticResource PhoneAccentBrush}" BorderThickness="2"
    Height="62" Margin="0,0,18,0" HorizontalAlignment="Left">
            <TextBlock Text="{Binding Key}" Foreground="{StaticResource
PhoneForegroundBrush}" FontSize="48" Padding="6"
    FontFamily="{StaticResource PhoneFontFamilySemiLight}"
HorizontalAlignment="Left" VerticalAlignment="Center"/>
        </Border>
    </Border>
</DataTemplate>

<Style x:Key="PeopleCityJumpListStyle" TargetType="phone:LongListSelector">
    <Setter Property="GridCellSize"  Value="113,113"/>
    <Setter Property="LayoutMode" Value="List" />
    <Setter Property="ItemTemplate">
        <Setter.Value>
            <DataTemplate>
                <Border Background="{Binding Converter={StaticResource
BackgroundConverter}}" Height="113" Margin="6" >
```

```
                    <TextBlock Text="{Binding Key}"
FontFamily="{StaticResource PhoneFontFamilySemiBold}" FontSize="48"
Padding="6"
        Foreground="{Binding Converter={StaticResource
ForegroundConverter}}" VerticalAlignment="Center"/>
                </Border>
            </DataTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

The following figure shows the result of the code sample.

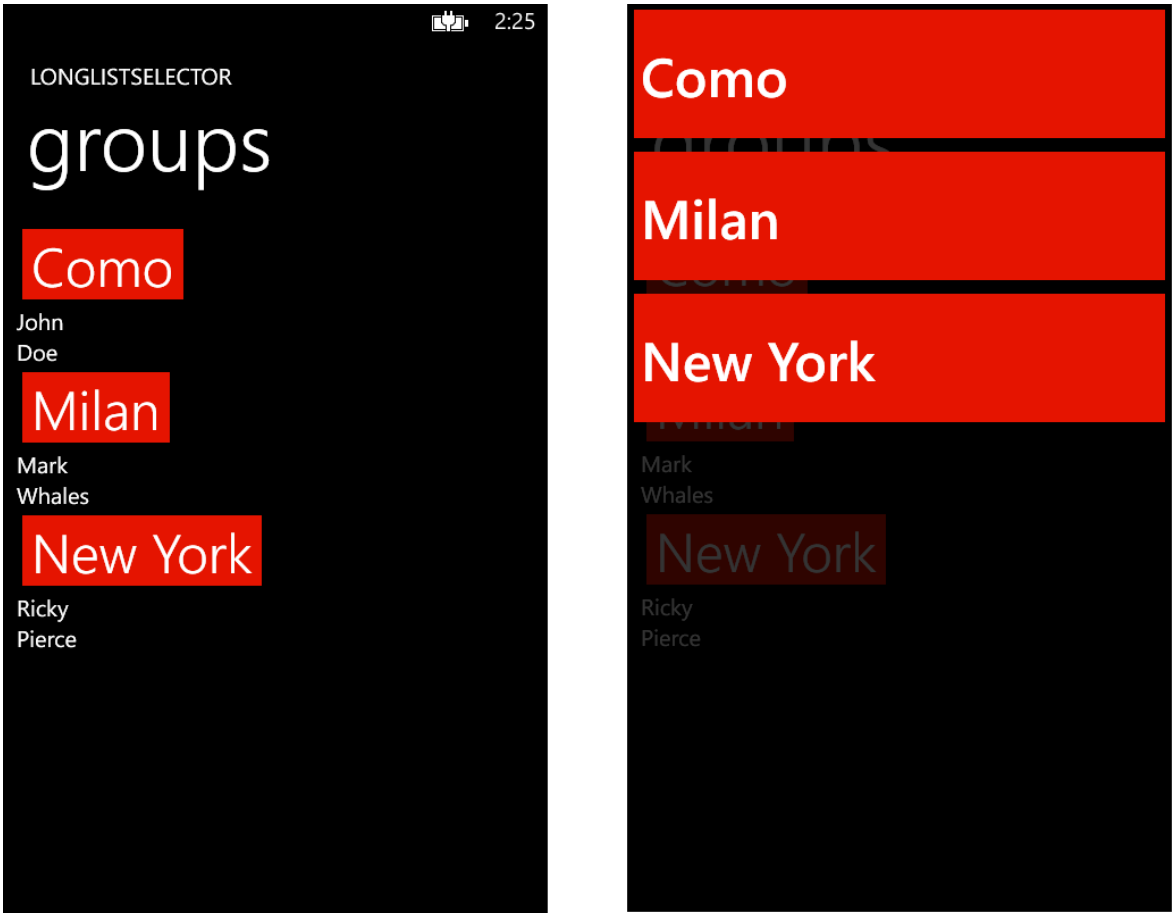

Figure 15: A LongListSelector Control Used to Group a Collection by a Custom Field

Remember to assemble the **LongListSelector** control by adding all the styles and templates you've just created, as shown in the following sample:

```
<phone:LongListSelector
        x:Name="PeopleByCity"
        ItemTemplate="{StaticResource PeopleItemTemplate}"
```

```
        GroupHeaderTemplate="{StaticResource
PeopleCityGroupHeaderTemplate}"
        JumpListStyle="{StaticResource PeopleCityJumpListStyle}"
        IsGroupingEnabled="True" />
```

Interacting with the list

The two key concepts to keep in mind when you interact with a **LongListSelector** control (or any other control that inherits from the **ItemsControl** class) are:

- The **SelectionChanged** event, which is triggered every time the user taps one of the elements on the list.
- The **SelectedItem** property, which stores the items that have been selected by the user.

With the combination of these two items, you'll be able to detect when and which item has been selected by the user, and properly respond. For example, you can redirect the user to a page where he or she can see the details of the selected item.

```
<phone:LongListSelector
        x:Name="People"
        GroupHeaderTemplate="{StaticResource PeopleGroupHeaderTemplate}"
        ItemTemplate="{StaticResource PeopleItemTemplate}"
        SelectionChanged="LongListSelectorAlphabetic_Loaded" />
```

The previous sample shows a **LongListSelector** control that has been subscribed to the **SelectionChanged** event. The following sample, instead, shows the event handler code:

```
private void People_OnSelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    Person selectedPerson = People.SelectedItem as Person;
    string uri = string.Format("/DetailPage.xaml?Id={0}",
selectedPerson.Id);
    NavigationService.Navigate(new Uri(uri, UriKind.Relative));
}
```

Thanks to the **SelectedItem** property, we retrieve the selected **Person** object and redirect the user to another page called **DetailPage.xaml** to display the details of the selected person. We'll cover navigation in-depth in the next chapter.

The Windows Phone toolkit

In this chapter we've seen just the basic controls, but you'll notice that the SDK is missing many controls that are used in other applications. Most of them are available in a library called Windows Phone Toolkit, which is available on CodePlex and NuGet. It's maintained directly by Microsoft and is a way to keep a separate, faster development process than the one needed to release a new SDK version.

Here is a brief list of the most important available controls:

- **`ToggleSwitch`**: Especially helpful for settings pages, since it's a switch that can be used to define a Boolean value (on/off).
- **`ContextMenu`**: A menu that can be displayed when users tap and hold an item.
- **`DatePicker`** and **`TimePicker`**: Used to select a date or time respecitvely.
- **`WrapPanel`**: A special container that can align nested controls next to each other and automatically wrap to a new line if no space remains.
- **`AutoCompleteBox`**: A special **`TextBox`** that can prompt suggestions to the user based on the text the user is typing.
- **`ListPicker`**: Used to display a list of items. It is useful especially when asking users to choose between different values.
- **`ExpanderView`**: Used to create elements that can be expanded using a tree structure to display other elements. A good example of this control is the Mail app; it's used to display conversations.
- **`MultiSelectList`**: Similar to a **`ListBox`**, but automatically places check boxes next to each item, allowing users to select multiple items.
- **`PhoneTextBox`**: A special **`TextBox`** control with many built-in features, like support for action icons, placeholders, a character counter, etc.
- **`HubTile`**: Can be used to recreate the Start screen experience offered by Live Tiles inside an application.
- **`CustomMessageBox`**: A special **`MessageBox`** that offers many more options than the standard one, like button customization, custom template support, etc.
- **`Rating`**: Gives users the ability to rate something inside an application. The user experience is similar to the one offered by the Store, where users can vote on an application.
- **`SpeechTextBox`**: Another special **`TextBox`** that supports vocal recognition so that users can dictate text instead of type it.

The Windows Phone Toolkit also includes a Windows Phone frame replacement (the class that manages views and navigation) with built-in support for animations so that you can easily add transition effects when users move from one page of the application to another. Let's dig deeper into this topic.

Page transitions

During this chapter, we've learned how to animate objects that are placed inside a page. Often, one of the easiest ways to improve the look and feel of our application is to add an animation during the transition from one page to another. The Windows Phone Toolkit plays an important role in achieving this result since the standard application frame provided by the SDK, which manages all our application's pages, doesn't support transitions. Instead, the toolkit offers a specific frame class, called **`TransitionFrame`**, which can be used to replace the original frame class, which is called **`PhoneApplicationFrame`**.

The first step is to replace the original frame. You can do this in the **App.xaml.cs** file, which contains a hidden region titled **`Phone application initialization`**. If you expand it, you'll find a method called **`InitializePhoneApplication()`** which, among other things, initializes the application's frame with the following code:

```
RootFrame = new PhoneApplicationFrame();
```

Replacing the original frame is easy. Once you've installed the Windows Phone Toolkit, you can change the initialization of the **`RootFrame`** object by using the **`TransitionFrame`** class, which is part of the **`Microsoft.Phone.Controls`** namespace, as shown in the following sample:

```
RootFrame = new TransitionFrame();
```

Now you're ready to set which animations you want to use in your pages according to the navigation type. Let's start by looking at the following sample code, which should be added in every page you want to animate with a transition. The code has to be placed under the main **`PhoneApplicationPage`** node before the page's layout is defined:

```
<toolkit:TransitionService.NavigationInTransition>
    <toolkit:NavigationInTransition>
        <toolkit:NavigationInTransition.Backward>
            <toolkit:TurnstileTransition Mode="BackwardIn"/>
        </toolkit:NavigationInTransition.Backward>
        <toolkit:NavigationInTransition.Forward>
            <toolkit:TurnstileTransition Mode="ForwardIn"/>
        </toolkit:NavigationInTransition.Forward>
    </toolkit:NavigationInTransition>
</toolkit:TransitionService.NavigationInTransition>

<toolkit:TransitionService.NavigationOutTransition>
    <toolkit:NavigationOutTransition>
        <toolkit:NavigationOutTransition.Backward>
            <toolkit:TurnstileTransition Mode="BackwardOut"/>
        </toolkit:NavigationOutTransition.Backward>
        <toolkit:NavigationOutTransition.Forward>
            <toolkit:TurnstileTransition Mode="ForwardOut"/>
```

```
        </toolkit:NavigationOutTransition.Forward>
    </toolkit:NavigationOutTransition>
</toolkit:TransitionService.NavigationOutTransition>
```

Transitions are added using the **TransitionService**, which is part of the Windows Phone Toolkit (make sure that the **xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls.Toolkit"** namespace is added to your page).

It supports two type of animations, specified using the **NavigationInTransition** property:

- **In** animations, which are applied when users move to the current page.
- **Out** animations, which are applied when users move away from the current page.

For every transition type, you also have the chance to specify two additional conditions:

- The **Backward** property is used to specify the transition to use when users reach the page after pressing the Back button.
- The **Forward** property is used to specify the transition to use when users reach the page through the regular navigation flow.

Finally, you're able to specify which transition you want to use for every scenario. The toolkit offers a list of predefined animations, like **RotateTransition** to apply a rotation effect, **TurnstileTransition** to simulate browsing a book, or **SlideTransition** to apply a slide effect. Every transition offers a **Mode** property, which can be used to customize the effect that is applied. The previous sample shows a **TurnstileTransition** effect applied during every navigation, with a different effect according to the navigation's type (backward or forward).

Customize the turnstile transition

The animation framework can be used not only to apply a transition to the entire page, but also to every control in the page. This scenario is supported by the **TurnstileFeatherTransition** control, which applies a turnstile effect to the controls in the page. You can decide which order will be used to animate and enter the controls in the page with the **FeatheringIndex** property.

The first step is to add a **TransitionService** to your page and define a set of **TurnstileFeatherTransition** animations, as in the following sample:

```
<toolkit:TransitionService.NavigationInTransition>
    <toolkit:NavigationInTransition>
        <toolkit:NavigationInTransition.Backward>
            <toolkit:TurnstileFeatherTransition Mode="BackwardIn"/>
        </toolkit:NavigationInTransition.Backward>
        <toolkit:NavigationInTransition.Forward>
            <toolkit:TurnstileFeatherTransition Mode="ForwardIn"/>
        </toolkit:NavigationInTransition.Forward>
    </toolkit:NavigationInTransition>
</toolkit:TransitionService.NavigationInTransition>
<toolkit:TransitionService.NavigationOutTransition>
    <toolkit:NavigationOutTransition>
```

```
        <toolkit:NavigationOutTransition.Backward>
            <toolkit:TurnstileFeatherTransition Mode="BackwardOut"/>
        </toolkit:NavigationOutTransition.Backward>
        <toolkit:NavigationOutTransition.Forward>
            <toolkit:TurnstileFeatherTransition Mode="ForwardOut"/>
        </toolkit:NavigationOutTransition.Forward>
    </toolkit:NavigationOutTransition>
</toolkit:TransitionService.NavigationOutTransition>
```

Then, you're able to apply the **TurnstileFeatherTransition.FeatheringIndex** property to any control in the page and specify the order in which they will appear, starting with 0 to set the first control that will enter in the page.

```
<StackPanel>
    <TextBlock Text="First Element"
               toolkit:TurnstileFeatherEffect.FeatheringIndex="0"/>
    <TextBlock Text="Second Element"
               toolkit:TurnstileFeatherEffect.FeatheringIndex="1"/>
    <TextBlock Text="Third Element"
               toolkit:TurnstileFeatherEffect.FeatheringIndex="2"/>
</StackPanel>
```

In the previous sample, the three **TextBlock** controls will appear in the page starting with the first one (whose **FeatheringIndex** is equal to 0) and ending with the last one (whose **FeatheringIndex** is equal to 2).

A quick recap

It's been a long journey so far and we've just scratched the surface; covering all the XAML features would require an entire book. In this chapter, we've considered some key concepts used in Windows Phone development:

- We introduced the basic XAML concepts like properties, events, namespaces, and resources.
- We learned how data binding works. It's one of the most powerful XAML features and it's very important to understand it to be productive.
- We've seen some of the basic controls that are included in the SDK and how we can use them to define the user interface of our application.
- We discussed some controls that are specific to the Windows Phone experience, like the **Panorama**, **Pivot**, and **ApplicationBar** controls.

Chapter 3 Core Concepts

Asynchronous programming

Nowadays, asynchronous programming is a required skill for developers. In the past, most applications worked with a synchronous approach: the user started an operation and, until it was completed, the application was frozen and completely unusable.

This behavior isn't acceptable today, especially in the mobile world. Users wouldn't buy a phone that doesn't allow them to answer a call because an application is stuck trying to complete an operation.

There are two approaches in Windows Phone to manage asynchronous programming: **callbacks** and the **async/await** pattern.

Callbacks

In most cases, especially in Windows Phone 8, callbacks have been replaced with asynchronous methods that use the async and await pattern. However, there are some APIs that still use the callback approach, so it's important to learn it.

Callbacks are delegate methods that are called when an asynchronous operation is terminated. This means that the code that triggers the operation and the code that manages the result are stored in two different methods.

Let's look at the following example of the **`WebClient`** class, one of the basic APIs of the framework that performs network operations like downloading and uploading files:

```
private void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += new
DownloadStringCompletedEventHandler(client_DownloadStringCompleted);
    client.DownloadStringAsync(new Uri("http://wp.qmatteoq.com",
UriKind.Absolute));
    Debug.WriteLine("Download has started");
}

void client_DownloadStringCompleted(object sender,
DownloadStringCompletedEventArgs e)
{
    MessageBox.Show(e.Result);
}
```

The file download (in this case, we're downloading the HTML of a webpage) is started when we call the **`DownloadStringAsync()`** method, but we need to subscribe to the **`DownloadStringCompleted`** event, which is triggered when the download is complete, to manage the result. Usually, the event handler (the callback method) has a parameter that contains the result of the request. In the previous sample, it's called **`e`** and its type is **`DownloadStringCompletedEventArgs`**.

Notice the message "Download has started" that is written in the Visual Studio **Output** window. Since the method is asynchronous, the message will be displayed immediately after the download has started since the code will continue to run until the download is completed. Then, the execution moves to the callback method.

Async and await

The callback approach is hard to read. Unlike synchronous code, the flow of execution "jumps" from one method to another, so a developer can't simply read line after line to understand what's going on. In addition, if you need to write a custom method that starts a network operation and returns the downloaded file, you simply can't do it because the logic is split in two different pieces.

The async and await pattern was introduced in C# 5.0 to resolve these issues. The Windows Runtime is heavily based on this approach and most of the APIs we're going to see in this book use it.

When we use the **`async`** and **`await`** keywords, we're going to write sequential code as if it's synchronous—the compiler will execute one row after the other. To better understand what happens under the hood, keep in mind that when we start an asynchronous operation, the compiler sets a bookmark and then quits the method. This way, the UI thread is free and users can keep interacting with the application. When the asynchronous operation is completed, the compiler goes back to the bookmark and continues the execution.

The async and await pattern is based on the **`Task`** class, which is the base return type of an asynchronous method. A method can return:

- **`Task`** if it's a **`void`** method: The compiler should wait for the operation to be completed to execute the next line, but it doesn't expect a value back.
- **`Task<T>`** if the method returns a value. The compiler will wait for the operation to be completed and will return the result (whose type is **`T`**) to the main method, which will be able to perform additional operations on it.

Let's see an example. We're going to add to our project a library called **Async for .NET**, which has been developed by Microsoft and is available on NuGet. Its purpose is to add async and await support to old technologies that don't support it since they're not based on C# 5.0, like Windows Phone 7 or Silverlight. It's also useful in Windows Phone 8 applications, since it adds asynchronous methods to APIs that are still based on the callback pattern.

One of these APIs is the **WebClient** class we've previously seen. By installing this library we can use the **DownloadStringTaskAsync()** method which supports the new pattern. This method's return type is **Task<string>**. This means that the operation is asynchronous and it will return a string.

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    string result = await
client.DownloadStringTaskAsync("http://wp.qmatteoq.com");
    MessageBox.Show(result);
}
```

First, let's note how to use this new syntax. Any method that contains an asynchronous method has to be marked with the **async** keyword. In this case, it's an event handler that is invoked when the user presses a button.

Next, we need to add the **await** keyword as a prefix of the asynchronous method. In our example, we've placed it before the method **DownloadStringTaskAsync()**. This keyword tells the compiler to wait until the method is completed before moving on.

When the asynchronous method is started, you can assume that the compiler has set a bookmark and then quit the method because the application is still responsive. When the download is completed, the compiler returns to the bookmark and continues the execution. The download's result is stored in the **result** variable, whose value is displayed on the screen using a **MessageBox**.

As you can see, even if the code under the hood is asynchronous and doesn't freeze the UI, it appears synchronous: the code is executed one line at a time and the **MessageBox** isn't displayed until the **DownloadStringTaskAsync()** method has finished its job.

The dispatcher

Sometimes, especially when you have to deal with callback methods, the operations are executed in a background thread, which is different from the thread that manages the UI. This approach is very helpful in keeping the UI thread as free as possible so that the interface is always fast and responsive.

Sometimes a background thread may need to interact with the user interface. For example, let's say a background thread has completed its job and needs to display the result in a **TextBlock** control placed in the page. If you try to do it, an **UnauthorizedAccessException** error with the message "Invalid cross-thread access" will be displayed.

The reason for this error is that a background thread isn't allowed to access the controls in a page since they are managed by a different thread. The solution is to use a framework's **Dispatcher** class. Its purpose is to dispatch operations to the UI thread from another thread. This way, you can interact with XAML controls without issues because the thread is the same. The following sample illustrates how to use it:

```
Dispatcher.BeginInvoke(() =>
{
    txtResult.Text = "This is the result";
})
```

You simply have to pass the **Action** that needs to be executed on the UI thread as a parameter of the **BeginInvoke()** method of the **Dispatcher** class. In the previous sample, the **Action** is defined using an **anonymous method**.

Tip: Anonymous methods are called such because they don't have a specific name—they simply define the operations to be performed.

Dispatcher is not needed when you work with the async and await pattern because it makes sure that even if the operation is executed in one of the free threads available in the thread pool, the result is always returned in the UI thread.

Navigation

Another important concept to learn when you develop a Windows Phone application is navigation. In most cases, your application will be made of different pages, so you need a way to move from one page to another.

The framework offers a built-in navigation system, which can be managed using the **NavigationService** class.

Navigating to a page is easy. Just call the **Navigate()** method, passing a **Uri**, which is the path of the page starting from your project's root, as its parameter.

```
private void OnGoToPage2Clicked(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/Pages/Page2.xaml",
UriKind.Relative));
}
```

The navigation system provides, in every page, two events that can be managed to intercept the navigation events: **OnNavigatedTo()** is triggered when you move from another page to the current page; **OnNavigateFrom()** is triggered when you move from the current page to another page. Another important page event that can be subscribed is **Loaded**, which is triggered when the page is completely loaded. It's commonly used to load the data that needs to be displayed on the page.

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        Loaded += MainPage_Loaded;
    }

    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        Debug.WriteLine("Page is loaded");
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        base.OnNavigatedTo(e);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        base.OnNavigatedFrom(e);
    }
}
```

Passing parameters to another page

A common requirement when dealing with navigation is to pass a parameter to another page, for example, a master/detail scenario where you select a list item in the master page and display its details in a new page.

The navigation framework offers a built-in way to pass simple parameters like strings and numbers that is inherited directly from the web world: query string parameters. They're added directly to the URL of the page where you want to redirect the user, as shown in the following sample:

You'll be able to retrieve the parameter in the landing page by using the **OnNavigatedTo** event and the **NavigationContext** class, as in the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("id"))
    {
        int id = int.Parse(NavigationContext.QueryString["id"]);
        MessageBox.Show("The id is " + id);
    }
}
```

The **NavigationContext** class offers a collection called **QueryString**, which contains all the available parameters. You'll be able to get the value by using its key (which is the parameter's name).

In some cases, simple parameters aren't enough. Sometimes you need to pass complex objects to another page. The master/detail scenario is, also in this case, a good example. In many situations, to properly populate the detail page, you need access to the whole object that has been selected in the list. A good approach to this is to pass the detail page a plain parameter that can be used to identify the selected item, like a unique identifier. Then, in the **OnNavigatedTo** event handler of the detail page, load the item with the received identifier from your data collection (which can be a database or a remote service).

Manipulating the navigation stack

Every application has its own navigation stack, which is the list of pages that the user has moved through while using the application. Every time the user navigates to a new page, it's added to the stack. When the user goes back, the last page from the stack is removed. When one page is left in the stack and the user presses the Back button, the application is closed.

The **NavigationService** class offers a few ways to manipulate the stack. In the beginning, there's a specific property called **BackStack** which is a collection of all the pages in the stack. When we discuss Fast App Resume later in this chapter, we'll see one of the available methods to manipulate the stack: **RemoveBackEntry()**, which removes the last page from the stack.

Another important method is **GoBack()**, which redirects the user to the previous page in the stack. Its purpose is to avoid creating circular navigation issues in your application.

Figure 16 illustrates an example of circular navigation and how to correct it. Let's say that you have an application made by two pages, a main page and a settings page. You choose to use the **Navigate()** method of the **NavigateService** class for every navigation. The user navigates from the main page to the settings page, and again to the main page. The issue here is that every time you call the **Navigate()** method, you add a new page to the navigation stack. When the user is on the main page and presses the Back button, he or she expects to quit the app. Instead, the second time, the user will be redirected to the Settings page since the previous navigation added the page to the stack.

The correct approach is to use the **`GoBack()`** method. Instead of adding a new page to the stack, this method simply redirects the user to the previous one. The result will be the same—the user, from the settings page, is taken back to the main page—but since there are no pages added to the stack, the application will behave as expected. If the user presses the Back button, the app will be closed.

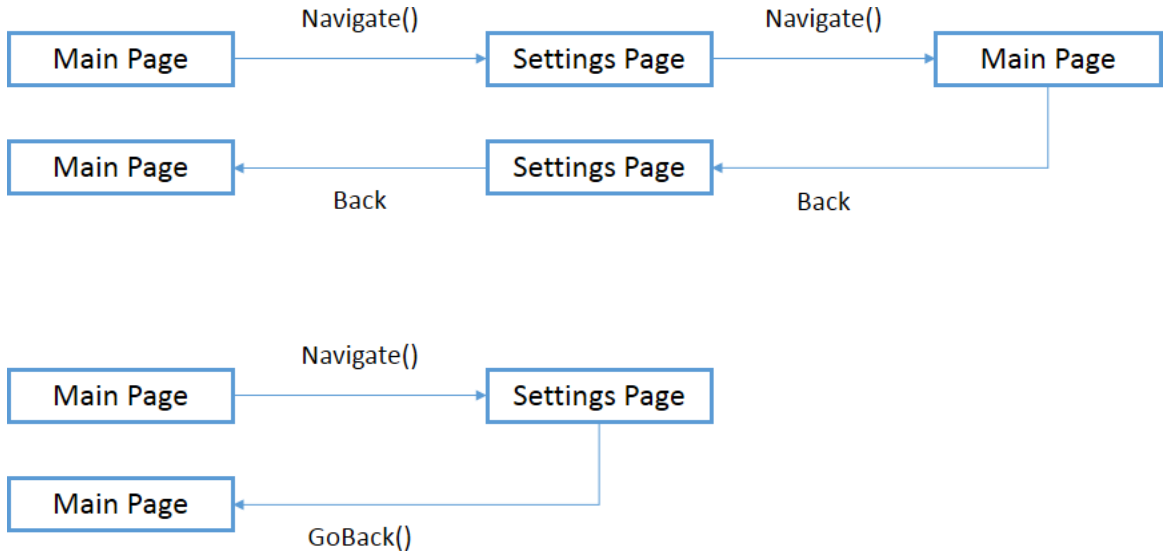

Figure 16: Circular Navigation (Top) and Correct Navigation (Bottom)

Intercepting the navigation: the UriMapper class

Later in the book we'll deal with different situations where the application is opened with a special **`Uri`**, such as using the Speech API or data sharing (which we'll talk about in Chapter 7).

In these cases, we're going to use a navigation feature called **`UriMapper`**, which is a class that can act as a middle man in every navigation operation. When the user navigates from one page to another (the starting page can be also be the phone's Home screen), the class is able to intercept it and redirect the user to another page if needed.

To create a **`UriMapper`**-based class, simply add a new class to your project. It has to inherit from the **`UriMapperBase`** class. It will require you to implement the **`MapUri()`** method, which is invoked during navigation as shown in the following sample:

```
public class UriMapper: UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        string tempUri = HttpUtility.UrlDecode(uri.ToString());
        if (tempUri.Contains("/FileTypeAssociation"))
        {
            //Manage the selected file.
            return new Uri("/Pages/FilePage.xaml", UriKind.Relative);
        }
```

```
        else
        {
            return uri;
        }
    }
}
```

The **MapUri()** method takes in the source **Uri** and needs to return the new **Uri** to redirect the user. In the previous sample, we checked whether the source **Uri** contained a specific string (in this case, it's one that belongs to the data sharing scenario we'll see in Chapter 7). In the affirmative case, we redirect the user to a specific page of the application that is able to manage the scenario; otherwise, we don't change the navigation flow by returning the original **Uri**.

After you've created a **UriMapper** class, you'll have to assign it to the **UriMapper** property of the **RootFrame** object that is declared in the **App.xaml.cs** file. You'll have to expand the region called **Phone application initialization** (which is usually collapsed), and change the **InitializePhoneApplication()** method like in the following sample:

```
private void InitializePhoneApplication()
{
    if (phoneApplicationInitialized)
        return;

    // Create the frame but don't set it as RootVisual yet; this allows the
    // splash screen to remain active until the application is ready to
render.
    RootFrame = new PhoneApplicationFrame();
    RootFrame.UriMapper = new UriMapper();
    RootFrame.Navigated += CompleteInitializePhoneApplication;

    // Handle navigation failures.
    RootFrame.NavigationFailed += RootFrame_NavigationFailed;

    // Handle reset requests for clearing the backstack.
    RootFrame.Navigated += CheckForResetNavigation;

    // Ensure we don't initialize again.
    phoneApplicationInitialized = true;
}
```

The application life cycle

Mobile applications have a different life cycle than traditional desktop applications due to their different requirements, like power management and performance optimization. The traditional multitasking approach that you experience in Windows doesn't fit well to the mobile experience: the necessary power would drain the battery very quickly. Plus, if you open too many applications, you can suffer severe performance issues that will degrade the user experience.

The following figure explains the life cycle Microsoft has introduced for Windows Phone applications to satisfy the required conditions:

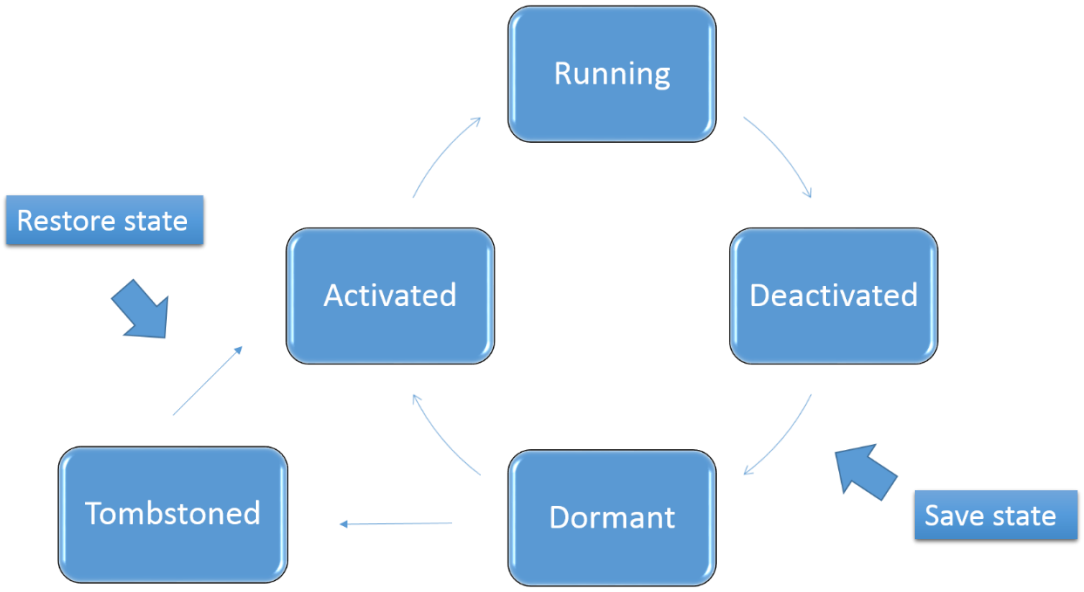

Figure 17: Application Life Cycle

When an application is running, it can be suspended at any time. For example, the user can press the Start button or tap a notification to open another application. In this case, the application is "frozen" (the **dormant** state); it's preserved in memory, but all the running threads and operations are stopped. In this status, the application is using system memory, but the CPU is free. This way, other applications can use the available resources without suffering performance issues.

When the user decides to go back to the application (by using the Back button or the task switcher), its instance is restored from memory and the previous threads and operations are restarted.

If users plays with their device for a long time, too many applications may be opened and there won't be enough memory left to keep other applications suspended. In this case, the operating system can start to "kill" longer-running applications to free some memory for the newly opened apps—the older apps are moved into a **tombstoned** state. To give users the same smooth experience as when apps are simply suspended, developers have the ability to save the state of the app when it is tombstoned and restore it later. When users open a tombstoned app, they will find the application in the exact previous state, as if it's never been closed.

The tombstoning experience was the only one available in Windows Phone 7, but it often led to a bad user experience. Because the app was killed, the application always needed to start from scratch, causing longer loading times. In Windows Phone 7.5, Microsoft introduced the **Fast Application Switching** concept that has been translated into the current life cycle, where apps are put in a dormant state and are killed only if needed.

Based on this information, we can understand that applications aren't able to run in the background—when an application is suspended, all of its running operations are canceled. In Chapter 9, we'll look at available techniques for executing some operations even when the app is not running.

As a developer, you are not notified by the operating system when an application is tombstoned. This is why, in the schema shown in Figure 17, you can see that the save state operation is made every time the application is suspended. You'll always have to deal with it, since you won't know in advance whether your app will be tombstoned.

Instead, the restore operation is made only if the application has been tombstoned. When the app is in a dormant state, the entire process is stored in memory, so there's no data loss when the app is restored.

Since tombstoning isn't deterministic, Visual Studio offers a way to test it. By default, applications are moved to the dormant state when suspended. You can change this behavior by right-clicking on your project and choosing **Properties**. In the **Debug** section you'll find the **Tombstone upon deactivation while debugging** option. When it's selected, applications will always be tombstoned when they're suspended, so you can test that everything works fine and no data is lost in this scenario.

Controlling the life cycle

You can control the application life cycle thanks to the **`PhoneApplicationService`** class, which is declared in the **App.xaml** file as a lifetime object.

```
<Application.ApplicationLifetimeObjects>
    <shell:PhoneApplicationService
        Launching="Application_Launching" Closing="Application_Closing"
        Activated="Application_Activated"
Deactivated="Application_Deactivated"/>
</Application.ApplicationLifetimeObjects>
```

As you can see, the **`PhoneApplicationService`** class is registered to manage four different events that are available in the **App.xaml.cs** file:

- **`Application_Launching`** is invoked when the app starts for the first time.
- **`Application_Closing`** is invoked when the app is completely closed. It only happens when the user presses the Back button in the application's main page.
- **`Application_Activated`** is invoked when the app is resumed from a suspended state.
- **`Application_Deactivated`** is invoked when the app is suspended.

Remember the “save state” and “restore state” operations in Figure 17? Usually, they’re managed in these events. State is saved when the **Application_Deactivated** event occurs, and it’s restored when the **Application_Activated** event occurs.

Specifically, one of the parameters of the **Application_Activated** event handler contains important information—it shows whether the application comes from a dormant or tombstoned state as demonstrated in the following sample:

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    if (e.IsApplicationInstancePreserved)
    {
        Debug.WriteLine("The app was dormant");
    }
    else
    {
        Debug.WriteLine("The app was tombstoned");
    }
}
```

The key is the **IsApplicationInstancePreserved** property of the **ActivatedEventArgs** object, which is a Boolean. If it’s true, the app was in a dormant state; otherwise, it was killed, and it’s time to restore the data we’ve previously saved.

The **PhoneApplicationService** class can also be useful to store the state of your application when it’s suspended. In fact, it offers a property called **State** that is a collection whose type is **Dictionary<string, object>**. It can be used to save any object that is identified by a key.

Typically, you’re going to save data in the **State** property when the app is suspended, and restore the data when the app is resumed from a tombstoned state.

However, there are some important things to keep in mind:

- It’s best to save data as soon as possible and not only when the **Application_Deactivated** event is raised. Unexpected bugs or other errors can cause users to lose data during the execution.
- The content of the **State** property is kept in memory only if the app is resumed; if the user decides to open the application from scratch (for example, by tapping on its Tile instead of using the Back button), the collection is erased and the saved data will be lost. If you need to persist the data across multiple executions, it’s better to save to the local storage, which we’ll cover in Chapter 4.

Fast App Resume

The life cycle previously described has a flaw: users can resume a suspended application only by using the Back button or the task switcher. If they pin a Tile for the application on the Start screen and tap on it, a new instance of the app will be opened and the suspended one will be terminated.

Windows Phone 8 has introduced a new feature called **Fast App Resume**, which can be activated to address the flaw previously described. Regardless of the opening point, if there's a suspended instance, it will always be used.

Supporting Fast App Resume requires a bit of work because, by default, the Windows Phone template is made to use the old life cycle, even when the feature is activated.

The first step is to modify the manifest file, and unfortunately, it's one of the situations where the visual editor isn't helpful. In fact, we'll need to manually edit the file. To do this, simply right-click the **WMAppManifest.xml** file inside the **Properties** folder and choose the **View code** option.

You'll find a node called **`DefaultTask`** that tells the application which page to load first when the app starts. You'll need to add an attribute called **`ActivationPolicy`** to set its value to **`Resume`**, like the following sample:

```
<Tasks>
  <DefaultTask  Name ="_default" NavigationPage="MainPage.xaml"
ActivationPolicy="Resume" />
</Tasks>
```

What happens when users tap on the main Tile of the application and the Fast App Resume has been enabled? The operating system triggers two navigations: the first one is toward the last opened page in the application, and the second one is toward the main page of the application since the application Tile is associated with it through the default navigation Uri.

At this point, you have two choices:

- You can keep using the old approach. In this case, you have to remove the last visited page from the back stack; otherwise you'll break the navigation system. In fact, when users are on the main page of the application and press the Back button, they expect to quit the application, not go back to an old page.
- You can support Fast App Resume. In this case, you have to stop the navigation to the main application page so that users are taken to the last visited page of the application.

The default Windows Phone 8 templates contain an implementation of the first approach. Open the **App.xaml.cs** file and look for a method called **`InitializePhoneApplication()`**, which takes care of initializing the frame that manages the various pages and the navigation system. You'll find that the application subscribes to an event called **`Navigated`** of the **`RootFrame`** class, which is triggered every time the user has moved from one page of the application to another.

The method that is assigned as the handler of this event is called **`ClearBackafterReset()`**, which has the following definition:

```
private void ClearBackStackAfterReset(object sender, NavigationEventArgs e)
{
    // Unregister the event so it doesn't get called again.
    RootFrame.Navigated -= ClearBackStackAfterReset;
```

```
    // Only clear the stack for 'new' (forward) and 'refresh' navigations.
    if (e.NavigationMode != NavigationMode.New && e.NavigationMode !=
NavigationMode.Refresh)
        return;

    // For UI consistency, clear the entire page stack.
    while (RootFrame.RemoveBackEntry() != null)
    {
        ; // Do nothing.
    }
}
```

This method does exactly what was previously described: the navigation to the main page is not canceled but, using the **RemoveBackEntry()** method of the **RootFrame** class, the page stack of the application is cleaned.

Take note of the **NavigationMode** property: it specifies the status of the navigation, like **New** when users navigate to a new page, or **Back** when users go back to a previous page. The key status to manage Fast App Resume is **Reset**, which is set when users are navigating to the main application page but an old instance of the app is being used.

We're going to use the **NavigationMode** property together with some changes to the original code. The first step is to change the **Navigated** event handler to simply store, in a global Boolean property of the **App.xaml.cs** class, whether the **NavigationMode** is reset. We're going to use this information in another event handler. In fact, we need to subscribe to the **Navigating** event of the **RootFrame** class. This event is similar to the **Navigated** one, except that it is triggered before the navigation starts and not after. This difference is important for our purpose, since we have the chance to cancel the navigation operation before it's executed.

Here is what we do in the **Navigating** event handler:

```
void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
    if (reset && e.IsCancelable && e.Uri.OriginalString ==
"/MainPage.xaml")
    {
        e.Cancel = true;
        reset = false;
    }
}
```

In this event, we wait for a specific condition to happen: the **NavigationMode** is **Reset** and navigation to the main page is triggered. This situation occurs when users tap on the main Tile and an instance of the app is already in memory. The first navigation redirects to the last visited page, and the second one (the **Reset** mode) redirects to the main page. It's this scenario that we need to manage. By setting the **Cancel** property of the method's parameter, we abort the navigation to the main page and leave the user on the last-visited page of the app. The experience is exactly the same as when the user returns to the application using the Back button.

Here is how the complete code required to implement Fast App Resume looks in the **App.xaml.cs** file:

```
private bool reset;

// Do not add any additional code to this method.
private void InitializePhoneApplication()
{
    if (phoneApplicationInitialized)
        return;

    // Create the frame but don't set it as RootVisual yet; this allows the
    // splash screen to remain active until the application is ready to
render.
    RootFrame = new PhoneApplicationFrame();
    RootFrame.Navigated += CompleteInitializePhoneApplication;

    // Handle navigation failures.
    RootFrame.NavigationFailed += RootFrame_NavigationFailed;

    // Handle reset requests for clearing the backstack.
    RootFrame.Navigated += RootFrame_Navigated;

    RootFrame.Navigating += RootFrame_Navigating;

    // Ensure we don't initialize again.
    phoneApplicationInitialized = true;
}

void RootFrame_Navigated(object sender, NavigationEventArgs e)
{
    reset = e.NavigationMode == NavigationMode.Reset;
}

void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
    if (reset && e.IsCancelable && e.Uri.OriginalString ==
"/MainPage.xaml")
    {
        e.Cancel = true;
        reset = false;
    }
}
```

You may be wondering why all this effort is needed to support Fast App Resume. Why isn't it automatically implemented by the operating system? The answer is that Fast App Resume isn't suitable for every application—you have to be very careful when you decide to support it, because you can ruin the user experience if you don't implement it well.

For example, if your users spend most of the time in the main page of your application (for example, a social networking app), they probably prefer to go straight to the Home page when they open the application, instead of resuming what they were doing before.

Here are two tips for improving the Fast App Resume experience:

- If your application has many pages and a complex navigation hierarchy, add to the inner pages a quick way to go back to the home page (like a button in the Application Bar).
- Add a timing condition. If the application hasn't been used recently, disable Fast App Resume and redirect the user to the home page.

Manage orientation

Typically, a Windows Phone device is used in portrait mode, but there are some scenarios where users can benefit from landscape mode. For example, if your application offers a way to insert long text, it can be more convenient for users to type using the landscape keyboard. Alternatively, in other scenarios, you can use the landscape mode to provide the content with a different layout.

By default, every new page added to a Windows Phone is displayed in portrait mode and does not support the landscape mode. When you rotate the phone, nothing happens. The orientation behavior is controlled by two attributes of the **PhoneApplicationPage** node in the XAML:

```
<phone:PhoneApplicationPage
    x:Class="Webinar.Rest.MainPage"
    SupportedOrientations="Portrait" Orientation="Portrait">
```

- The **SupportedOrientations** property defines which orientations are supported by the page. To support both orientations, you need to set the value to **PortraitOrLandscape**.
- The **Orientation** property defines the default orientation of the page.

Once you've set the **SupportedOrientations** property to **PortraitOrLandscape**, the page's layout will automatically adapt to landscape mode when the phone is rotated.

If you want further control (for example, because you want to deeply change the page's layout between portrait and landscape modes), the APIs offer a page event called **OrientationChanged** that is raised every time the phone is rotated from portrait to landscape mode and vice versa. The event is exposed directly by the **PhoneApplicationPage** class, so you can subscribe it in the XAML, as in the following sample:

```
<phone:PhoneApplicationPage
    x:Class="Webinar.Rest.MainPage"
    SupportedOrientations="Portrait" Orientation="Portrait"
    OrientationChanged="MainPage_OnOrientationChanged">
```

The event handler that will be generated in the code will return the information about the new orientation. You can use it to move or change the aspect of the controls placed in the page. The following sample shows how to set a different **Height** of a **Button** control, according to the detected orientation:

```
private void MainPage_OnOrientationChanged(object sender,
OrientationChangedEventArgs e)
{
    if (e.Orientation == PageOrientation.PortraitUp || e.Orientation ==
PageOrientation.PortraitDown)
    {
        Button1.Height = 100;
    }
    else
    {
        if (e.Orientation == PageOrientation.LandscapeLeft || e.Orientation
== PageOrientation.LandscapeRight)
        {
            Button1.Height = 500;
        }
    }
}
```

A quick recap

If you're new to Windows Phone development, this chapter is very important, because we've learned some basic core concepts that we're going to use again in later chapters.

Here is a quick recap:

- Asynchronous programming is must-have knowledge for mobile developers. We've learned how callbacks and the new async and await pattern will help you keep the user interface fast and responsive.
- It's highly unlikely that your application is made of just one page, so we've learned how to support navigation between different pages and how to correctly manage the navigation flow.
- Application life cycle is another key concept. You have to understand how a Windows Phone application works under the hood to correctly manage all the multitasking scenarios. We've talked about Fast Application Switching, an improvement introduced in Windows Phone 7.5 to make the suspending and resuming process faster, and Fast App Resume, a new Window Phone 8 feature that improves the resuming experience.

Chapter 4 Data Access: Storage

Local storage

The Internet plays an important role in mobile applications. Most Windows Phone applications available in the Store make use of the network connection offered by every device. However, relying only on the network connection can be a mistake; users can find themselves in situations where no connection is available. In addition, data plans are often limited, so the fewer network operations we do, the better the user experience is.

Windows Phone offers a special way to store local data called **isolated storage**. It works like a regular file system, so you can create folders and files as on a computer hard drive. The difference is that the storage is isolated—only your applications can use it. No other applications can access your storage, and users are not able to see it when they connect their phone to the computer. Moreover, as a security measure, the isolated storage is the only storage that the application can use. You're not allowed to access the operating system folders or write data in the application's folder.

Local storage is one of the features which offers duplicated APIs—the old Silverlight ones based on the **`IsolatedStorageFile`** class and the new Windows Runtime ones based on the **`LocalFolder`** class. As mentioned in the beginning of the book, we're going to focus on the Windows Runtime APIs.

Working with folders

The base class that identifies a folder in the local storage is called **`StorageFolder`**. Even the root of the storage (which can be accessed using the **`ApplicationData.Current.LocalStorage`** class that is part of the **`Windows.Storage`** namespace) is a **`StorageFolder`** object.

This class exposes different asynchronous methods to interact with the current folder, such as:

- **`CreateFolderAsync()`** to create a new folder in the current path.
- **`GetFolderAsync()`** to get a reference to a subfolder of the current path.
- **`GetFoldersAsync()`** to get the list of folders available in the current path.
- **`DeleteAsync()`** to delete the current folder.
- **`RenameAsync()`** to rename a folder.

In the following sample, you can see how to create a folder in the local storage's root:

```
private async void OnCreateFolderClicked(object sender, RoutedEventArgs e)
{
    await
```

```
ApplicationData.Current.LocalFolder.CreateFolderAsync("myFolder");
}
```

Unfortunately, the APIs don't have a method to check if a folder already exists. The simplest solution is to try to open the folder using the **`GetFolderAsync()`** method and intercept the **`FileNotFoundException`** error that is raised if the folder doesn't exist, as shown in the following sample:

```
private async void OnOpenFileClicked(object sender, RoutedEventArgs e)
{
    StorageFolder folder;
    try
    {
        folder = await
ApplicationData.Current.LocalFolder.GetFolderAsync("myFolder");
    }
    catch (FileNotFoundException exc)
    {
        folder = null;
    }

    if (folder == null)
    {
        MessageBox.Show("The folder doesn't exist");
    }
}
```

Working with files

Files, instead, are identified by the **`StorageFile`** class, which similarly offers methods to interact with files:

- **`DeleteAsync()`** to delete a file.
- **`RenameAsync()`** to rename a file.
- **`CopyAsync()`** to copy a file from one location to another.
- **`MoveAsync()`** to move a file from one location to another.

The starting point to manipulate a file is the **`StorageFolder`** class we've previously discussed, since it offers methods to open an existing file (**`GetFileAsync()`**) or to create a new one in the current folder (**`CreateFileAsync()`**).

Let's examine the two most common operations: writing content to a file and reading content from a file.

How to create a file

As already mentioned, the first step to create a file is to use the **CreateFile()** method on a **StorageFolder** object. The following sample shows how to create a new file called **file.txt** in the local storage's root:

```
private async void OnCreateFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("file.txt",
CreationCollisionOption.ReplaceExisting);
}
```

You can also pass the optional parameter **CreationCollisionOption** to the method to define the behavior to use in case a file with the same name already exists. In the previous sample, the **ReplaceExisting** value is used to overwrite the existing file.

Now that you have a file reference thanks to the **StorageFile** object, you are able to work with it using the **OpenAsync()** method. This method returns the file stream, which you can use to write and read content.

The following sample shows how to write text inside the file:

```
private async void OnCreateFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("file.txt",
CreationCollisionOption.ReplaceExisting);
    IRandomAccessStream randomAccessStream = await
file.OpenAsync(FileAccessMode.ReadWrite);

    using (DataWriter writer = new
DataWriter(randomAccessStream.GetOutputStreamAt(0)))
    {
        writer.WriteString("Sample text");
        await writer.StoreAsync();    }
}
```

The key is the **DataWriter** class, which is a Windows Runtime class that can be used to easily write data to a file. We simply have to create a new **DataWriter** object, passing as a parameter the output stream of the file we get using the **GetOuputStreamAt()** method on the stream returned by the **OpenAsync()** method.

The **DataWriter** class offers many methods to write different data types, like **WriteDouble()** for decimal numbers, **WriteDateTime()** for dates, and **WriteBytes()** for binary data. In the sample we write text using the **WriteString()** method, and then we call the **StoreAsync()** and **FlushAsync()** methods to finalize the writing operation.

Note: The `using` statement can be used with classes that support the IDisposable interface. They are typically objects that lock a resource until the operation is finished, like in the previous sample. Until the writing operation is finished, no other methods can access the file. With the `using` statement, we make sure that the lock is released when the operation is completed.

How to read a file

The operation to read a file is not very different from the writing one. In this case, we also need to get the file stream using the **OpenFile()** method. The difference is that, instead of using the **DataWriter** class, we're going to use the **DataReader** class, which does the opposite operation. Look at the following sample code:

```
private async void OnReadFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("file.txt");
    IRandomAccessStream randomAccessStream = await
file.OpenAsync(FileAccessMode.Read);

    using (DataReader reader = new
DataReader(randomAccessStream.GetInputStreamAt(0)))
    {
        uint bytesLoaded = await reader.LoadAsync((uint)
randomAccessStream.Size);
        string readString = reader.ReadString(bytesLoaded);
        MessageBox.Show(readString);
    }
}
```

In this case, instead of the **CreateFileAsync()** method, we use the **GetFileAsync()** method, which can be used to get a reference to an already existing file. Then, we start the reading procedure using the **DataReader** class, this time using the input stream that we get using the **GetInputStreamAt()** method.

Like the **DataWriter** class, **DataReader** also offers many methods to read different data types, like **ReadDouble()**, **ReadDateTime()**, and **ReadBytes()**. In this case, we read the text we've previously written by using the **ReadString()** method, which requires the size of the file as its parameter.

A special folder: InstalledLocation

The local storage is the only storage we can use to write our application's data, but in some cases, we may need to include in our project some existing files that need to be processed by the application.

The Windows Runtime offers an API to provide access to the folder where the application is installed and where all the files that are part of your Visual Studio project are copied. It's called **`Package.Current.InstalledLocation`**, and it's part of the **`Windows.ApplicationModel`** namespace.

The **`InstalledLocation`**'s type is **`StorageFolder`**, like the folders in local storage, so you can use the same methods to work with files and folders. Keep in mind that you won't be able to write data, but only read it.

In the following sample, we copy a file from the application's folder to the local storage so that we gain write access.

```
private async void OnCopyFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
Package.Current.InstalledLocation.GetFileAsync("file.xml");
    await file.CopyAsync(ApplicationData.Current.LocalFolder);
}
```

Note: During development you may notice that you'll be able to execute write operations in the application's folder. Don't count on it—during the certification process, the app is locked, so when the app is distributed on the Windows Phone Store, the write access is revoked and you'll start getting exceptions.

Manage settings

One common scenario in mobile development is the need to store settings. Many applications offer a Settings page where users can customize different options.

To allow developers to quickly accomplish this task, the SDK includes a class called **`IsolatedStorageSettings`**, which offers a dictionary called **`ApplicationSettings`** that you can use to store settings.

Note: The `IsolatedStorageSettings` class is part of the old storage APIs; the Windows Runtime offers a new API to manage settings but, unfortunately, it isn't available in Windows Phone.

Using the **ApplicationSettings** property is very simple: its type is **Dictionary<string, object>** and it can be used to store any object.

In the following sample, you can see two event handlers: the first one saves an object in the settings, while the second one retrieves it.

```
private void OnSaveSettingsClicked(object sender, RoutedEventArgs e)
{
    IsolatedStorageSettings settings =
IsolatedStorageSettings.ApplicationSettings;
    settings.Add("name", "Matteo");
    settings.Save();
}

private void OnReadSettingsClicked(object sender, RoutedEventArgs e)
{
    IsolatedStorageSettings settings =
IsolatedStorageSettings.ApplicationSettings;
    if (settings.Contains("name"))
    {
        MessageBox.Show(settings["name"].ToString());
    }
}
```

The only thing to highlight is the **Save()** method, which you need to call every time you want to persist the changes you've made. Except for this, it works like a regular **Dictionary** collection.

Note: Under the hood, settings are stored in an XML file. The API automatically takes care of serializing and deserializing the object you save. We'll talk more about serialization later in this chapter.

Debugging the local storage

A common requirement for a developer working with local storage is the ability to see which files and folders are actually stored. Since the storage is isolated, developers can't simply connect the phone to a computer and explore it.

The best way to view an application's local storage is by using a third-party tool available on CodePlex called **Windows Phone Power Tools**, which offers a visual interface for exploring an application's local storage.

The tool is easy to use. After you've installed it, you'll be able to connect to a device or to one of the available emulators. Then, in the **isolated storage** section, you'll see a list of all the applications that have been side-loaded from Visual Studio. Each one will be identified by its application ID (which is a GUID). Like a regular file explorer, you can expand the tree structure and analyze the storage's content. You'll be able to save files from the device to your PC, copy files from your PC to the application storage, and even delete items.

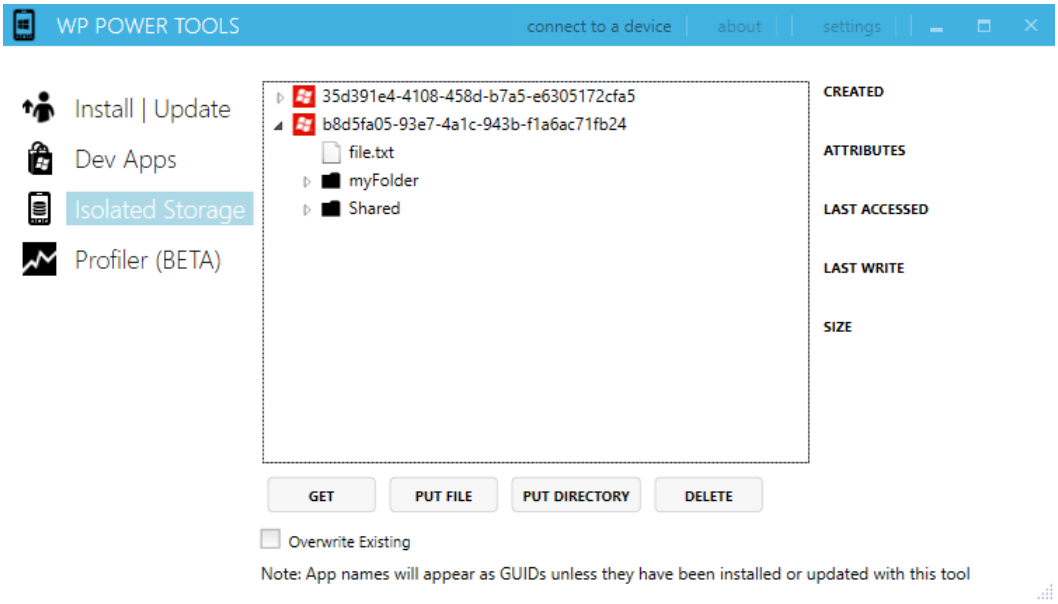

Figure 18: Local Storage of a Windows Phone Application

Storing techniques

In the previous section, we discussed the basic APIs available to store files and folders in your application. In this section, we'll go deeper to see the best ways to store your application's data, so that it can be maintained across different applications.

Serialization and deserialization

Serialization is the simplest way to store an application's data in the local storage. It's the process that converts complex objects into plain text so that they can be stored in a text file, using XML or JSON as output. Deserialization is the opposite process; the plain text is converted back into objects so that they can be used by the application.

In a Windows Phone application that uses these techniques, serialization is typically applied every time the application's data is changed (when a new item is added, edited, or removed) to minimize the risk of losing data if something happens, like an unexpected crash or a suspension. Deserialization, instead, is usually applied when the application starts for the first time.

Serialization is very simple to use, but its usage should be limited to applications that work with small amounts of data, since everything is kept in memory during the execution. Moreover, it best suits scenarios where the data to track is simple. If you have to deal with many relationships, databases are probably a better solution (we'll talk more about this later in the chapter).

In the following samples, we're going to use the same **`Person`** class we used in previous chapters.

```
public class Person
{
    public string Name { get; set; }
    public string Surname { get; set; }
}
```

We assume that you will have a collection of **`Person`** objects, which represents your local data:

```
List<Person> people = new List<Person>
                          {
                              new Person
                                  {
                                      Name = "Matteo",
                                      Surname = "Pagani"
                                  },
                              new Person
                                  {
                                      Name = "John",
                                      Surname = "Doe"
                                  }
                          };
```

Serialization

To serialize our application's data we're going to use the local storage APIs we learned about in the previous section. We'll use the **`CreateFile()`** method again, as shown in the following sample:

```
private async void OnSerializeClicked(object sender, RoutedEventArgs e)
{
    DataContractSerializer serializer = new
DataContractSerializer(typeof(List<Person>));

    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("people.xml");
```

```
    IRandomAccessStream randomAccessStream = await
file.OpenAsync(FileAccessMode.ReadWrite);

    using (Stream stream = randomAccessStream.AsStreamForWrite())
    {
        serializer.WriteObject(stream, people);
        await stream.FlushAsync();
    }
}
```

The **DataContractSerializer** class (which is part of the **System.Runtime.Serialization** namespace) takes care of managing the serialization process. When we create a new instance, we need to specify which data type we're going to serialize (in the previous sample, it's **List<Person>**). Next, we create a new file in the local storage and get the stream needed to write the data. The serialization operation is made by calling the **WriteObject()** method of the **DataContractSerializer** class, which requires as parameters the stream location in which to write the data and the object to serialize. In this example, it's the collection of **Person** objects we've previously defined.

If you take a look at the storage content using the Windows Phone Power Tools, you'll find a **people.xml** file, which contains an XML representation of your data:

```
<ArrayOfPerson xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Storage.Classes">
  <Person>
    <Name>Matteo</Name>
    <Surname>Pagani</Surname>
  </Person>
  <Person>
    <Name>John</Name>
    <Surname>Doe</Surname>
  </Person>
</ArrayOfPerson>
```

Tip: The `DataContractSerializer` class uses XML as its output format. If you want to use JSON instead, you'll have to use the `DataContractJsonSerializer` class, which works in the same way.

Deserialization

The deserialization process is very similar and involves, again, the storage APIs to read the file's content and the **DataContractSerializer** class. The following sample shows how to deserialize the data we serialized in the previous section:

```
private async void OnDeserializeClicked(object sender, RoutedEventArgs e)
{
```

```
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("people.xml");
    DataContractSerializer serializer = new
DataContractSerializer(typeof(List<Person>));

    IRandomAccessStream randomAccessStream = await
file.OpenAsync(FileAccessMode.Read);

    using (Stream stream = randomAccessStream.AsStreamForRead())
    {
        List<Person> people = serializer.ReadObject(stream) as
List<Person>;
    }
}
```

The only differences are:

- We get a stream to read by using the **`AsStreamForRead()`** method.
- We use the **`ReadObject()`** method of the **`DataContractSerializer`** class to deserialize the file's content, which takes the file stream as its input parameter. It's important to note that the method always returns a generic object, so you'll always have to cast it to your real data type (in the sample, we cast it as **`List<Person>`**).

Using databases: SQL CE

When you develop complex applications, you probably have to deal with complex data. Databases are a good solution to manage this scenario because they support relationships, and because the entire dataset is not kept in memory, only the needed items are.

SQL CE is the database solution that was introduced in Windows Phone 7.5. It's a stand-alone database, which means that data is stored in a single file in the storage without needing a DBMS to manage all the operations.

Windows Phone uses SQL CE 3.5 (the latest release at this time is 4.0, but it is not supported) and doesn't support SQL query execution. Every operation is made using LINQ to SQL, which is one of the first of Microsoft's ORM solutions.

Note: ORM (Object-Relation Mapping) solutions are libraries that are able to automatically translate object operations (insert, edit, remove) into database operations. This way, you can keep working on your project using an object-oriented approach. ORM will take care of writing the required SQL queries to store your data in the database.

The approach used by SQL CE on Windows Phone is called **code first**. The database is created the first time the data is needed, according to the entities definition that you're going to store in tables. Another solution is to include an already existing SQL CE file in your Visual Studio project. In this case, you'll only be able to work with it in read-only mode.

How to define the database

The first step is to create the entities that you'll need to store in your database. Each entity will be mapped to a specific table.

Entity definition is made using attributes, which are part of the **System.Data.Linq.Mapping** namespace. Each property is decorated with an attribute, which will be used to translate it into a column. In the following sample we adapt the familiar **Person** class to be stored in a table:

```
[Table]
public class Person
{
    [Column(IsPrimaryKey = true, CanBeNull = false, IsDbGenerated = true)]
    public string Id { get; set; }

    [Column]
    public string Name { get; set; }

    [Column]
    public string Surname { get; set; }
}
```

The entire entity is marked with the **Table** attribute, while every property is marked with the **Column** attribute. Attributes can be customized with some properties, like:

- **IsPrimaryKey** to apply to columns that are part of the primary key.
- **IsDbGenerated** in case the column's value needs to be automatically generated every time a new row is inserted (for example, an automatically incremented number).
- **Name** if you want to assign to the column a different name than the property.
- **DbType** to customize the column's type. By default, the column's type is automatically set by the property's type.

Working with the database: the DataContext

DataContext is a special class that acts as an intermediary between the database and your application. It exposes all the methods needed to perform the most common operations, like insert, update, and delete.

The **DataContext** class contains the connection string's definition (which is the path where the database is stored) and all the tables that are included in the database. In the following sample, you can see a **DataContext** definition that includes the **Person** table we've previously defined:

```
public class DatabaseContext: DataContext
{
    public static string ConnectionString = "Data
source=isostore:/Persons.sdf";
```

```
    public DatabaseContext(string connectionString):base(connectionString)
    {

    }

    public Table<Person> Persons;
}
```

A separate class of your project inherits from the **DataContext** class. It will force you to implement a public constructor that supports a connection string as its input parameter. There are two connection string types, based on the following prefixes:

- **isostore:/** means that the file is stored in the local storage. In the previous sample, the database's file name is **Persons.sdf** and it's stored in the storage's root.
- **appdata:/** means that the file is stored in the Visual Studio project instead. In this case, you're forced to set the **File Mode** attribute to **Read Only**.

```
public static string ConnectionString = "Data source=appdata:/Persons.sdf;
File Mode=Read Only";
```

Eventually, you can also encrypt the database by adding a **Password** attribute to the connection string:

```
public static string ConnectionString = "Data source=isostore:/Persons.sdf;
Password='password'";
```

Creating the database

As soon as the data is needed, you'll need to create the database if it doesn't exist yet. For this purpose, the **DataContext** class exposes two methods:

- **DatabaseExists()** returns whether the database already exists.
- **CreateDatabase()** effectively creates the database in the storage.

In the following sample, you can see a typical database initialization that is executed every time the application starts:

```
private void OnCreateDatabaseClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        if (!db.DatabaseExists())
        {
```

```
            db.CreateDatabase();
        }
    }
}
```

Working with the data

All the operations are made using the **Table<T>** object that we've declared in the **DataContext** definition. It supports standard LINQ operations, so you can query the data using methods like **Where()**, **FirstOrDefault()**, **Select()**, and **OrderBy()**.

In the following sample, you can see how we retrieve all the **Person** objects in the table whose name is Matteo:

```
private void OnShowClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        List<Person> persons = db.Persons.Where(x => x.Name ==
"Matteo").ToList();
    }
}
```

The returned result can be used not only for display purposes, but also for editing. To update the item in the database, you can change the values of the returned object by calling the **SubmitChanges()** method exposed by the **DataContext** class.

To add new items to the table, the **Table<T>** class offers two methods: **InsertOnSubmit()** and **InsertAllOnSubmit()**. The first method can be used to insert a single object, while the second one adds multiple items in one operation (in fact, it accepts a collection as a parameter).

```
private void OnAddClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        Person person = new Person
        {
            Name = "Matteo",
            Surname = "Pagani"
        };
        db.Persons.InsertOnSubmit(person);
        db.SubmitChanges();
    }
}
```

Please note again the **`SubmitChanges()`** method: it's important to call it every time you modify the table (by adding a new item or editing or deleting an already existing one), otherwise changes won't be saved.

In a similar way, you can delete items by using the **`DeleteOnSubmit()`** and **`DeleteAllOnSubmit()`** methods. In the following sample, we delete all persons with the name Matteo:

```
private void OnDeleteClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        List<Person> persons = db.Persons.Where(x => x.Name ==
"Matteo").ToList();
        db.Persons.DeleteAllOnSubmit(persons);
        db.SubmitChanges();
    }
}
```

Relationships

In the previous sections, we've talked about data that is stored in a single table. Now it's time to introduce relationships, which are a way to connect two or more tables. As an example, we'll add a new **`Order`** entity to our database, which we'll use to save the orders made by users stored in the **`Person`** table.

With LINQ to SQL we'll be able to:

- Add a **`Person`** property to the **`Order`** entity that will store a reference to the user who made the order.
- Add an **`Orders`** collection to the **`Person`** entity that will contain all the orders made by the user.

This is accomplished by using a **foreign key**, which is a property declared in the **`Order`** entity that will hold the primary key value of the user who made the order.

Here is how the **`Order`** class looks:

```
[Table]
public class Order
{
    [Column(IsPrimaryKey = true)]
    public int OrderCode
    {
        get;
        set;
```

```
    }

    [Column]
    public double TotalPrice
    {
        get;
        set;
    }

    [Column]
    public string ShippingAddress
    {
        get;
        set;
    }

    [Column]
    public int PersonId
    {
        get;
        set;
    }

    private EntityRef<Person> _Person;

    [Association(Name = "PersonOrders",
        Storage = "_Person",
        ThisKey = "PersonId",
        OtherKey = "PersonId",
        IsForeignKey = true)]
    public Person Person
    {
        get
        {
            return this._Person.Entity;
        }
        set
        {
            Person previousValue = this._Person.Entity;
            if (((previousValue != value) ||
(this._Person.HasLoadedOrAssignedValue == false)))
            {
                if ((previousValue != null))
                {
                    this._Person.Entity = null;
                    previousValue.Orders.Remove(this);
                }
                this._Person.Entity = value;
                if ((value != null))
                {
```

```
                    value.Orders.Add(this);
                    this.PersonId = value.Id;
                }
                else
                {
                    this.PersonId = default(int);
                }
            }
        }
    }
}
```

There are two key properties in the class definition:

- **PersonId** is the foreign key, which simply holds the person's ID.
- **Person** is a real **Person** object that, thanks to the **Association** attribute, is able to hold a reference to the user who made the order. The property's setter contains some logic to manage whether you're adding a new value or removing an already existing one.

Of course, we have to also change the **Person** class definition in order to manage the relationships:

```
[Table]
public class Person
{
    public Person()
    {
        _Orders = new EntitySet<Order>();
    }
    [Column(IsPrimaryKey = true, CanBeNull = false, IsDbGenerated = true)]
    public int Id { get; set; }

    [Column]
    public string Name { get; set; }

    [Column]
    public string Surname { get; set; }

    private EntitySet<Order> _Orders;

    [Association(Name = "PersonOrders",
    Storage = "_Orders",
    ThisKey = "PersonId",
    OtherKey = "PersonId",
    DeleteRule = "NO ACTION")]
    public EntitySet<Order> Orders
    {
        get
```

```
        {
            return this._Orders;
        }
        set
        {
            this._Orders.Assign(value);
        }
    }
}
```

Also in this class, we've defined a new property called **Orders**, whose type is **EntitySet<T>**, where **T** is the type of the other table involved in the relationship. Thanks to the **Association** attribute, we are be able to access all the orders made by a user simply by querying the **Orders** collection.

In the following samples, you can see two common operations in which a relationship is involved: creation and selection.

```
private void OnAddClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        Person person = new Person
        {
            Name = "Matteo",
            Surname = "Pagani",
        };

        Order order = new Order
        {
            TotalPrice = 55,
            ShippingAddress
                = "Fake Street, Milan",
            Person = person
        };

        db.Orders.InsertOnSubmit(order);
        db.SubmitChanges();
    }
}

private void OnQueryClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        Order result = db.Orders.FirstOrDefault(x => x.OrderCode == 1);
```

```
        MessageBox.Show(result.Person.Name);
    }
}
```

Since **Person** is a property of the **Order** class, it's enough to create a new order and set the object that represents the user who made the order as a value of the **Person** property.

In the same way, when we get an order we are able to get the user's details simply by querying the **Person** property. In the previous sample, we display the name of the user who made the order.

Updating the schema

A common scenario when you're planning to release an application update is that you've changed the database schema by adding a new table or new column, for example.

SQL CE in Windows Phone offers a specific class to satisfy this requirement, called **DatabaseSchemaUpdater**, which offers some methods to update an already existing database's schema.

Note: The DatabaseSchemaUpdater's purpose is just to update the schema of an already existing database. You still need to update your entities and DataContext definition to reflect the new changes.

The key property offered by the **DatabaseSchemaUpdater** class is **DatabaseSchemaVersion**, which is used to track the current schema's version. It's important to properly set it every time we apply an update because we're going to use it when the database is created or updated to recognize whether we're using the latest version.

After you've modified your entities or the **DataContext** definition in your project, you can use the following methods:

- **AddTable<T>()** if you've added a new table (of type **T**).
- **AddColumn<T>()** if you've added a new column to a table (of type **T**).
- **AddAssociation<T>()** if you've added a new relationship to a table (of type **T**).

The following sample code is executed when the application starts and needs to take care of the schema update process:

```
private void OnUpdateDatabaseClicked(object sender, RoutedEventArgs e)
{
    using (DatabaseContext db = new
DatabaseContext(DatabaseContext.ConnectionString))
    {
        if (!db.DatabaseExists())
        {
```

```
            db.CreateDatabase();
            DatabaseSchemaUpdater updater =
db.CreateDatabaseSchemaUpdater();
            updater.DatabaseSchemaVersion = 2;
            updater.Execute();
        }
        else
        {
            DatabaseSchemaUpdater updater =
db.CreateDatabaseSchemaUpdater();
            if (updater.DatabaseSchemaVersion < 2)
            {
                updater.AddColumn<Person>("BirthDate");
                updater.DatabaseSchemaVersion = 2;
                updater.Execute();
            }
        }
    }
}
```

We're assuming that the current database's schema version is 2. In case the database doesn't exist, we simply create it and, using the **`DatabaseSchemaUpdater`** class, we update the **`DatabaseSchemaVersion`** property. This way, the next time the data will be needed, the update operation won't be executed since we're already working with the latest version.

Instead, if the database already exists, we check the version number. If it's an older version, we update the current schema. In the previous sample, we've added a new column to the **`Person`** table, called **`BirthDate`** (which is the parameter requested by the **`AddColumn<T>()`** method). Also in this case we need to remember to properly set the **`DatabaseSchemaVersion`** property to avoid further executions of the update operation.

In both cases, we need to apply the described changes by calling the **`Execute()`** method.

SQL Server Compact Toolbox: An easier way to work with SQL CE

Erik Ej, a Microsoft MVP, has developed a powerful Visual Studio tool called **SQL Server Compact Toolbox** that can be very helpful for dealing with SQL CE and Windows Phone applications.

Two versions of the tool are available:

- As an extension that's integrated into commercial versions of Visual Studio.
- As a stand-alone tool for Visual Studio Express since it does not support extensions.

The following are some of the features supported by the tool:

- Automatically create entities and a **`DataContext`** class starting from an already existing SQL CE database.

- The generated **`DataContext`** is able to copy a database from your Visual Studio project to your application's local storage. This way, you can start with a prepopulated database and, at the same time, have write access.
- The generated **`DataContext`** supports logging in the Visual Studio Output Window so you can see the SQL queries generated by LINQ to SQL.

Using databases: SQLite

SQLite, from a conceptual point of view, is a similar solution to SQL CE: it's a stand-alone database solution, where data is stored in a single file without a DBMS requirement.

The pros of using SQLite are:

- It offers better performance than SQL CE, especially with large amounts of data.
- It is open source and cross-platform; you'll find a SQLite implementation for Windows 8, Android, iOS, web apps, etc.

SQLite support has been introduced only in Windows Phone 8 due to the new native code support feature (since the SQLite engine is written in native code), and it's available as a Visual Studio extension that you can download here.

After you've installed it, you'll find the SQLite for Windows Phone runtime available in the **Add reference** window, in the **Windows Phone Extension** section. Be careful; this runtime is just the SQLite engine, which is written in native code. If you need to use a SQLite database in a C# application, you'll need a third-party library that is able to execute the appropriate native calls for you.

In actuality, there are two available SQLite libraries: **sqlite-net** and **SQLite Wrapper for Windows Phone**. Unfortunately, neither of them is as powerful and flexible as the LINQ to SQL library that is available for SQL CE.

Let's take a brief look at them. We won't dig too deeply. Since they're in constant development, things can change very quickly.

sqlite-net

sqlite-net is a third-party library. The original version for Windows Store apps is developed by Frank A. Krueger, while the Windows Phone 8 port is developed by Peter Huene.

The Windows Phone version is available on GitHub. Its configuration procedure is a bit tricky and changes from time to time, so be sure to follow the directions provided by the developer on the project's home page.

sqlite-net offers a LINQ approach to use the database that is similar to the code-first one offered by LINQ to SQL with SQL CE.

For example, in sqlite-net, tables are mapped with your project's entities. The difference is that, this time, attributes are not required since every property will be automatically translated into a column. Attributes are needed only if you need to customize the conversion process, as in the following sample:

```
public class Person
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }

    [MaxLength(50)]
    public string Name { get; set; }

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

Surname doesn't have any attribute, so it will be automatically converted into a **varchar** column. Instead, we set **Id** as a primary key with an auto increment value, while we specify that **Name** can have a maximum length of 50 characters.

All the basic operations with the database are accomplished using the **SQLiteAsyncConnection** class, which exposes asynchronous methods to create tables, query the data, delete items, etc. It requires as an input parameter the local storage path where the database will be saved.

As with SQL CE and LINQ to SQL, we need to create the database before using it. This is done by calling the **CreateTableAsync<T>()** method for every table we need to create, where **T** is the table's type. In the following sample, we create a table to store the **Person** entity:

```
private async Task CreateDatabase()
{
    SQLiteAsyncConnection conn = new
SQLiteAsyncConnection(Path.Combine(ApplicationData.Current.LocalFolder.Path
, "people.db"), true);
    await conn.CreateTableAsync<Person>();
}
```

We don't have a method to verify whether the table already exists since it's not needed; if the table we're creating already exists, the **CreateTableAsync<T>()** method simply won't do anything.

In a similar way to LINQ to SQL, queries are performed using the **Table<T>** object. The only difference is that all the LINQ methods are asynchronous.

```
private async void OnReadDataClicked(object sender, RoutedEventArgs e)
{
```

```
    SQLiteAsyncConnection conn = new
SQLiteAsyncConnection(Path.Combine(ApplicationData.Current.LocalFolder.Path
, "people.db"), true);
    List<Person> person = await conn.Table<Person>().Where(x => x.Name ==
"Matteo").ToListAsync();
}
```

In the previous sample, we retrieve all the **Person** objects whose name is Matteo.

Insert, update, and delete operations are instead directly executed using the **SQLiteAsyncConnection** object, which offers the **InsertAsync()**, **UpdateAsync()**, and **DeleteAsync()** methods. It is not required to specify the object's type; sqlite-net will automatically detect it and execute the operation on the proper table. In the following sample, you can see how a new record is added to a table:

```
private async void OnAddDataClicked(object sender, RoutedEventArgs e)
{
    SQLiteAsyncConnection conn = new
SQLiteAsyncConnection(Path.Combine(ApplicationData.Current.LocalFolder.Path
, "people.db"), true);

    Person person = new Person
    {
        Name = "Matteo",
        Surname = "Pagani"
    };

    await conn.InsertAsync(person);
}
```

sqlite-net is the SQLite library that offers the easiest approach, but it has many limitations. For example, foreign keys are not supported, so it's not possible to easily manage relationships.

SQLite Wrapper for Windows Phone

SQLite Wrapper for Windows Phone has been developed directly by Microsoft team members (notably Peter Torr and Andy Wigley) and offers a totally different approach than sqlite-net. It doesn't support LINQ, just plain SQL query statements.

The advantage is that you have total control and freedom, since every SQL feature is supported: indexes, relationships, etc. The downside is that writing SQL queries for every operation takes more time, and it's not as easy and intuitive as using LINQ.

To learn how to configure the wrapper in your project, follow the instructions posted on the CodePlex project page. You'll have to download the project's source code and add the correct wrapper version to your solution—there are two separate libraries, one for Windows Phone 8 and one for Windows Store apps.

The key class is called **Database**, which takes care of initializing the database and offers all the methods needed to perform the queries. As a parameter, you need to set the local storage path to save the database. If the path doesn't exist, it will be automatically created. Then, you need to open the connection using the **OpenAsync()** method. Now you are ready to perform operations.

There are two ways to execute a query based on the value it returns.

If the query doesn't return a value—for example, a table creation—you can use the **ExecuteStatementAsync()** method as shown in the following sample:

```
private async void OnCreateDatabaseClicked(object sender, RoutedEventArgs
e)
{
    Database database = new Database(ApplicationData.Current.LocalFolder,
"people.db");

    await database.OpenAsync();

    string query = "CREATE TABLE PEOPLE " +
                   "(Id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL," +
                   "Name varchar(100), " +
                   "Surname varchar(100))";

    await database.ExecuteStatementAsync(query);
}
```

The previous method simply executes the query against the opened database. In the sample, we create a **People** table with two fields, **Name** and **Surname**.

The query, instead, can contain some dynamic parameters or return some values. In this case, we need to introduce a new class called **Statement** as demonstrated in the following sample:

```
private async void OnAddDataClicked(object sender, RoutedEventArgs e)
{
    Database database = new Database(ApplicationData.Current.LocalFolder,
"people.db");

    await database.OpenAsync();

    string query = "INSERT INTO PEOPLE (Name, Surname) VALUES (@name,
@surname)";
    Statement statement = await database.PrepareStatementAsync(query);
```

```
    statement.BindTextParameterWithName("@name", "Matteo");
    statement.BindTextParameterWithName("@surname", "Pagani");

    await statement.StepAsync();
}
```

The **Statement** class identifies a query, but it allows additional customization to be performed with it. In the sample, we use it to assign a dynamic value to the **Name** and **Surname** parameters. We set the placeholder using the @ prefix (**@name** and **@surname**), and then we assign them a value using the **BindTextParameterWithName()** method, passing the parameter's name and the value.

BindTextParameterWithName() isn't the only available method, but it's specifically for string parameters. There are other methods based on the parameter's type, such as **BindIntParameterWithName()** for numbers.

To execute the query, we use the **StepAsync()** method. Its purpose isn't just to execute the query, but also to iterate the resulting rows.

In the following sample, we can see how this method can be used to manage the results of a **SELECT** query:

```
private async void OnGetDataClicked(object sender, RoutedEventArgs e)
{
    Database database = new Database(ApplicationData.Current.LocalFolder,
"people.db");

    await database.OpenAsync();

    string query = "SELECT * FROM PEOPLE";
    Statement statement = await database.PrepareStatementAsync(query);

    while (await statement.StepAsync())
    {
        MessageBox.Show(statement.GetTextAt(0) + " " +
statement.GetTextAt(1));
    }
}
```

The **StepAsync()** method is included inside a **while** statement. At every loop iteration, we'll get the reference to the next row returned by the query, starting from the first one. After we've iterated all the rows, the application will quit the **while** loop.

When we have a row's reference, we can access its values by using the column index and the **Get()** method. We have a **Get()** variant for every data type, like **GetText()**, **GetInt()**, etc.

Another way is to access the columns using the **`Columns`** collection with the column name as the index. In this case, you first have to call the **`EnableColumnsProperty()`** method, as shown in the following sample:

```
private async void OnGetSomeDataWithColumnsPropertyClicked(object sender,
RoutedEventArgs e)
{
    Database database = new Database(ApplicationData.Current.LocalFolder,
"people.db");

    await database.OpenAsync();

    string query = "SELECT * FROM PEOPLE";
    Statement statement = await database.PrepareStatementAsync(query);

    statement.EnableColumnsProperty();

    while (await statement.StepAsync())
    {
        MessageBox.Show(statement.Columns["Name"] + " " +
statement.Columns["Surname"]);
    }
}
```

Keep in mind that this approach is slower than using the column's index.

A quick recap

This chapter delivered more key concepts that every Windows Phone developer should be familiar with. Managing local data is important, and in this chapter we've discussed the following approaches:

- Correctly using files and folders in the isolated storage thanks to the Windows Runtime APIs.
- Easily managing our application's settings by using the **`IsolatedStorageSettings`** class.
- Storing our application's data using serialization and deserialization in simple app scenarios.
- In case of more complex applications, we've seen how we can better organize our data using databases. We analyzed two available solutions: SQL CE and SQLite. They both offer a stand-alone database platform. SQL CE is exclusive to Windows Phone, but it is more powerful and easier to use; SQLite is open source and cross-platform, but you have to rely on third-party libraries which aren't as powerful as LINQ to SQL for SQL CE.

Chapter 5 Data Access: Network

Checking the Internet connection

All Windows Phone devices have a built-in network connection but, for the same reason we learned how to locally store data, we need to be ready to manage how users are using our application while a connection is missing.

For this purpose, the Windows Phone framework includes a class that can be used to discover information about the Internet connection called **DeviceNetworkInformation**, which is part of the **Microsoft.Phone.Net.NetworkInformation** namespace.

The most important one is **IsNetworkAvailable**, which tells us whether an Internet connection is available. We should always use this API before performing an operation that requires a connection, like the following sample:

```
private void OnCheckConnectionClicked(object sender, RoutedEventArgs e)
{
    if (DeviceNetworkInformation.IsNetworkAvailable)
    {
        MessageBox.Show("You are connected to Internet");
        //Perform network operations.
    }
    else
    {
        MessageBox.Show("You aren't connected to Internet");
    }
}
```

The class also offers an event called **NetworkAvailabilityChanged** which is triggered every time the connection status changes. It's useful if you want to quickly react to network changes, such as enabling or disabling certain application features.

```
public MainPage()
{
    InitializeComponent();
    DeviceNetworkInformation.NetworkAvailabilityChanged +=
DeviceNetworkInformation_NetworkAvailabilityChanged;
}

private void DeviceNetworkInformation_NetworkAvailabilityChanged(object
sender, NetworkNotificationEventArgs e)
{
```

```
    if (e.NotificationType ==
NetworkNotificationType.InterfaceDisconnected)
    {
        MessageBox.Show("Disconnected");
    }
    else if (e.NotificationType == NetworkNotificationType.InterfaceConnected)
    {
        MessageBox.Show("Connected");
    }
}
```

The return parameters contain a property called **NotificationType**, of the type **NetworkNotificationType**, which tells us the current network status.

However, with the **DeviceNetworkInformation** you'll be able to also get other information about the current network status, like whether the user has enabled the cellular data connection (**IsCellularDataEnabled**), the Wi-Fi connection (**IsWiFiEnabled**), or roaming options (**IsCellularDataRoamingOptions**).

The framework offers another useful class to deal with network connections called **NetworkInterface**. By using the **NetworkInterfaceType** property, you'll be able to identify which connection type is currently in use. For example, we can use this property to avoid downloading big files while using a cellular connection.

NetworkInterfaceType is an enumerator that can assume many values. The most important ones are:

- **MobileBroadbandGsm** and **MobileBroadbandCdma** when the phone is connected to a cellular network (GSM or CDMA, according to the country).
- **Wireless80211**, when the phone is connected to a Wi-Fi network.

In the following sample, we display a message on the screen with the current connection type:

```
private void OnCheckConnectionTypeClicked(object sender, RoutedEventArgs e)
{
    if (NetworkInterface.NetworkInterfaceType ==
NetworkInterfaceType.MobileBroadbandGsm ||
NetworkInterface.NetworkInterfaceType ==
NetworkInterfaceType.MobileBroadbandCdma)
    {
        MessageBox.Show("Mobile");
    }
    else if (NetworkInterface.NetworkInterfaceType ==
NetworkInterfaceType.Wireless80211)
    {
        MessageBox.Show("Wi-Fi");
    }
}
```

Performing network operations: HttpClient

The Windows Phone framework has two built-in classes for performing network operations: **WebClient** and **HttpWebRequest**. Unfortunately, neither of them is perfect. **WebClient** is very easy to use, but it's based on the old callback approach (unless you install the **Async for .NET** package we discussed in Chapter 3). **HttpWebRequest** is very powerful, but it is complex to use and based on an old asynchronous pattern that is hard to understand.

The Windows Runtime has introduced a new class called **HttpClient**, which takes the best of both worlds. It's powerful and offers great performance, but it's easy to use and offers methods that are based on the new async and await pattern. This class is available in the full Windows Runtime for Windows Store apps, but it's not included in the Windows Phone Runtime subset—you'll have to install it from NuGet.

The **HttpClient** class, as we'll see later, is great not just for performing generic network operations like downloading and uploading file, but also for interacting with web services. In fact, it exposes asynchronous methods for every HTTP command, like GET, POST, PUT, etc.

Note: To be able to interact with the network you'll have to enable the ID_CAP_NETWORKING option in the manifest file. It is enabled in every new Windows Phone project by default.

Downloading data

Files are usually downloaded using the GET HTTP command, so **HttpClient** offers the **GetAsync()** method. To make developers' lives simpler, **HttpClient** has some built-in methods to download the most common file types, such as **GetStringAsync()** for downloading text files like XML, RSS, or REST responses; or **GetByteArrayAsync()** and **GetStreamAsync()** to get binary file content.

Downloading strings is really easy. The method **GetStringAsync()** requires as a parameter the file's URL and returns the file's content as a string. In the following sample, you can see how to download my blog's RSS feed:

```
private async void OnDownloadStringClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://feeds.feedburner.com/qmatteoq_eng");
}
```

Once you have the string, you can perform the required operations according to your scenario. For example, in the previous sample I could have parsed the returned XML with LINQ to XML to display the news list on the screen.

Downloading binary files can be accomplished in many ways. You can use **GetByteArrayAsync()** if you prefer to work with bytes, or **GetStreamAsync()** if you prefer to manipulate the file's content as a stream.

In the following sample, you'll see how to download a picture using the **GetByteArrayAsync()** method. It returns a byte's array, which can be easily saved in the local storage using the APIs we learned about in Chapter 4.

```
private async void OnDownloadFileClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();

    byte[] bytes = await
client.GetByteArrayAsync("http://www.syncfusion.com/Content/en-
US/Home/Images/syncfusion-logo.png");
    StorageFile storageFile = await
ApplicationData.Current.LocalFolder.CreateFileAsync("picture.png",
CreationCollisionOption.ReplaceExisting);
    IRandomAccessStream accessStream = await
storageFile.OpenAsync(FileAccessMode.ReadWrite);

    using (IOutputStream outputStream = accessStream.GetOutputStreamAt(0))
    {
        DataWriter writer = new DataWriter(outputStream);
        writer.WriteBytes(bytes);
        await writer.StoreAsync();
    }
}
```

By using the **DataWriter** class, we're able to save the byte array returned by the **HttpClient** class into a file in the local storage called picture.png.

If you need full control over the download operation, you can use the generic **GetAsync()** method, which returns a **HttpResponseMessage** object with the entire response's content, like headers, status code, etc.

In the following sample you can see how, by using the **GetAsync()** method, we're able to download a picture as a **Stream** and display it in an **Image** control placed on the page. To the method we pass a second parameter of type **HttpCompletionOption**, which tells the class when to mark the operation as completed. Since we want the full content of the response (which is the downloaded image), we use the **ResponseContentRead** option.

```
private async void OnDownloadStreamClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    HttpResponseMessage httpResponseMessage = await
client.GetAsync("http://www.syncfusion.com/Content/en-
```

```
US/Home/Images/syncfusion-logo.png",
HttpCompletionOption.ResponseContentRead);

    Stream stream = await httpResponseMessage.Content.ReadAsStreamAsync();

    BitmapImage image = new BitmapImage();
    image.SetSource(stream);
    Logo.Source = image;
}
```

Notice that the specific methods we've seen offered by the **HttpClient** class are simply a shortcut to the methods offered by the **Content** property of the **HttpResponseMessage** class (in the previous sample, we use the **ReadAsStreamAsync()** method to return the response's content as a **Stream**).

Uploading data

Uploading data is accomplished in a similar way: the operation is usually performed using the POST command, so the **HttpClient** class exposes the **PostAsync()** method for this purpose.

The content to send is prepared using the **HttpContent** class, which offers many implementations: **StreamContent** to send a file's stream, **ByteArrayContent** to send a binary file, **StringContent** to send a string, or **MultipartFormDataContent** to send content encoded using multipart/form-data MIME type.

In the following sample, you can see how to upload a file's stream to a service:

```
private async void OnUploadFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile storageFile = await
ApplicationData.Current.LocalFolder.GetFileAsync("picture.png");
    IRandomAccessStream accessStream = await
storageFile.OpenAsync(FileAccessMode.ReadWrite);

    HttpClient client = new HttpClient();
    HttpContent content = new
StreamContent(accessStream.AsStreamForRead());
    await client.PostAsync("http://wp8test.azurewebsites.net/api/values",
content);
}
```

After we have retrieved the stream of a file stored in the local storage, we pass it to a new instance of the **StreamContent** class. Then, we call the **PostAsync()** method, passing the service's URL and the **HttpContent** object as parameters.

Using REST services

REST (Representational State Transfer) is, without a doubt, the most used approach nowadays in web service development, especially in the mobile world.

REST services have become very popular for two reasons:

- They are based on the HTTP protocol and they expose operations using the standard HTTP commands (like GET, POST, PUT, etc.).
- They return data using standard languages like XML and JSON.

Almost every platform natively supports these technologies, so you don't have to look for special libraries to interact with them as is needed for WSDL-based web services.

We've already seen how, with the **`HttpClient`** class, it's simple to interact with the web by using the standard HTTP commands. Most of the time, to interact with a REST service, you'll simply have to execute the **`GetStringAsync()`** method to get the XML or JSON response.

Once you have the result, most of the time you'll have to convert it into objects to be used inside the application. In Chapter 4 we discussed the easiest way to accomplish this task: deserialization, which means translating plain text into complex objects. We can use the **`DataContractSerializer`** or **`DataContractJsonSerializer`** classes to do this. In this case, you just have to refer to the deserialization process since the procedure is the same.

```
private async void OnConsumeServiceClicked(object sender, RoutedEventArgs
e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values");

    using (MemoryStream ms = new
MemoryStream(Encoding.Unicode.GetBytes(result)))
    {
        DataContractJsonSerializer serializer = new
DataContractJsonSerializer(typeof(List<Person>));
        List<Person> people = serializer.ReadObject(ms) as List<Person>;
    }
}
```

We assume that the service returns, in JSON format, a list of persons:

```
[
   {
      "id":1,
      "name":"Matteo",
      "surname":"Pagani"
```

```
    },
    {
        "id":2,
        "name":"John",
        "surname":"Doe"
    }
]
```

With the help of the **DataContractJsonSerializer** class, we are able to convert the previous JSON into a list of **Person** objects, which is the same class we have used in many other samples in this book. The difference in this sample is that we are able to control the serialization process, in case, for example, the property names returned from the JSON are different than the ones we use in our classes. This is a very common scenario when dealing with JSON since property names are typically lowercase, while in C# objects they are CamelCase. This result is achieved by using the **[DataMember]** attribute, which can be applied to the properties we want to serialize. In the following sample, you can see that the attribute offers a property called **Name**, which can be used to specify which property name we expect to find in the JSON file.

```
public class Person
{
    [DataMember(Name = "id")]
    public int Id { get; set; }

    [DataMember(Name = "name")]
    public string Name { get; set; }

    [DataMember(Name = "surname")]
    public string Surname { get; set; }
}
```

This approach has a downside: REST services always return a plain string, while the **DataContractJsonSerializer** class requires a **Stream** as an input parameter of the **ReadObject()** method, so we're always forced to convert it using a **MemoryStream** object.

There is another way to accomplish the same result. Let me introduce JSON.NET, a third-party library that offers some additional useful features for handling JSON data. In addition, it offers better performance and is able to deserialize complex JSON files quicker.

It can be easily installed using NuGet, and its official website offers comparisons with other JSON libraries and detailed documentation.

In the following sample, we use JSON.NET to achieve the same result of the code that we've previously used:

```
private async void OnGetDataClicked(object sender, RoutedEventArgs e)
{
```

```
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values");

    List<Person> people =
JsonConvert.DeserializeObject<List<Person>>(result);
}
```

The **JsonConvert** class (which is part of the **Newtonsoft.Json** namespace) exposes the **DeserializeObject<T>()** method, where **T** is the object's type we expect in return. As an input parameter, it requires only the JSON string that we've downloaded from the service.

It's also possible to control the deserialization process using attributes, like we previously did with the **DataMember** attribute. In the following sample, you can see how we can manually define how JSON properties should be translated:

```
public class Person
{
    [JsonProperty("id")]
    public int Id { get; set; }

    [JsonProperty("name")]
    public string Name { get; set; }

    [JsonProperty("surname")]
    public string Surname { get; set; }
}
```

Tip: You will often have to use third-party services, so you won't know the exact mapping between entities and JSON data. There is a website that will help you in this scenario: http://json2csharp.com/. Simply paste the JSON returned by your service, and it will generate for you the needed C# classes to map the JSON data.

LINQ to JSON

Another interesting feature introduced by JSON.NET is LINQ to JSON, which is a LINQ-based language that, similar to LINQ to XML, can be used to manipulate a JSON string to extract just the information you need. This approach is useful when you do not really need to use deserialization, but just need to extract some data from the JSON response you received from the service.

The starting point to use LINQ to JSON is the **JObject** class, which identifies a JSON file.To start working with it, you simply have to call the **Parse()** method, passing as parameter the JSON string, as shown in the following sample:

```
private async void OnParseJson(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values");

    JObject json = JObject.Parse(result);
}
```

Now you're ready to execute some operations. Let's take a look at the most common ones.

Simple JSON

Here is an example of a simple JSON file:

```
{
   "Id":1,
   "Name":"Matteo",
   "Surname":"Pagani"
}
```

With LINQ to JSON, we are able to extract a single property's value in the following way:

```
private async void OnParseJson(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values/1");

    JObject json = JObject.Parse(result);
    string value = json["Name"].Value<string>();
}
```

The **`JObject`** class is treated like a collection, so you can simply access a property using its name as a key. In the end, you can extract the value by using the **`Value<T>()`** method, where **`T`** is the data type we expect to be stored.

Complex JSON

Like C# objects, JSON objects can also have complex properties, as shown in the following sample:

```
{
   "Id":1,
   "Name":"Matteo",
   "Surname":"Pagani",
   "Address":{
      "Street":"Fake address",
      "City":"Milan"
   }
}
```

Address is a complex property because it contains other nested properties. To access these properties, we need to use the **SelectToken()** method, passing the full JSON path as a parameter:

```
private async void OnParseJson(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values/1");

    JObject json = JObject.Parse(result);
    string city = json.SelectToken("Address.City").Value<string>();
}
```

With the **Address.City** path, we are able to extract the value of the **City** property that is part of the **Address** node.

JSON collections

When you handle JSON collections, you can use the **Children()** method to get access to all the children nodes of a specific property. Let's use, as an example, a JSON code that we've previously seen:

```
[
   {
      "Id":1,
      "Name":"Matteo",
      "Surname":"Pagani"
   },
   {
      "Id":2,
      "Name":"John",
      "Surname":"Doe"
   }
]
```

In this case, we can use the **JArray** class and the **Children()** method to extract all the collection's elements. In the following sample you can see how we use it to get a subcollection with only the **Name** property's values.

```
private async void OnGetDataClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await
client.GetStringAsync("http://wp8test.azurewebsites.net/api/values");

    JArray json = JArray.Parse(result);
    List<string> list = json.Children().Select(x =>
x["Name"].Value<string>()).ToList();
}
```

Background transfers

The **HttpClient** class we've previously seen, like the **WebClient** and **HttpWebRequest** classes, can be used only for foreground operations. When the application is suspended, network transfers are canceled.

When we have to deal with big data transfers, forcing the user to keep the app opened doesn't create the best user experience. For this scenario, Windows Phone 7.5 has introduced background transfer APIs that can be used to start a download or upload operation and to continue it even if the app is suspended.

However, there are some limitations that have been introduced to avoid battery issues and high data plan consumption:

- If the phone is connected to a cellular network, files bigger than 20 MB cannot be downloaded.
- If the phone is connected to a Wi-Fi network, files bigger than 100 MB can't be downloaded.
- The 100 MB limit can be exceeded only if the phone is connected to a Wi-Fi network and the battery is charging.
- A single application can queue up to 25 background transfer operations.
- The global operating system's queue can contain up to 500 background transfer operations.
- The phone can execute a maximum of two transfer operations at a time.

A background transfer is identified by the **BackgroundTransferRequest** class, which is part of the **Microsoft.Phone.BackgroundTransfer** namespace. As developers, we have control over some conditions that need to be satisfied for a background transfer created in our application to start, thanks to the **TransferPreferences** property that can get the following values:

- **None**, the default value: The transfer is started only if the phone is connected to a Wi-Fi network and the battery is charging.
- **AllowBattery**: The transfer is started only if the phone is connected to a Wi-Fi network, regardless of the power source.
- **AllowCelullar**: The transfer is started only if the phone is charging, regardless of the network connection.
- **AllowCellularAndBattery**: Always starts the transfer, regardless of the connection and power source conditions.

The **BackgroundTransferRequest** class exposes two event handlers that can be used to control the transfer:

- **TransferStatusChanged** is triggered when the transfer's status changes. The parameter returned by the method contains a **TransferStatus** object that notifies you of the current status (like **Completed** when the transfer ends, or **Paused** when the transfer is paused). There are also specific statuses that start with the **Waiting** prefix that tell you when a transfer is suspended because the conditions defined in the **TransferPreferences** property are not satisfied. For example, **WaitingForWiFi** is set when the transfer is waiting for the phone to be connected to a Wi-Fi network to start.
- **TransferProgressChanged** is triggered when a transfer's progress changes, meaning that new data has been downloaded or uploaded. Usually, this event handler is connected to a **ProgressBar** control since it exposes properties to notify you how much data has been transferred and how much data still needs to be downloaded or sent.

After you've defined a background transfer, you need to add it to the operating system's queue. Windows Phone will take care of starting it when the specified conditions are satisfied. To accomplish this task, we use the **BackgroundTransferService** class, which is the central background transfer manager. You can add, remove, or list background transfers that belong to the application.

In the following sample you can see a background transfer definition:

```
private BackgroundTransferRequest backgroundRequest;

private void OnStartBackgroundDownloadClicked(object sender, RoutedEventArgs
e)
{
    Uri sourceUrl = new Uri("http://wpsauce.com/wp-
content/uploads/2011/11/windows_phone_logo.jpg");
    Uri destinationUrl = new
Uri("/Shared/Transfers/windows_phone_logo.jpg",
UriKind.RelativeOrAbsolute);
    backgroundRequest = new BackgroundTransferRequest(sourceUrl,
destinationUrl);
    backgroundRequest.TransferStatusChanged +=
backgroundRequest_TransferStatusChanged;
    backgroundRequest.TransferPreferences =
TransferPreferences.AllowCellularAndBattery;
    BackgroundTransferService.Add(backgroundRequest);
}
```

```
void backgroundRequest_TransferStatusChanged(object sender,
BackgroundTransferEventArgs e)
{
    if (backgroundRequest.TransferStatus == TransferStatus.Completed)
    {
        //Manage the downloaded file.
        BackgroundTransferService.Remove(backgroundRequest);
    }
}
```

We register this transfer to be executed regardless of the available network connection and power source. The previous sample is related to a download operation, so we need to define a source URI (the file to download) and a destination URI (the local storage path where the file will be saved). Unlike what we've seen with **HttpClient**, we don't have to take care of the saving process; the file will be automatically downloaded and saved in the local storage since the download can also finish when the app is suspended. Both source and destination URIs are passed as parameters of the **BackgroundTransferRequest** constructor.

Note: Background transfers that are used to perform download operations always have to save the file inside the Shared/Transfers path in the local storage, which is automatically created when the app is installed—otherwise you'll get an exception. When the download is complete, you are free to move the file to another position if needed, but you can't schedule a background transfer that tries to download a file in a different folder.

Next, we subscribe to the **TransferStatusChanged** event. If the download is completed while the app is in the foreground, we are able to manage the downloaded file—for example, if it's an image, we can display it. Notice the **Remove()** operation that we perform on the **BackgroundTransferService** class. It's really important to always perform this task because the operating system won't automatically remove completed transfers from the application's queue and it can lead to unexpected issues since an application can't schedule more than 25 transfers.

Instead, if you need to upload a file, you'll need to create a **BackgroundTransferRequest** object in a different way. You still need to define two URIs: the source (the file to upload) and the destination (a service that is able to receive the file using the HTTP command set in the **Method** property). The destination URI can be passed in the **BackgroundTransferRequest**'s constructor (like we did previously), but the source URI needs to be set in the **UploadLocation** property, like in the following sample:

```
private void OnUploadFile()
{
    Uri destinationUrl = new
Uri("http://wp8test.azurewebsites.com/api/values", UriKind.Relative);
    Uri sourceUri = new Uri("/Shared/Transfers/image.png",
UriKind.Relative);
```

```
    BackgroundTransferRequest request = new
BackgroundTransferRequest(destinationUrl);
    request.UploadLocation = sourceUri;
    request.Method = "POST";
    BackgroundTransferService.Add(request);
}
```

A quick recap

In this chapter we've seen how to work with one of the most used features of a smartphone: an Internet connection. In detail, we've learned:

- Even if an Internet connection is a must-have for every device, we should be aware that sometimes users may not have an available connection. It's important to properly check whether the phone is connected to the Internet before doing any network operation.
- **`HttpClient`** is a new class introduced in Windows Runtime that helps perform network operations. We've seen how to use it to download and upload files, and interact with services.
- Downloading and uploading files is a common task, but nowadays more and more applications have to interact with web services to get the data they need. In this chapter we've learned how, thanks to the JSON.NET library, it's easy to work with REST services and convert JSON data into C# objects.
- **`HttpClient`** is a great help, but it works only if the application is in the foreground. When it's suspended, network operations are canceled. For this purpose, the framework offers some specific APIs to perform download and upload operations even in the background when the app is not in use.

Chapter 6 Integrating with the Hardware

Geolocation

All Windows Phone devices have built-in geolocation hardware. By using a combination of 3G, Wi-Fi, and GPS signal, the phone is able to identify the user's location and make it available to every app, thanks to a set of APIs included in the Windows Runtime.

Geolocation is another scenario where APIs are duplicated. The original API set was part of the Silverlight framework, but it has been expanded in the Windows Runtime.

The new main class for working with geolocation services is called **`Geolocator`**, and it's part of the **`Windows.Devices.Geolocation`** namespace.

Note: To use geolocation services, you'll need to enable the ID_CAP_LOCATION capability in the manifest file.

The first step to use geolocation services is to check the **`LocationStatus`** property value of the **`Geolocator`** class to identify the current status of the services. Specifically, we have to manage the **`PositionStatus.Disabled`** status. In this case, the user has disabled the geolocation services in the phone's settings, so we don't have to execute any operation related to geolocation, otherwise we will get an exception. If you want to keep track of the geolocation service's status, there's a specific event handler called **`StatusChanged`** that is invoked every time the status changes. It helps you to identify, for example, that the GPS is ready or that the user is in a location that is hard to track.

There are two ways to interact with the **`Geolocator`** class: asking for a single position (for example, a Twitter client that needs to geolocalize a tweet), or subscribing to an event handler that can be used to continuously track the user's location (for example, a running tracker app).

To ask for a single position, you simply have to call the asynchronous method **`GetGeopositionAsync()`**. It will return a **`Geoposition`** object containing a **`Coordinate`** property that will help you to identify where the user is.

Note: The Geoposition object has another property called `CivicAddress` which should contain a reference to the user's location using civic references (like the city, street address, etc.). This property isn't supported, so it will always return incorrect information. Later in this chapter we'll look at how to get the civic position of the user.

The following code sample demonstrates how to get a single user's position:

```
private async void OnGetSinglePositionClicked(object sender, RoutedEventArgs
e)
{
    Geolocator geolocator = new Geolocator();
    if (geolocator.LocationStatus != PositionStatus.Disabled)
    {
        Geoposition geoposition = await geolocator.GetGeopositionAsync();
        MessageBox.Show(string.Format("The user's coordinates are {0} -
{1}", geoposition.Coordinate.Latitude, geoposition.Coordinate.Latitude))
    }
}
```

To continuously track the user's position, instead, you need to subscribe to the **PositionChanged** event handler, which is invoked every time the user moves from the previous location. You can control how often this event is raised by setting three properties of the **Geolocator** object:

- **DesiredAccuracy**, which is the geolocation accuracy. The higher it is set, the more precise the result will be, and the more battery power will be consumed.
- **MovementThreshold**, which is the distance, in meters, the user should move from the previous location before the **PositionChanged** event is triggered.
- **ReportInterval**, which is the minimum number of milliseconds that should pass between two detections.

The **PositionChanged** event returns a parameter that contains a **Position** property of type **Geoposition**—it works the same way as we've previously seen for the **GetGeopositionAsync()** method.

```
private void OnStartTrackingPosition(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.MovementThreshold = 100;
    geolocator.ReportInterval = 1000;
    geolocator.DesiredAccuracy = PositionAccuracy.High;

    geolocator.PositionChanged += geolocator_PositionChanged;
}

private void geolocator_PositionChanged(Geolocator sender,
PositionChangedEventArgs args)
{
    Dispatcher.BeginInvoke(() =>
    {
        Latitude.Text = args.Position.Coordinate.Latitude.ToString();
        Longitude.Text = args.Position.Coordinate.Longitude.ToString();
```

```
    });
}
```

In the previous sample, we track the user's location every time he or she moves 100 meters from the previous location, only if at least one second has passed since the previous detection. Every time the **PositionChanged** event is raised, we display the **Latitude** and **Longitude** properties' values in two different **TextBlock** controls. Note that we're using the **Dispatcher** class we discussed in Chapter 3. It's required because the **PositionChanged** event is managed in a background thread, so we are not able to directly interact with the UI.

Tip: You can easily test geolocation services by using the tool that comes with the emulator. Simply click a position on the map and the related coordinates will be sent to the emulator.

Background tracking

Windows Phone 8 has introduced the ability to keep tracking the user's position when the application is suspended. This means that the **PositionChanged** event handler will continue to be invoked even if the app is not running in the foreground.

Background tracking can continue unless:

- The application stops tracking the user's location by deregistering the **PositionChanged** and **StatusChanged** event handlers.
- The application has been running in the background for four hours without being reopened.
- Battery Saver mode is enabled.
- The phone is running out of free memory.
- Geolocation services have been disabled.
- The user opens another application that is able to track his or her location in the background.

To activate background tracking, you need to edit the manifest file using the manual editor (right-click the file and choose **View code**) since the option is not supported by the visual editor. You'll have to edit the **DefaultTask** node in the following way:

```
<Tasks>
  <DefaultTask Name="_default" NavigationPage="MainPage.xaml">
    <BackgroundExecution>
      <ExecutionType Name="LocationTracking"/>
    </BackgroundExecution>
  </DefaultTask>
</Tasks>
```

Now, if the previous conditions are satisfied, the application will keep tracking the user's location even when it's not running in the foreground.

If you want to customize the user experience based on whether the application is running in the background, you can subscribe to a specific event of the **PhoneApplicationService** object, which is declared in the **App.xaml** file, and which we learned to use in Chapter 3. The event handler is called **RunningInBackground** and it's triggered every time the application is suspended. However, since it's using geolocation services, it will keep running in the background.

In the following sample, you can see how the **PhoneApplicationService** declaration will look after we've subscribed to the event:

```
<Application.ApplicationLifetimeObjects>
    <!--Required object that handles lifetime events for the application-->
    <shell:PhoneApplicationService
        Launching="Application_Launching" Closing="Application_Closing"
        Activated="Application_Activated"
Deactivated="Application_Deactivated"
        RunningInBackground="Application_RunningInBackground"
        />
</Application.ApplicationLifetimeObjects>
```

After you've subscribed to the event, you'll be able to manage it in the **App.xaml.cs** and, for example, set a global property that can be used in different pages to identify the current execution status. For example, an application can keep tracking the user's location and displaying it on a map, but if the app is suspended developers can choose to use another way to notify the user's location since the map isn't visible anymore, such as sending a notification or updating the application's Tile.

```
public static bool IsRunningInBackground { get; set; }

private void Application_RunningInBackground(object sender,
RunningInBackgroundEventArgs e)
{
    IsRunningInBackground = true;
}
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    IsRunningInBackground = false;
}
```

In the previous sample, we set a property (called **IsRunningInBackground**) to know whether the app is running in the background. We set it to true when the **RunningInBackground** event is triggered, and set it to false when the **Activated** event is raised, which means that the application has been reopened.

Interacting with the Map control

Windows Phone includes a built-in **Map** control that can be used to embed a map inside an application's page and can be used in combination with the geolocation services. The control has been greatly improved since Windows Phone 7, and it's now based on Nokia's cartography. In addition, it supports offline maps—if the user has downloaded maps for the current location, the control will be able to automatically use them. The **Map** control is part of the **Microsoft.Phone.Maps.Controls** namespace, so you'll need to add it to your XAML page before using it:

```
xmlns:maps="clr-
namespace:Microsoft.Phone.Maps.Controls;assembly=Microsoft.Phone.Maps"
```

The **Map** control exposes many properties for customization. The most useful ones are:

- **Center**, the geolocation coordinates the map is centered at.
- **ZoomLevel**, which is the zoom level from **1** (minimum) to **19** (maximum).
- **CartographicMode**, which can be used to switch between **Aerial** (satellite view), **Road** (road view), **Terrain** (landscape view), and **Hybrid** (a combination of the others).
- **ColorMode** can be used to set a **Light** or **Dark** theme, according to the luminosity conditions.

In the following sample, you can see a **Map** control included in a page:

```
<maps:Map x:Name="CustomMap"
            ZoomLevel="15"
            CartographicMode="Aerial"
            ColorMode="Light" />
```

Figure 19: The Map Control

Using the **Map** control in combination with the geolocation services should be easy: it would be enough to set the **Center** property of the Map with a **Geoposition** object returned by the **Geolocator** class. Unfortunately, there's a limitation since Windows Phone and the **Map** control use two different classes to store geolocation coordinates. The first one is called **Geocoordinate** and it's part of the **Windows.Devices.Geolocation** namespace, while the second one is called **GeoCoordinate** (with a capital C) and it's part of the **System.Device.Location** namespace.

Fortunately, there's a work-around: installing the Windows Phone Toolkit we discussed in Chapter 2. In addition to providing a useful set of additional controls, it also offers many helpers and extensions that are useful when working with the **Map** control.

Specifically, after adding the Windows Phone Toolkit to your project (the easiest way is using NuGet), you will be able to use an extension method called **ToGeoCoordinate()**, which is able to convert the original Windows Runtime class to the **Map** control's specific one. In the following sample, you can see how we use it to display the user's current location on a map:

```
private async void OnGetSinglePositionClicked(object sender, RoutedEventArgs e)
{
```

```
    Geolocator geolocator = new Geolocator();
    if (geolocator.LocationStatus != PositionStatus.Disabled)
    {
        Geoposition geoposition = await geolocator.GetGeopositionAsync();
        myMap.Center = geoposition.Coordinate.ToGeoCoordinate();
    }
}
```

Layers

The **Map** control supports layers that can be added on top of the map. A layer is a collection of visual objects that are displayed over the map, and it's represented by the **MapLayer** class.

Each **MapLayer** has a collection of overlays (the **MapOverlay** class); each one is an object that is displayed on the map.

An overlay can be composed of virtually any control. In fact, it's defined by a position using the **GeoCoordinate** property, and content using the **Content** property, which is a generic object. This means that you can add any object you want as content, as with any of the XAML visual controls available in the framework.

In the following sample, we create a **Rectangle** object and set it as **Content** of a **MapOverlay** object that we add to a new layer.

```
private void OnAddShapeClicked(object sender, RoutedEventArgs e)
{
    MapOverlay overlay = new MapOverlay
    {
        GeoCoordinate = myMap.Center,
        Content = new Rectangle
        {
            Fill = new SolidColorBrush(Colors.Blue),
            Width = 40,
            Height = 40
        }
    };
    MapLayer layer = new MapLayer();
    layer.Add(overlay);

    myMap.Layers.Add(layer);
}
```

Figure 20: A Layer Displayed over a Map Control

Routing

A common scenario in which you work with the **`Map`** control is with routing, where you have the ability to show routes on the map. Even if it sounds complicated, it's easy to implement using the **`RouteQuery`** class, which allows you to:

- Set different route options by using the **`TravelMode`** and **`RouteOptimization`** properties.
- Add a list of waypoints that compose the route.

The following code demonstrates a routing sample:

```
private void OnCalculateRouteClicked(object sender, RoutedEventArgs e)
{
    RouteQuery query = new RouteQuery
    {
        TravelMode = TravelMode.Driving,
```

```
        RouteOptimization = RouteOptimization.MinimizeTime,
    };

    List<GeoCoordinate> coordinates = new List<GeoCoordinate>();
    coordinates.Add(new GeoCoordinate(47.6045697927475, -
122.329885661602));
    coordinates.Add(new GeoCoordinate(47.605712890625, -122.330268859863));
    query.Waypoints = coordinates;

    query.QueryCompleted += query_QueryCompleted;
    query.QueryAsync();
}
```

The **TravelMode** and **RouteOptimization** options can be used to customize the route in a way similar to what many GPS navigators do. In the previous sample, we want the driving route that requires the least amount of time.

The path is set using the **Waypoints** property, which requires a collection of **GeoCoordinate** objects. Each object represents a point in the path that should be touched by the route.

The **RouteQuery** class works using the callback approach. We call the **QueryAsync()** method and subscribe to the **QueryCompleted** event, which is triggered when the route has been calculated, as shown in the following sample:

```
void query_QueryCompleted(object sender, QueryCompletedEventArgs<Route> e)
{
    MapRoute route = new MapRoute(e.Result);
    myMap.AddRoute(route);
}
```

Displaying a route on the **Map** control is easy—you simply have to create a new **MapRoute** object, passing the query result (stored in the **Result** property of the returned object) as a parameter, and add it to the **Map** using the **AddRoute()** method.

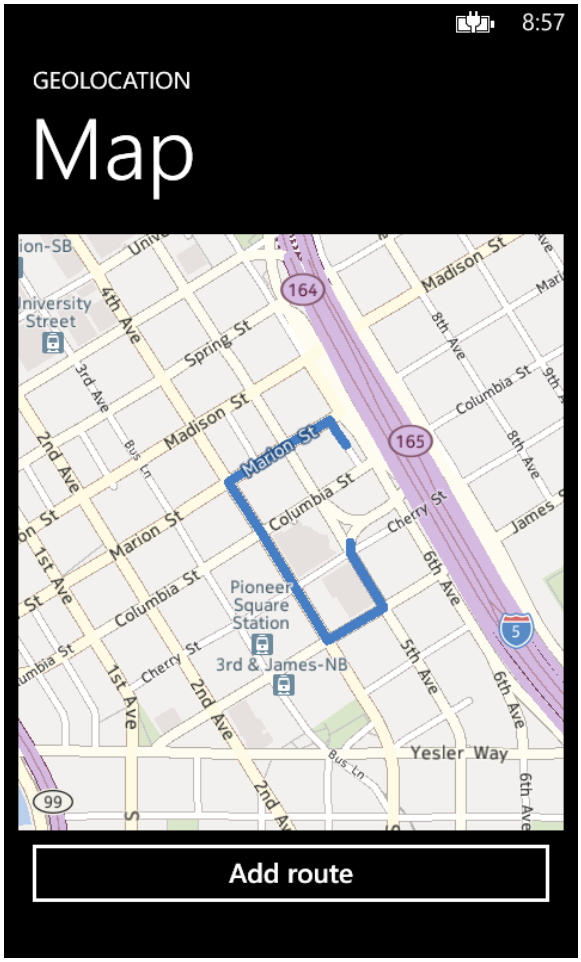

Figure 21: A Route Displayed on a Map

Working with coordinates

Until now, we've always worked with geolocation coordinates based on latitude and longitude, but often it's easier for users to understand a location based on its civic address. Windows Phone 8 has introduced two classes to perform geocoding conversion: **`GeoCodeQuery`** (to convert an address into a set of numeric coordinates) and **`ReverseGeocodeQuery`** (to convert latitude and longitude into an address).

They both work in the same way, since they use the same callback approach we've seen for routing. After you've defined the operation to perform, you can start the search using the **`QueryAsync()`** method. Once the search has completed, you can use the **`QueryCompleted`** event handler to manage the results.

The **`GeocodeQuery`** class requires you to set two parameters: **`GeoCoordinate`** and, most important, **`SearchTerm`**, which is the search keyword. **`GeoCoordinate`** shouldn't be required (since the purpose of this class is to find the location's coordinate), but you'll have to set it anyway with a fake value as shown in the following sample. Otherwise, you won't get any result.

```
private void OnResolveCoordinatesClicked(object sender, RoutedEventArgs e)
{
    GeocodeQuery query = new GeocodeQuery
        {
               GeoCoordinate = new GeoCoordinate(0, 0),
               SearchTerm = "Milan,Italy"
        };
    query.QueryCompleted += query_QueryCompleted;
    query.QueryAsync();
}
```

The **ReverseGeoCodeQuery** class, instead, only requires the **GeoCoordinate** property to be set with the location's coordinates.

```
private void OnResolveAddressClicked(object sender, RoutedEventArgs e)
{
    ReverseGeocodeQuery reverseQuery = new ReverseGeocodeQuery
        {
             GeoCoordinate = new GeoCoordinate(45.3967, 9.3163)
        };
    reverseQuery.QueryCompleted += reverseQuery_QueryCompleted;
    reverseQuery.QueryAsync();
}
```

The **QueryCompleted** event handler is the same for both classes and returns a collection of **MapLocation** objects. If you used the **GeocodeQuery** class, you'll probably be interested in the **GeoCoordinate** object, which contains the latitude and longitude of the searched location.

```
void query_QueryCompleted(object sender,
QueryCompletedEventArgs<IList<MapLocation>> e)
{
    var item = e.Result.FirstOrDefault();
    myMap.SetView(item.BoundingBox, MapAnimationKind.Parabolic);
}
```

The previous sample shows a way to center the map on the position returned by the service.

Instead, if you used the **ReverseGeocodeQuery** class, you'll find the information you're looking for in the **Information** property that contains data like **City**, **Street**, **Address**, etc.

```
void reverseQuery_QueryCompleted(object sender,
QueryCompletedEventArgs<IList<MapLocation>> e)
{
```

```
    var item = e.Result.FirstOrDefault();
    MessageBox.Show(string.Format("{0},{1}",
item.Information.Address.Street, item.Information.Address.City));
}
```

In both cases, the event handler returns a collection of **MapLocation** objects because, especially if you searched a location by a keyword, the service can return multiple results. The previous example shows the information about the first item in the collection.

How to publish an application that uses the Map control

You can freely use the **Map** control during the testing phase, but when you submit it to the Store, you'll need production credentials.

The credentials can be obtained during the submission process. One of the optional steps is called **Map service**, which will give you two codes called **ApplicationId** and **AuthenticationToken**. Once you have the codes, you need to set them in the following way when the application starts:

```
Microsoft.Phone.Maps.MapsSettings.ApplicationContext.ApplicationId =
"ApplicationId";
Microsoft.Phone.Maps.MapsSettings.ApplicationContext.AuthenticationToken =
"AuthenticationToken";
```

Movement sensors

Windows Phone devices have many movement sensors that can be used by applications, like the accelerometer, gyroscope, and compass. The Windows Runtime has introduced a new set of APIs which are part of the **Windows.Devices.Sensors** namespace:

- The **Accelerometer** class can be used to interact with the accelerometer.
- The **Gyrometer** class can be used to interact with the gyroscope.
- The **Compass** class can be used to interact with the compass.
- **OrientationSensor** is a special class that can combine values obtained from all the available sensors.

Note: To use sensors, you'll need to enable the ID_CAP_SENSORS option in the manifest file. If you also want to use gyroscope and compass, you need to enable the ID_REQ_MAGNETOMETER and ID_REQ_GYROSCOPE capabilities in the Requirements section of the manifest file. This way, users with devices without one of these sensors won't be able to download your application.

All the sensors work in the same way. You'll be able to get a reference to the sensor using the **`GetDefault()`** method. If it's not available on the phone (for example, not all devices have a gyroscope), you'll get a **`null`** reference in return. It's important to always check if the returned sensor is **`null`** before doing any operation.

Like the geolocation services, you have two ways to interact with sensors:

- The **`GetCurrentReading()`** method, which returns a single detection.
- The **`ReadingChanged`** event handler, which is triggered every time the phone moves to a new position.

In this section, we'll use as an example the **`OrientationSensor`** class, which is a special sensor that is able to combine all the values returned by the available sensors and automatically filter all the out-of-scale data. The class returns an **`OrientationSensorReading`** object, which contains all the information about the current position. You'll be able to get the device position by using the **`Quaternion`** and **`RotationMatrix`** properties. In the following samples, you can see two ways to achieve the same result: getting a single reading, and subscribing to notifications that are sent every time the position changes. The device's coordinates on the x-axis, y-axis, and z-axis are displayed on the screen using three **`TextBlock`** controls:

```
//Single reading.
private void OnGetReadingClicked(object sender, RoutedEventArgs e)
{
    OrientationSensor orientationSensor = OrientationSensor.GetDefault();
    if (orientationSensor != null)
    {
        OrientationSensorReading reading =
orientationSensor.GetCurrentReading();
        txtX.Text = reading.Quaternion.X.ToString();
        txtY.Text = reading.Quaternion.Y.ToString();
        txtZ.Text = reading.Quaternion.Z.ToString();
    }
    else
    {
        MessageBox.Show("The sensor is not available");
    }
}
//Continuous reading.
private void OnGetReadingClicked(object sender, RoutedEventArgs e)
{
    OrientationSensor orientationSensor = OrientationSensor.GetDefault();
    if (orientationSensor != null)
    {
        orientationSensor.ReadingChanged +=
orientationSensor_ReadingChanged;
    }
    else
```

```
    {
        MessageBox.Show("The sensor is not available");
    }
}

void orientationSensor_ReadingChanged(OrientationSensor sender,
OrientationSensorReadingChangedEventArgs args)
{
    Dispatcher.BeginInvoke(() =>
    {
        txtX.Text = args.Reading.Quaternion.X.ToString();
        txtY.Text = args.Reading.Quaternion.Y.ToString();
        txtZ.Text = args.Reading.Quaternion.Z.ToString();
    });
}
```

Please note that if you decide to subscribe to the **ReadingChanged** event, you'll need a **Dispatcher** to communicate with the user interface since the event handler is managed by a background thread.

If you need to use a specific sensor, the code to use is very similar. You'll simply have to use the specific sensor class and manage the specific reading object that you'll get in return.

Tip: The Windows Phone emulator features a tool to simulate movement sensors. Unfortunately, only the accelerometer is supported; for every other sensor, including the OrientationSensor, you'll need a real device.

Determining the current hardware

Windows Phone offers a class called **DeviceStatus** that can be used to get information about the current device, like:

- The firmware version, with the **DeviceFirmwareVersion** property.
- The hardware version, with the **DeviceHardwareVersion** property.
- The manufacturer, with the **DeviceManufacturer** property.
- The device name, with the **DeviceName** property.
- The amount of total memory available, with the **DeviceTotalMemory** property.

Plus, you have access to some useful APIs to get the current status of the battery. They belong to the **Windows.Phone.Devices.Power** namespace, and you'll be able to use the **Battery** class to identify the percentage of remaining battery charge (with the **RemainingChargePercent** property) and the remaining time before the battery totally discharges (with the **RemainingDischargeTime** property).

The **Battery** class behaves like a sensor; you'll have to use the **GetDefault()** method to get a reference to it (even if, in this case, you can avoid checking whether the returned object is **null** since every phone has a battery), as in the following sample:

```
private void OnGetBatteryClicked(object sender, RoutedEventArgs e)
{
    int remainingCharge = Battery.GetDefault().RemainingChargePercent;
    TimeSpan remainingTime = Battery.GetDefault().RemainingDischargeTime;
}
```

In addition, the **DeviceStatus** class offers a property called **PowerSource**, which tells you the current power source, and an event handler, called **PowerSourceChanged**, which is triggered every time the current power source changes (from battery to external or vice versa).

```
private void OnGetBatteryClicked(object sender, RoutedEventArgs e)
{
    DeviceStatus.PowerSourceChanged += DeviceStatus_PowerSourceChanged;
}

void DeviceStatus_PowerSourceChanged(object sender, EventArgs e)
{
    string message = DeviceStatus.PowerSource == PowerSource.Battery ?
"Batteria" : "Alimentazione di rete";
    MessageBox.Show(message);
}
```

It can be useful if, for example, you want to avoid performing power-consuming operations if the phone is not connected to an external power source.

Note: To get access to hardware info, you'll need to enable the ID_CAP_IDENTITY_DEVICE capability in the manifest file.

Proximity

Under the proximity category we can include all the new APIs that have been introduced in the Windows Runtime to connect two devices together without using an Internet connection. In Windows Phone you can achieve this result by using two technologies: Bluetooth and NFC.

Bluetooth is well known and can be used to connect devices within a range of 10 meters. It's been available since the first Windows Phone release, but only Windows Phone 8 has introduced APIs that are available to developers.

NFC is a more recent technology that has started to gain some traction in recent years. It can be used to exchange small amounts of data within a close range (the two devices should basically touch each other). NFC is an interesting technology since it works not only with active devices (like two phones), but also with passive devices (like chips embedded in a sticker or in a magazine page). In addition, Windows Phone is able to extend NFC and use it also to create a Bluetooth communication channel without needing to manually pair the two devices. This way, you can overcome NFC's limitations and use Bluetooth to transfer larger data files, like images.

Note: To use Proximity APIs, you'll need to enable the ID_CAP_PROXIMITY option in the manifest file.

The easiest way to test applications that use the Proximity APIs is with real devices, but there's also a third-party tool called Proximity Tapper available on CodePlex that can be used to simulate the connection between different emulators (since Visual Studio is able to run only one specific emulator at a time, you'll have to use different emulator versions, for example a WVGA and a WXGA one).

Exchanging messages

A common scenario when working with NFC is the exchange of messages, which represents a small amount of data. There are some standard messages that Windows Phone is able to manage automatically (for example, when you receive a URI or a contact), and some custom messages which can be managed only by third-party apps.

The first step, as withr every other sensor we've seen so far, is to use the **`GetDefault()`** method of the **`ProximityDevice`** class to get access to the proximity sensor. In this case, we also need to check if the sensor reference is **`null`** before moving on, since some devices don't support NFC.

Every message is identified by a specific keyword. The Windows Phone APIs natively support three message types—text, URI, and binary. Let's see how to manage them.

Text messages

Publishing a text message is easy. We use the **`PublishMessage()`** method of the **`ProximityDevice`** class, which requires the message type and the content as parameters. In the following sample, you can see how we send a text message, identified by the **`Windows.SampleMessage`** keyword.

```
private void OnSendMessageClicked(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();

    if (device != null)
    {
```

```
        device.PublishMessage("Windows.SampleMessage", "Sample message",
MessageSent);
    }
}

private void MessageSent(ProximityDevice sender, long messageId)
{
    MessageBox.Show("The message has been sent");
    sender.StopPublishingMessage(messageId);
}
```

As you can see, the **PublishMessage()** method accepts a third, optional parameter, which is an event that is raised when the message has been received by the other device. This event can be useful, as shown in the previous sample, to stop sending the message once it has been received by calling the **StopPublishingMessage()** method on the **ProximityDevice** object. You need to set the message ID, which is passed as a parameter of the method.

The phone receiving the message should instead call the **SubscribeForMessage()** method. Unlike the publishing method, this method is the same regardless of the data we expect. The difference is that, according to the message, we can use some specific properties to parse it and extract the information we need.

In the following sample you can see how it's easy to extract the message content thanks to the **DataAsString** property of the **ProximityMessage** class:

```
private void OnReadMessageClicked(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();

    if (device != null)
    {
        device.SubscribeForMessage("Windows.SampleMessage",
messageReceived);
    }
}

private void messageReceived(ProximityDevice sender, ProximityMessage
message)
{
    Dispatcher.BeginInvoke(() =>
                                {
                                    MessageBox.Show(message.DataAsString);
                                });
    sender.StopSubscribingForMessage(message.SubscriptionId);
}
```

This code is not very different from the code used to send the message; the **SubscribeForMessage()** method requires the message type and the event handler that is invoked when the message has been received.

The message is received thanks to the **ProximityMessage** object that is returned as a parameter. In this case, since it's a text message, we can extract the content using the **DataAsString** property. Note that again in this situation we cancel the subscription by using the **StopSubscribingForMessage()** method so that the application won't listen anymore for the incoming message.

URI

Sending URIs works in a similar way, except that we need to use the **PublishUriMessage()** method that accepts the **Uri** to send as a parameter. In this case, we don't need to set the message type since it's implicit.

```
private void OnPublishUriClicked(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();
    if (device != null)
    {
        device.PublishUriMessage(new Uri("http://wp.qmatteoq.com"));
    }
}
```

The difference is that **Uri** messages are directly supported by Windows Phone, so you'll be able to exchange them without using a receiver application. The operating system, once it has received the **Uri**, will simply ask the user if he or she wants to open the browser to see it.

However, if you still want to manage **Uri**s in your application, you can subscribe to receive them. In this case, you'll have to listen for the incoming **WindowsUri** message. In addition, you'll need some more code to extract it since it's not treated as a **string**; you'll need to work directly with a byte array, as in the following sample:

```
private void OnReceiveUriClicked(object o, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();
    if (device != null)
    {
        device.SubscribeForMessage("WindowsUri", messageReceived);
    }
}

private void messageReceived(ProximityDevice sender, ProximityMessage
message)
{
    byte[] array = message.Data.ToArray();
```

```
    string uri = Encoding.Unicode.GetString(array, 0, array.Length);
    Dispatcher.BeginInvoke(() =>
    {
        MessageBox.Show(uri);
    });
}
```

NDEF messages

NDEF (NFC Data Exchange Format) is a standard protocol used to define NFC messages that can be used to exchange different data types across different platforms. Some common scenarios are exchanging geolocation data, mail sending, and social network sharing.

NDEF isn't natively supported by Windows Phone APIs. To avoid requiring developers to manually create the binary data needed to compose a message, a developer named Andreas Jakl has created a library called **NDEF Library For Proximity API**, which is available on CodePlex and can be easily installed using NuGet.

This library features many classes that encapsulate the needed logic for the most common messages, like **`NdefGeoRecord`** for sharing geolocation coordinates, **`NdefLaunchAppRecord`** to open another Windows Phone application, and **`NdefTelRecord`** to start a phone call to a specific number. You can see a complete list of the supported records on the official website.

In the following sample, you'll see how to send a message that contains an **`NdefSmsRecord`** which can be used to create a new SMS message with predefined text:

```
private void OnSendNdefMessageClicked(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();
    if (device != null)
    {
        NdefSmsRecord record = new NdefSmsRecord
                                    {
                                        SmsNumber = "0123456789",
                                        SmsBody = "This is the text"
                                    };

        NdefMessage message = new NdefMessage
                                    {
                                        record
                                    };

        device.PublishBinaryMessage("NDEF",
message.ToByteArray().AsBuffer());
    }
}
```

The first step is to create the record and set the needed properties. In this sample, we need to set the **SmsNumber** (the phone number that will receive the message) and **SmsBody** (the message's text) properties.

Next, we need to encapsulate the record into a new message, which is identified by the **NdefMessage** class. In the end, we can send it; in this scenario we need to use the **PublishBinaryMessage()** method of the **ProximityDevice** class, since it's not a standard message but a binary one. As you can see, we pass as parameter (other than the message's type, which is **NDEF**) the message as a bytes array.

If the message's type is natively supported by Windows Phone, the device will automatically manage the incoming message. With the previous sample, the operating system will prompt to the user to send a SMS message. Instead, if we want to receive it within an application, we'll need to do some additional work. Since it's a binary message, we'll need to extract the needed information. Fortunately, the NDEF Library for Proximity API will help us, as you can see in the following sample:

```
private void OnReceiveNdefMessage(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();
    if (device != null)
    {
        device.SubscribeForMessage("NDEF", NdefMessageReceived);
    }
}

private void NdefMessageReceived(ProximityDevice sender, ProximityMessage
message)
{
    NdefMessage receivedMessage =
NdefMessage.FromByteArray(message.Data.ToArray());
    foreach (NdefRecord record in receivedMessage)
    {
        if (record.CheckSpecializedType(true) == typeof(NdefSmsRecord
        {
            NdefSmsRecord ndefSmsRecord = new NdefSmsRecord(record);
            Dispatcher.BeginInvoke(() =>
            {
                MessageBox.Show(ndefSmsRecord.SmsBody);
            });
        }
    }
}
```

We need to subscribe for incoming NDEF messages. When we receive the message, we are able to convert the binary data (which is available in the **Data** property of the **ProximityMessage** class) into a **NdefMessage** again, thanks to the **FromByteArray()** method.

An **NdefMessage** object can contain more than one record, so we need to iterate it and extract every **NdefRecord** object that is stored. In the previous sample, since we expect to only get an **NdefSmsRecord**, we manage only this scenario. The task is accomplished by using the **CheckSpecializedType()** method on the record, which returns its data type. We manage it only if it's the type we're expecting. We are able to get the original record simply by creating a new **NdefSmsRecord** object and passing, as a parameter, the **NdefRecord** object stored in the message.

Once we have it, we are able to access all its properties. In the previous sample, we show the user the body of the SMS message.

Writing messages to a tag

As mentioned previously, NFC can be used with passive devices, like tags and stickers. Windows Phone devices are able to write data to NFC tags simply by adding the **:WriteTag** suffix to the message's type when publishing a message.

In the following code you can see how to adapt the previous sample to write the record into a tag instead of sending it to another device:

```
private void OnSendNdefMessageClicked(object sender, RoutedEventArgs e)
{
    ProximityDevice device = ProximityDevice.GetDefault();
    if (device != null)
    {
        NdefSmsRecord record = new NdefSmsRecord
                                    {
                                        SmsNumber = "0123456789",
                                        SmsBody = "This is the text"
                                    };

        NdefMessage message = new NdefMessage
                                    {
                                        record
                                    };

        device.PublishBinaryMessage("NDEF:WriteTag",
message.ToByteArray().AsBuffer());
    }
}
```

Creating a communication channel using NFC

As previously mentioned, NFC can be used only to exchange small amounts of data—it can't be used to maintain a stable communication channel. The best technology to achieve this result is Bluetooth. However, we're able to use NFC as a shortcut to establish a Bluetooth channel between two devices that have the same installed application.

Note: In this case we also need to enable the ID_CAP_NETWORKING option in the manifest file.

The starting point is the **`PeerFinder`** class, which can be used (on both devices) to start the connection and look for another device to pair with. Using it is quite simple: you have to subscribe to the **`TriggeredConnectionStateChanged`** event that is triggered when the connection status changes, and start the pairing process by calling the **`Start()`** method.

```
private async void OnConnectToPeerClicked(object sender, RoutedEventArgs e)
{
    PeerFinder.TriggeredConnectionStateChanged +=
PeerFinder_TriggeredConnectionStateChanged;
    PeerFinder.Start();
}
```

You'll need to use the same code on both devices. However, there's a work-around to automatically execute it in case the app is opened by a pairing request. In fact, when you execute the previous code and connect the two devices together, Windows Phone will automatically intercept the incoming message and will prompt the user to open the required application. When this happens, the application is opened with the following special URI:

```
MainPage.xaml?ms_nfp_launchargs=Windows.Networking.Proximity.PeerFinder:
StreamSocket
```

By using the **`OnNavigatedTo`** event on the main page (covered in Chapter 3) we are able to intercept this special URI and automatically start the pairing process, as shown in the following sample:

The **`TriggeredConnectionStateChanged`** event can be used to manage the actual connection status. The most important state is **`Completed`**, which is triggered when the connection has been successfully established and you can start to exchange data.

In the following sample, you can see that once the connection is established, we store the channel (identified by a **`StreamSocket`** object) in another variable to be used later for further communications.

```
private StreamSocket socket;

void PeerFinder_TriggeredConnectionStateChanged(object sender,
TriggeredConnectionStateChangedEventArgs args)
{
    switch (args.State)
    {
        case TriggeredConnectState.Completed:
            socket = args.Socket;
            StartListeningForMessages();
            PeerFinder.Stop();
            break;
    }
}
```

Other than saving a reference to the channel, we start listening for incoming messages (we'll see how in a moment) and we call the **`Stop()`** method of the **`PeerFinder`** class. Since the channel has been created, we can stop the pairing process.

Listening for incoming message

The listening process works in a way similar to polling; until the channel is opened, we keep asking the other device if there's a new message. In the following sample, we exchange text messages by using the **`DataReader`** and **`DataWriter`** classes we learned to use in Chapter 4 in the context of storing data in the local storage.

```
private bool listening;

private async void StartListeningForMessages()
{
    if (socket != null)
    {
        if (!listening)
        {
            listening = true;
            while (listening)
            {
                var message = await GetMessage();
                if (listening)
                {
                    if (message != null)
                    {
                        Dispatcher.BeginInvoke(() =>
MessageBox.Show(message));
                    }
                }
            }
```

```
        }
    }
}


private async Task<string> GetMessage()
{
    DataReader dataReader = new DataReader(socket.InputStream);
    uint bytesRead = await dataReader.LoadAsync(sizeof(uint));
    if (bytesRead > 0)
    {
        uint strLength = (uint)dataReader.ReadUInt32();
        bytesRead = await dataReader.LoadAsync(strLength);
        if (bytesRead > 0)
        {
            String message = dataReader.ReadString(strLength);
            return message;
        }
    }

    return string.Empty;
}
```

While the channel is opened and the listening process is active, we keep calling the **GetMessage()** method. If there's an incoming message in the channel, we display it to the user. The polling procedure is implemented using a **while** loop, which is repeated until the **isListening** variable is set to **true**.

The **GetMessage()** method is simply a helper that, by using the **DataReader** class, is able to get the channel's data (which is stored as binary in the **InputStream** property of the **StreamSocket** class) and convert it into a plain string.

Sending a message

To send a message, we need to use the **DataWriter** class to write the data in the **OutputStream** channel of the **StreamSocket** object. We have to send two pieces of information: the message size by using the **WriteInt32()** method, and the message text by using the **WriteString()** message.

```
public async Task SendMessage(string message)
{
    if (socket != null)
    {
        DataWriter dataWriter = new DataWriter(socket.OutputStream);

        dataWriter.WriteInt32(message.Length);
        await dataWriter.StoreAsync();

        dataWriter.WriteString(message);
```

```
        await dataWriter.StoreAsync();
    }
}
```

If you want to send a message on the channel, it's enough to use the **SendMessage()** method we've just defined:

```
private async void OnSendMessageOnChannelClicked(object sender,
RoutedEventArgs e)
{
    await SendMessage("This is my first message");
}
```

Creating a communication channel using Bluetooth

The approach to creating a communication channel using Bluetooth is the same we've previously seen. A channel is identified by the **StreamSocket** class, and we can send and listen for incoming messages exactly in the same way.

What changes is the way we connect to another device. With NFC, the peer-to-peer communication is necessarily made between the two devices that are connected. With Bluetooth, instead, you can virtually connect to any device within your phone's range. We're going to use the **PeerFinder** class again, but this time, instead of starting the pairing process using the **Start()** method, we look for all the available peers with the **FindAllPeersAsync()** method. It returns a collection of **PeerInformation** objects—each one is a device that can connect with our application.

In the following sample, we simply display the list of available devices to the user, by setting the collection as **ItemsSource** of a **ListBox** control:

```
private async void OnFindNearPeers(object sender, RoutedEventArgs e)
{
    PeerFinder.Start();
    try
    {
        IReadOnlyList<PeerInformation> peers = await
PeerFinder.FindAllPeersAsync();
        PeersList.ItemsSource = peers;
    }
    catch (Exception exc)
    {
        if ((uint)exc.HResult == 0x8007048F)
        {
            MessageBox.Show("Bluetooth is turned off");
        }
```

```
    }
}
```

Notice that we've embedded our code into a **try/catch** statement; in fact, the user may have Bluetooth turned off. While we start looking for other peers, if we get an exception with the error code **0x8008048F**, it means that we are in this situation, so we have to properly manage it (for example, by informing the user that he or she needs to turn it on to use the application).

After the user has chosen which device he or she wants to connect with, we need to call the **ConnectAsync()** method of the **PeerFinder** class, passing the **PeerInformation** object that represents the device as a parameter. Next, exactly like we did for the NFC communication, we start listening for messages and stop looking for other peers using the **Stop()** method, as shown in the following sample:

```
private async void PeerList_OnSelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    PeerInformation peer = PeerList.SelectedItem as PeerInformation;
    socket = await PeerFinder.ConnectAsync(peer);
    StartListeningForMessages();
    PeerFinder.Stop();
}
```

The **StartListeningForMessage()** method is the same method we used to deal with NFC communication.

On the other side (the phone that the user has selected to interact with), if we want to accept the incoming communication request from another phone, we need to again use the **Start()** method of the **PeerFinder** class and subscribe to the **ConnectionRequested** event, which is triggered when another device has requested to establish a connection.

In the event handler we simply have to call the **ConnectAsync()** method of the **PeerFinder** class like we did with the phone that started the connection request. We'll get a reference to the device that sent the request in the method's parameters.

```
private void OnListenToConnectionClicked(object sender, RoutedEventArgs e)
{
    PeerFinder.ConnectionRequested += PeerFinder_ConnectionRequested;
    PeerFinder.Start();
}

async void PeerFinder_ConnectionRequested(object sender,
ConnectionRequestedEventArgs args)
{
    socket = await PeerFinder.ConnectAsync(args.PeerInformation);
    StartListeningForMessages();
    PeerFinder.Stop();
}
```

A quick recap

In this chapter we've seen how to interact with the hardware features that every Windows Phone device has. Specifically:

- We've seen how to interact with the geolocation services, retrieve the user's location, and interact with the `Map` control.
- We've learned how to use the motion sensors to determine the device's position in space. It's a feature particularly useful with games, since many of them are easier to control with the accelerometer rather than with virtual controls.
- We've briefly covered the hardware APIs that are useful for obtaining information about the device on which the app is running.
- We've discussed Proximity APIs, which have been introduced in the Windows Runtime to connect two devices without requiring an Internet connection. Specifically, we've talked about two technologies: NFC and Bluetooth.

Chapter 7 Integrating with the Operating System

Launchers and choosers

When we discussed storage in Chapter 4, we introduced the concept of isolated applications. In the same way that storage is isolated such that you can't access the data stored by another application, the application itself is isolated from the operating system. The biggest benefit of this approach is security. Even if a malicious application is able to pass the certification process, it won't have the chance to do much damage because it doesn't have direct access to the operating system. But sooner or later, you'll need to interact with one of the many Windows Phone features, like sending a message, making a phone call, playing a song, etc.

For all these scenarios, the framework has introduced launchers and choosers, which are sets of APIs that demand a specific task from the operating system. Once the task is completed, control is returned to the application.

Launchers are "fire and forget" APIs. You demand the operation and don't expect anything in return—for example, starting a phone call or playing a video.

Choosers are used to get data from a native application—for example, contacts from the People Hub—and import it into your app.

All the launchers and choosers are available in the **`Microsoft.Phone.Tasks`** namespace and share the same behavior:

- Every launcher and chooser is represented by a specific class.
- If needed, you set some properties that are used to define the launcher or chooser's settings.
- With a chooser, you'll need to subscribe to the **`Completed`** event, which is triggered when the operation is completed.
- The **`Show()`** method is called to execute the task.

Note: Launchers and choosers can't be used to override the built-in Windows Phone security mechanism, so you won't be able to execute operations without explicit permission from the user.

In the following sample, you can see a launcher that sends an email using the **`EmailComposeTask`** class:

```
private void OnComposeMailClicked(object sender, RoutedEventArgs e)
{
    EmailComposeTask mailTask = new EmailComposeTask();
```

```
    mailTask.To = "mail@domain.com";
    mailTask.Cc = "mail2@domain.com";
    mailTask.Subject = "Subject";
    mailTask.Body = "Body";

    mailTask.Show();
}
```

The following sample demonstrates how to use a chooser. We're going to save a new contact in the People Hub using the **SaveContactTask** class.

```
private void OnSaveContactClicked(object sender, RoutedEventArgs e)
{
    SaveContactTask task = new SaveContactTask();
    task.Completed += task_Completed;

    task.FirstName = "John";
    task.LastName = "Doe";
    task.MobilePhone = "1234567890";

    task.Show();
}

void task_Completed(object sender, SaveContactResult e)
{
    if (e.TaskResult == TaskResult.OK)
    {
        MessageBox.Show("The contact has been saved successfully");
    }
}
```

Every chooser returns a **TaskResult** property, with the status of the operation. It's important to verify that the status is **TaskResult.OK** before moving on, because the user could have canceled the operation.

The following is a list of all the available launchers:

- **MapsDirectionTask** is used to open the native Map application and calculate a path between two places.
- **MapsTask** is used to open the native Map application centered on a specific location.
- **MapDownloaderTask** is used to manage the offline maps support new to Windows Phone 8. With this task, you'll be able to open the Settings page used to manage the downloaded maps.
- **MapUpdaterTask** is used to redirect the user to the specific Settings page to check for offline maps updates.

- **ConnectionSettingsTask** is used to quickly access the different Settings pages to manage the different available connections, like Wi-Fi, cellular, or Bluetooth.
- **EmailComposeTask** is used to prepare an email and send it.
- **MarketplaceDetailTask** is used to display the detail page of an application on the Windows Phone Store. If you don't provide the application ID, it will the open the detail page of the current application.
- **MarketplaceHubTask** is used to open the Store to a specific category.
- **MarketplaceReviewTask** is used to open the page in the Windows Phone Store where the user can leave a review for the current application.
- **MarketplaceSearchTask** is used to start a search for a specific keyword in the Store.
- **MediaPlayerLauncher** is used to play audio or a video using the internal Windows Phone player. It can play both files embedded in the Visual Studio project and those saved in the local storage.
- **PhoneCallTask** is used to start a phone call.
- **ShareLinkTask** is used to share a link on a social network using the Windows Phone embedded social features.
- **ShareStatusTask** is used to share custom status text on a social network.
- **ShareMediaTask** is used to share one of the pictures from the Photos Hub on a social network.
- **SmsComposeTask** is used to prepare a text message and send it.
- **WebBrowserTask** is used to open a URI in Internet Explorer for Windows Phone.
- **SaveAppointmentTask** is used to save an appointment in the native Calendar app.

The following is a list of available choosers:

- **AddressChooserTask** is used to import a contact's address.
- **CameraCaptureTask** is used to take a picture with the integrated camera and import it into the application.
- **EmailAddressChooserTask** is used to import a contact's email address.
- **PhoneNumberChooserTask** is used to import a contact's phone number.
- **PhotoChooserTask** is used to import a photo from the Photos Hub.
- **SaveContactTask** is used to save a new contact in the People Hub. The chooser simply returns whether the operation completed successfully.
- **SaveEmailAddressTask** is used to add a new email address to an existing or new contact. The chooser simply returns whether the operation completed successfully.
- **SavePhoneNumberTask** is used to add a new phone number to an existing contact. The chooser simply returns whether the operation completed successfully.
- **SaveRingtoneTask** is used to save a new ringtone (which can be part of the project or stored in the local storage). It returns whether the operation completed successfully.

Getting contacts and appointments

Launchers already provide a basic way of interacting with the People Hub, but they always require user interaction. They open the People Hub and the user must choose which contact to import.

However, in certain scenarios you need the ability to programmatically retrieve contacts and appointments for the data. Windows Phone 7.5 introduced some new APIs to satisfy this requirement. You just have to keep in mind that, to respect the Windows Phone security constraints, these APIs only work in read-only mode; you'll be able to get data, but not save it (later in this chapter, we'll see that Windows Phone 8 has introduced a way to override this limitation).

In the following table, you can see which data you can access based on where the contacts are saved.

Provider	Contact Name	Contact Picture	Other Information	Calendar Appointments
Device	Yes	Yes	Yes	Yes
Outlook.com	Yes	Yes	Yes	Yes
Exchange	Yes	Yes	Yes	Yes
SIM	Yes	Yes	Yes	No
Facebook	Yes	Yes	Yes	No
Other social networks	No	No	No	No

To know where the data is coming from, you can use the **`Accounts`** property, which is a collection of the accounts where the information is stored .In fact, you can have information for the same data split across different accounts.

Working with contacts

Each contact is represented by the **`Contact`** class, which contains all the information about a contact, like **`DisplayName`**, **`Addresses`**, **`EmailAddresses`**, **`Birthdays`**, etc. (basically, all the information that you can edit when you create a new contact in the People Hub).

Note: To access the contacts, you need to enable the ID_CAP_CONTACTS option in the manifest file.

Interaction with contacts starts with the **Contacts** class which can be used to perform a search by using the **SearchAsync()** method. The method requires two parameters: the keyword and the filter to apply. There are two ways to start a search:

- A generic search: The keyword is not required since you'll simply get all the contacts that match the selected filter. This type of search can be achieved with two filter types: **FilterKind.PinnedToStart** which returns only the contacts that the user has pinned on the Start screen, and **FilterKind.None** which simply returns all the available contacts.
- A search for a specific field: In this case, the search keyword will be applied based on the selected filter. The available filters are **DisplayName**, **EmailAddress**, and **PhoneNumber**.

The **SearchAsync()** method uses a callback approach; when the search is completed, an event called **SearchCompleted** is raised.

In the following sample, you can see a search that looks for all contacts whose name is John. The collection of returned contacts is presented to the user with a **ListBox** control.

```
private void OnStartSearchClicked(object sender, RoutedEventArgs e)
{
    Contacts contacts = new Contacts();
    contacts.SearchCompleted += new
EventHandler<ContactsSearchEventArgs>(contacts_SearchCompleted);
    contacts.SearchAsync("John", FilterKind.DisplayName, null);
}

void contacts_SearchCompleted(object sender, ContactsSearchEventArgs e)
{
    Contacts.ItemsSource = e.Results;
}
```

Tip: If you want to start a search for another field that is not included in the available filters, you'll need to get the list of all available contacts by using the `FilterKind.None` option and apply a filter using a LINQ query. The difference is that built-in filters are optimized for better performance, so make sure to use a LINQ approach only if you need to search for a field other than a name, email address, or phone number.

Working with appointments

Getting data from the calendar works in a very similar way: each appointment is identified by the **Appointment** class, which has properties like **Subject**, **Status**, **Location**, **StartTime**, and **EndTime**.

To interact with the calendar, you'll have to use the **Appointments** class that, like the **Contacts** class, uses a method called **SearchAsync()** to start a search and an event called **SearchCompleted** to return the results. The only two required parameters to perform a search are the start date and the end date. You'll get in return all the appointments within this time frame. Optionally, you can also set a maximum number of results to return or limit the search to a specific account.

In the following sample, we retrieve all the appointments that occur between the current date and the day before, and we display them using a **ListBox** control.

```
private void OnStartSearchClicked(object sender, RoutedEventArgs e)
{
    Appointments calendar = new Appointments();
    calendar.SearchCompleted += calendar_SearchCompleted;
    DateTime start = DateTime.Now.AddMonths(-1);
    DateTime end = DateTime.Now;
    calendar.SearchAsync(start, end, null);
}

void calendar_SearchCompleted(object sender, AppointmentsSearchEventArgs e)
{
    Calendar.ItemsSource = e.Results;
}
```

Tip: The only way to filter the results is by start date and end date. If you need to apply additional filters, you'll have to perform LINQ queries on the results returned by the search operation.

A private contact store for applications

The biggest limitation of the contacts APIs we've seen so far is that we're only able to read data, not write it. There are some situations in which having the ability to add contacts to the People Hub without asking the user's permission is a requirement, such as a social network app that wants to add your friends to your contacts list, or a synchronization client that needs to store information from a third-party cloud service in your contact book.

Windows Phone 8 has introduced a new class called **ContactStore** that represents a private contact book for the application. From the user's point of view, it behaves like a regular contacts source (like Outlook.com, Facebook, or Gmail). The user will be able to see the contacts in the People Hub, mixed with all the other regular contacts. From a developer point of view, the store belongs to the application; you are free to read and write data, but every contact you create will be part of your private contact book, not the phone's contact list. This means that if the app is uninstalled, all the contacts will be lost.

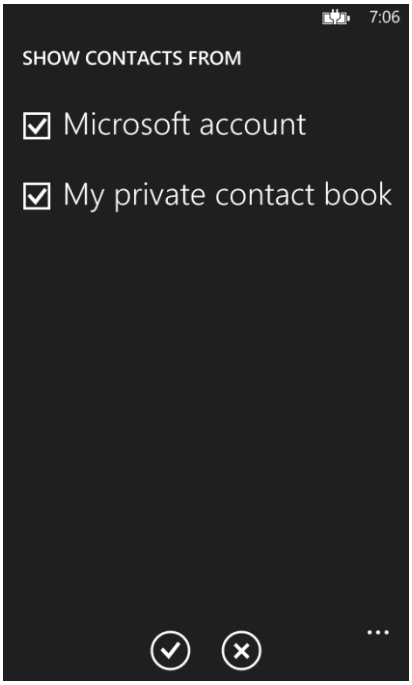

Figure 22: The ContactStore Feature Displayed in the People Hub's Available Accounts

The **ContactStore** class belongs to the **Windows.Phone.PersonalInformation** namespace and it offers a method called **CreateOrOpenAsync()**. The method has to be called every time you need to interact with the private contacts book. If it doesn't exist, it will be created; otherwise, it will simply be opened.

When you create a **ContactStore** you can set how the operating system should provide access to it:

- The first parameter's type is **ContactStoreSystemAccessMode**, and it's used to choose whether the application will only be able to edit contacts that belong to the private store (**ReadOnly**), or the user will also be able to edit information using the People Hub (**ReadWrite**).
- The second parameter's type is **ContactStoreApplicationAccessMode**, and it's used to choose whether other third-party applications will be able to access all the information about our contacts (**ReadOnly**) or only the most important ones, like name and picture (**LimitedReadOnly**).

The following sample shows the code required to create a new private store:

```
private async void OnCreateStoreClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);
}
```

Tip: After you've created a private store, you can't change the permissions you've defined, so you'll always have to call the `CreateOrOpenAsync()` method with the same parameters.

Creating contacts

A contact is defined by the **`StoredContact`** class, which is a bit different from the **`Contact`** class we've previously seen. In this case, the only properties that are directly exposed are **`GivenName`** and **`FamilyName`**. All the other properties can be accessed by calling the **`GetPropertiesAsync()`** method of the **`StoredContact`** class, which returns a collection of type **`Dictionary<string, object>`**.

Every item of the collection is identified by a key (the name of the contact's property) and an object (the value). To help developers access the properties, all the available keys are stored in an enum object named **`KnownContactProperties`**. In the following sample, we use the key **`KnowContactProperties.Email`** to store the user's email address.

```
private async void OnCreateStoreClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);

    StoredContact contact = new StoredContact(store);
    contact.GivenName = "Matteo";
    contact.FamilyName = "Pagani";
    IDictionary<string, object> properties = await
contact.GetPropertiesAsync();
    properties.Add(KnownContactProperties.Email, "info@qmatteoq.com");

    await contact.SaveAsync();
}
```

Tip: Since the `ContactStore` is a dictionary, two values cannot have the same key. Before adding a new property to the contact, you'll have to make sure that it doesn't exist yet; otherwise, you'll need to update the existing one.

The **`StoredContact`** class also supports a way to store custom information by accessing the extended properties using the **`GetExtendedPropertiesAsync()`** method. It works like the standard properties, except that the property key is totally custom. These kind of properties won't be displayed in the People Hub since Windows Phone doesn't know how to deal with them, but they can be used by your application.

In the following sample, we add new custom information called **MVP Category**:

```
private async void OnCreateStoreClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);

    StoredContact contact = new StoredContact(store);
    contact.GivenName = "Matteo";
    contact.FamilyName = "Pagani";

    IDictionary<string, object> extendedProperties = await
contact.GetExtendedPropertiesAsync();
    extendedProperties.Add("MVP Category", "Windows Phone Development");

    await contact.SaveAsync();
}
```

Searching for contacts

Searching contacts in the private contact book is a little tricky because there's no direct way to search a contact for a specific field.

Searches are performed using the **`ContactQueryResult`** class, which is created by calling the **`CreateContactQuery()`** method of the **`ContactStore`** object. The only available operations are **`GetContactsAsync()`**,which returns all the contacts, and **`GetContactCountAsync()`**,which returns the number of available contacts.

You can also define in advance which fields you're going to work with, but you'll still have to use the **`GetPropertiesAsync()`** method to extract the proper values. Let's see how it works in the following sample, in which we look for a contact whose email address is **info@qmatteoq.com**:

```
private async void OnSearchContactClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);
    ContactQueryOptions options = new ContactQueryOptions();
    options.DesiredFields.Add(KnownContactProperties.Email);

    ContactQueryResult result = store.CreateContactQuery(options);
    IReadOnlyList<StoredContact> contactList = await
result.GetContactsAsync();

    foreach (StoredContact contact in contactList)
```

```
    {
        IDictionary<string, object> properties = await
contact.GetPropertiesAsync();
        if (properties.ContainsKey(KnownContactProperties.Email) &&
            properties[KnownContactProperties.Email].ToString() ==
"info@qmatteoq.com")
        {
            MessageBox.Show("Contact found!");
        }
    }
}
```

You can define which fields you're interested in by creating a new **ContactQueryOptions** object and adding it to the **DesiredFields** collection. Then, you can pass the **ContactQueryOptions** object as a parameter when you create the **ContactQueryResult** one. As you can see, defining the fields isn't enough to get the desired result. We still have to query each contact using the **GetPropertiesAsync()** method to see if the information value is the one we're looking for.

The purpose of the **ContactQueryOptions** class is to prepare the next query operations so they can be executed faster.

Updating and deleting contacts

Updating a contact is achieved in the same way as creating new one: after you've retrieved the contact you want to edit, you have to change the required information and call the **SaveAsync()** method again, as in the following sample:

```
private async void OnSearchContactClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);
    ContactQueryOptions options = new ContactQueryOptions();
    options.DesiredFields.Add(KnownContactProperties.Email);

    ContactQueryResult result = store.CreateContactQuery(options);
    IReadOnlyList<StoredContact> contactList = await
result.GetContactsAsync();

    foreach (StoredContact contact in contactList)
    {
        IDictionary<string, object> properties = await
contact.GetPropertiesAsync();
        if (properties.ContainsKey(KnownContactProperties.Email) &&
```

```
            properties[KnownContactProperties.Email].ToString() ==
"info@qmatteoq.com")
        {
            properties[KnownContactProperties.Email] = "mail@domain.com";
            await contact.SaveAsync();
        }
    }
}
```

After we've retrieved the user whose email address is **info@qmatteoq.com**, we change it to **mail@domain.com**, and save it.

Deletion works in a similar way, except that you'll have to deal with the contact's ID, which is a unique identifier that is automatically assigned by the store (you can't set it; you can only read it). Once you've retrieved the contact you want to delete, you have to call the **`DeleteContactAsync()`** method on the **`ContactStore`** object, passing as parameter the contact ID, which is stored in the **`Id`** property of the **`StoredContact`** class.

```
private async void OnSearchContactClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);
    ContactQueryOptions options = new ContactQueryOptions();
    options.DesiredFields.Add(KnownContactProperties.Email);

    ContactQueryResult result = store.CreateContactQuery(options);
    IReadOnlyList<StoredContact> contactList = await
result.GetContactsAsync();

    foreach (StoredContact contact in contactList)
    {
        IDictionary<string, object> properties = await
contact.GetPropertiesAsync();
        if (properties.ContainsKey(KnownContactProperties.Email) &&
            properties[KnownContactProperties.Email].ToString() ==
"info@qmatteoq.com")
        {
            await store.DeleteContactAsync(contact.Id);
        }
    }
}
```

In the previous sample, after we've retrieved the contact with the email address **info@qmatteoq.com**, we delete it using its unique identifier.

Dealing with remote synchronization

When working with custom contact sources, we usually don't simply manage local contacts, but data that is synced with a remote service instead. In this scenario, you have to keep track of the remote identifier of the contact, which will be different from the local one since, as previously mentioned, it's automatically generated and can't be set.

For this scenario, the **StoredContact** class offers a property called **RemoteId** to store such information. Having a **RemoteId** also simplifies the search operations we've seen before. The **ContactStore** class, in fact, offers a method called **FindContactByRemoteIdAsync()**, which is able to retrieve a specific contact based on the remote ID as shown in the following sample:

```
private async void OnFindButtonClicked(object sender, RoutedEventArgs e)
{
    ContactStore store = await
ContactStore.CreateOrOpenAsync(ContactStoreSystemAccessMode.ReadWrite,
ContactStoreApplicationAccessMode.ReadOnly);

    string myRemoteId = "2918";

    RemoteIdHelper remoteHelper = new RemoteIdHelper();
    string taggedRemoteId = await remoteHelper.GetTaggedRemoteId(store,
myRemoteId);
    StoredContact contact = await
store.FindContactByRemoteIdAsync(taggedRemoteId);
}
```

There's one important requirement to keep in mind: the **RemoteId** property's value should be unique across any application installed on the phone that uses a private contact book; otherwise, you'll get an exception.

In this article published by Microsoft, you can see an implementation of a class called **RemoteIdHelper** that offers some methods for adding random information to the remote ID (using a GUID) to make sure it's unique.

Taking advantage of Kid's Corner

Kid's Corner is an interesting and innovative feature introduced in Windows Phone 8 that is especially useful for parents of young children. Basically, it's a sandbox that we can customize. We can decide which apps, games, pictures, videos, and music can be accessed.

As developers, we are able to know when an app is running in Kid's Corner mode. This way, we can customize the experience to avoid providing inappropriate content, such as sharing features.

Taking advantage of this feature is easy; we simply check the **Modes** property of the **ApplicationProfile** class, which belongs to the **Windows.Phone.ApplicationModel** namespace. When it is set to **Default**, the application is running normally. If it's set to **Alternate**, it's running in Kid's Corner mode.

```
private void OnCheckStatusClicked(object sender, RoutedEventArgs e)
{
    if (ApplicationProfile.Modes == ApplicationProfileModes.Default)
    {
        MessageBox.Show("The app is running in normal mode.");
    }
    else
    {
        MessageBox.Show("The app is running in Kid's Corner mode.");
    }
}
```

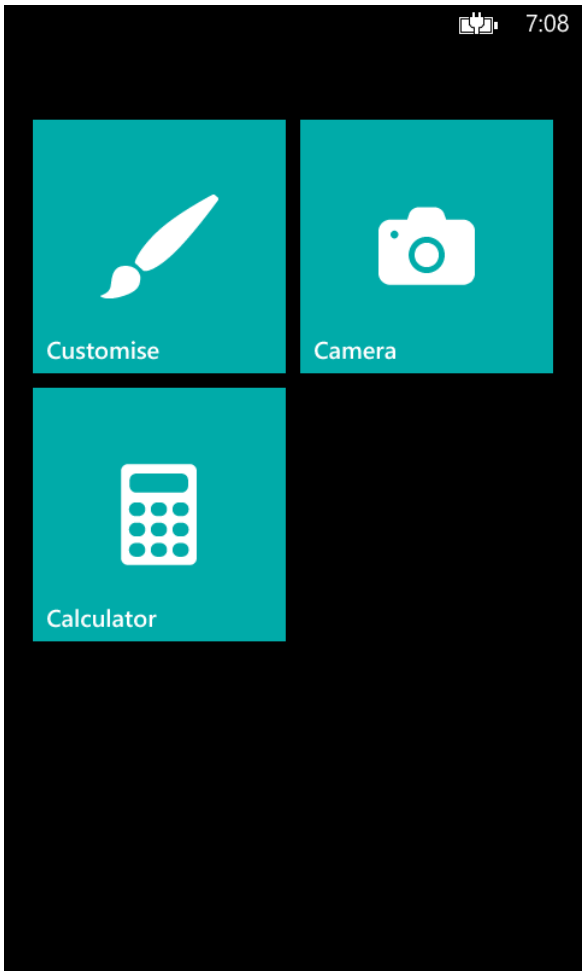

Figure 23: The Kid's Corner Start Screen

Speech APIs: Let's talk with the application

Speech APIs are one of the most interesting new features added in Windows Phone 8. From a user point of view, vocal features are managed in the Settings page. The Speech section lets users configure all the main settings like the voice type, but most of all, it's used to set up the language they want to use for the speech services. Typically, it's set with the same language of the user interface, and users have the option to change it by downloading and installing a new voice pack. It's important to understand how speech services are configured, because in your application, you'll be able to use speech recognition only for languages that have been installed by the user.

The purpose of speech services is to add vocal recognition support in your applications in the following ways:

- Enable users to speak commands to interact with the application, such as opening it and executing a task.
- Enable text-to-speech features so that the application is able to read text to users.
- Enable text recognition so that users can enter text by dictating it instead of typing it.

In this section, we'll examine the basic requirements for implementing all three modes in your application.

Figure 24: Dialog Provided by Windows Phone for Using Voice Commands

Voice commands

Voice commands are a way to start your application and execute a specific task regardless of what the user is doing. They are activated by tapping and holding the Start button. Windows Phone offers native support for many voice commands, such as starting a phone call, dictating email, searching via Bing, and more.

The user simply has to speak a command; if it's successfully recognized, the application will be opened and, as developers, we'll get some information to understand which command has been issued so that we can redirect the user to the proper page or perform a specific operation.

Voice commands are based on VCD files, which are XML files that are included in your project. Using a special syntax, you'll be able to define all the commands you want to support in your application and how the application should behave when they are used. These files are natively supported by Visual Studio. If you right-click on your project and choose **Add new item**, you'll find a template called **VoiceCommandDefinition** in the Windows Phone section.

The following code sample is what a VCD file looks like:

```
<?xml version="1.0" encoding="utf-8"?>

<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="it" Name="NotesCommandSet">
    <CommandPrefix>My notes</CommandPrefix>
    <Example> Open my notes and add a new note </Example>

    <Command Name="AddNote">
      <Example> add a new note </Example>
      <ListenFor> [and] add [a] new note </ListenFor>
      <ListenFor> [and] create [a] new note </ListenFor>
      <Feedback> I'm adding a new note... </Feedback>
      <Navigate Target="/AddNote.xaml" />
    </Command>

  </CommandSet>
</VoiceCommands>
```

A VCD file can contain one or more **`CommandSet`** nodes, which are identified by a **`Name`** and a specific language (the **`xml:lang`** attribute). The second attribute is the most important one. Your application will support voice commands only for the languages you've included in **`CommandSet`** in the VCD file (the voice commands' language is defined by users in the Settings page). You can have multiple **`CommandSet`** nodes to support multiple languages.

Each **CommandSet** can have a **CommandPrefix**, which is the text that should be spoken by users to start sending commands to our application. If one is not specified, the name of the application will automatically be used. This property is useful if you want to localize the command or if your application's title is too complex to pronounce. You can also add an **Example** tag, which contains the text displayed by the Windows Phone dialog to help users understand what kind of commands they can use.

Then, inside a **CommandSet**, you can add up to 100 commands identified by the **Command** tag. Each command has the following characteristics:

- A unique name, which is set in the **Name** attribute.
- The **Example** tag shows users sample text for the current command.
- **ListenFor** contains the text that should be spoken to activate the command. Up to ten **ListenFor** tags can be specified for a single command to cover variations of the text. You can also add optional words inside square brackets. In the previous sample, the **AddNote** command can be activated by pronouncing both "add a new note" or "and add new note."
- **Feedback** is the text spoken by Windows Phone to notify users that it has understood the command and is processing it.
- **NavigateTarget** can be used to customize the navigation flow of the application. If we don't set it, the application will be opened to the main page by default. Otherwise, as in the previous sample, we can redirect the user to a specific page. Of course, in both cases we'll receive the information about the spoken command; we'll see how to deal with them later.

Once we've completed the VCD definition, we are ready to use it in our application.

Note: To use speech services, you'll need to enable the ID_CAP_SPEECH_RECOGNITION option in the manifest file.

Commands are embedded in a Windows Phone application by using a class called **VoiceCommandService**, which belongs to the **Windows.Phone.Speech.VoiceCommands** namespace. This static class exposes a method called **InstallCommandSetFromFileAsync()**, which requires the path of the VCD file we've just created.

```
private async void OnInitVoiceClicked(object sender, RoutedEventArgs e)
{
    await VoiceCommandService.InstallCommandSetsFromFileAsync(new Uri("ms-
appx:///VoiceCommands.xml"));
}
```

The file path is expressed using a **Uri** that should start with the **ms-appx:///** prefix. This **Uri** refers to the Visual Studio project's structure, starting from the root.

Phrase lists

A VCD file can also contain a phrase list, as in the following sample:

```
<?xml version="1.0" encoding="utf-8"?>

<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="en" Name="NotesCommandSet">
    <CommandPrefix>My notes</CommandPrefix>
    <Example> Open my notes and add a new note </Example>

    <Command Name="OpenNote">
      <Example> open the note </Example>
      <ListenFor> open the note {number} </ListenFor>
      <Feedback> I'm opening the note... </Feedback>
      <Navigate />
    </Command>

    <PhraseList Label="number">
      <Item> 1 </Item>
      <Item> 2 </Item>
      <Item> 3 </Item>
    </PhraseList>

  </CommandSet>
</VoiceCommands>
```

Phrase lists are used to manage parameters that can be added to a phrase using braces. Each **`PhraseList`** node is identified by a **`Label`** attribute, which is the keyword to include in the braces inside the **`ListenFor`** node. In the previous example, users can say the phrase "open the note" followed by any of the numbers specified with the **`Item`** tag inside the **`PhraseList`**. You can have up to 2,000 items in a single list.

The previous sample is useful for understanding how this feature works, but it's not very realistic; often the list of parameters is not static, but is dynamically updated during the application execution. Take the previous scenario as an example: in a notes application, the notes list isn't fixed since users can create an unlimited number of notes.

The APIs offer a way to keep a **`PhraseList`** dynamically updated, as demonstrated in the following sample:

```
private async void OnUpdatePhraseListClicked(object sender, RoutedEventArgs e)
{
    VoiceCommandSet commandSet =
VoiceCommandService.InstalledCommandSets["NotesCommandSet"];
    await commandSet.UpdatePhraseListAsync("number", new string[] { "1",
"2", "3", "4", "5" });
}
```

First, you have to get a reference to the current command set by using the **VoiceCommandService.InstalledCommandSets** collection. As the index, you have to use the name of the set that you've defined in the VCD file (the **Name** attribute of the **CommandSet** tag). Once you have a reference to the set, you can call the **UpdatePhraseListAsync()** to update a list by passing two parameters:

- The name of the **PhraseList** (set using the **Label** attribute).
- The collection of new items, as an array of strings.

It's important to keep in mind that the **UpdatePhraseListAsync()** method overrides the current items in the **PhraseList**, so you will have to add all the available items every time, not just the new ones.

Intercepting the requested command

The command invoked by the user is sent to your application with the query string mechanism discussed in Chapter 3. When an application is opened by a command, the user is redirected to the page specified in the **Navigate** node of the VCD file. The following is a sample URI:

```
/MainPage.xaml?voiceCommandName=AddNote&reco=My%20notes%20create%20a%20new%
20note
```

The **voiceCommandName** parameter contains the spoken command, while the **reco** parameter contains the full text that has been recognized by Windows Phone.

If the command supports a phrase list, you'll get another parameter with the same name of the **PhraseList** and the spoken item as a value. The following code is a sample URI based on the previous note sample, where the user can open a specific note by using the **OpenNote** command:

```
/MainPage.xaml?voiceCommandName=OpenNote&reco=My%20notes%20open%20a%20new%2
0note&number=2
```

Using the APIs we saw in Chapter 3, it's easy to extract the needed information from the query string parameters and use them for our purposes, like in the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("voiceCommandName"))
    {
        string commandName =
NavigationContext.QueryString["voiceCommandName"];

        switch (commandName)
        {
```

```
            case "AddNote":
                //Create a new note.
                break;
            case "OpenNote":
                if (NavigationContext.QueryString.ContainsKey("number"))
                {
                    int selectedNote =
int.Parse(NavigationContext.QueryString["number"]);
                    //Load the selected note.
                }
                break;
        }
    }
}
```

We use a **switch** statement to manage the different supported commands that are available in the **NavigationContext.QueryString** collection. If the user is trying to open a note, we also get the value of the **number** parameter.

Working with speech recognition

In the beginning of this section, we talked about how to recognize commands that are spoken by the user to open the application. Now it's time to see how to do the same within the app itself, to recognize commands and allow users to dictate text instead of typing it (a useful feature in providing a hands-free experience).

There are two ways to start speech recognition: by providing a user interface, or by working silently in the background.

In the first case, you can provide users a visual dialog similar to the one used by the operating system when holding the Start button. It's the perfect solution to manage vocal commands because you'll be able to give both visual and voice feedback to users.

This is achieved by using the **SpeechRecognizerUI** class, which offers four key properties to customize the visual dialog:

- **ListenText** is the large, bold text that explains to users what the application is expecting.
- **Example** is additional text that is displayed below the **ListenText** to help users better understand what kind of speech the application is expecting.
- **ReadoutEnabled** is a **Boolean** property; when it's set to true, Windows Phone will read the recognized text to users as confirmation.
- **ShowConfirmation** is another **Boolean** property; when it's set to true, users will be able to cancel the operation after the recognition process is completed.

The following sample shows how this feature is used to allow users to dictate a note. We ask users for the text of the note and then, if the operation succeeded, we display the recognized text.

```
private async void OnStartRecordingClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizerUI sr = new SpeechRecognizerUI();
    sr.Settings.ListenText = "Start dictating the note";
    sr.Settings.ExampleText = "dictate the note";
    sr.Settings.ReadoutEnabled = false;
    sr.Settings.ShowConfirmation = true;

    SpeechRecognitionUIResult result = await sr.RecognizeWithUIAsync();
    if (result.ResultStatus == SpeechRecognitionUIStatus.Succeeded)
    {
        RecordedText.Text = result.RecognitionResult.Text;
    }
}
```

Notice how the recognition process is started by calling the **`RecognizeWithUIAsync()`** method, which returns a **`SpeechRecognitionUIResult`** object that contains all the information about the operation.

To silently recognize text, less code is needed since fewer options are available than were used for the dialog. We just need to start listening for the text and understand it. We can do this by calling the **`RecognizeAsync()`** method of the **`SpeechRecognizer`** class. The recognition result will be stored in a **`SpeechRecognitionResult`** object, which is the same that was returned in the **`RecognitionResult`** property by the **`RecognizeWithUIAsync()`** method we used previously.

Using custom grammars

The code we've seen so far is used to recognize almost any word in the dictionary. For this reason, speech services will work only if the phone is connected to the Internet since the feature uses online Microsoft services to parse the results.

This approach is useful when users talk to the application to dictate text, as in the previous samples with the note application. But if only a few commands need to be managed, having access to all the words in the dictionary is not required. On the contrary, complete access can cause problems because the application may understand words that aren't connected to any supported command.

For this scenario, the Speech APIs provide a way to use a custom grammar and limit the number of words that are supported in the recognition process. There are three ways to set a custom grammar:

- Using only the available standard sets.
- Manually adding the list of supported words.
- Storing the words in an external file.

Again, the starting point is the **SpeechRecognizer** class, which offers a property called **Grammars**.

To load one of the predefined grammars, use the **AddGrammarFromPredefinedType()** method, which accepts as parameters a string to identify it (you can choose any value) and the type of grammar to use. There are two sets of grammars: the standard **SpeechPredefinedGrammar.Dictation**, and **SpeechPredefinedGrammar.WebSearch**, which is optimized for web related tasks.

In the following sample, we recognize speech using the **WebSearch** grammar:

```
private async void OnStartRecordingWithoutUIClicked(object sender,
RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    recognizer.Grammars.AddGrammarFromPredefinedType("WebSearch",
SpeechPredefinedGrammar.WebSearch);
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    RecordedText.Text = result.Text;
}
```

Even more useful is the ability to allow the recognition process to understand only a few selected words. We can use the **AddGrammarFromList()** method offered by the **Grammars** property, which requires the usual identification key followed by a collection of supported words.

In the following sample, we set the **SpeechRecognizer** class to understand only the words “save” and “cancel”.

```
private async void OnStartRecordingClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    string[] commands = new[] { "save", "cancel" };
    recognizer.Grammars.AddGrammarFromList("customCommands", commands);

    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Text == "save")
    {
        //Saving
    }
    else if (result.Text == "cancel")
    {
        //Cancelling the operation
    }
    else
    {
        MessageBox.Show("Command not recognized");
    }
}
```

If the user says a word that is not included in the custom grammar, the **`Text`** property of the **`SpeechRecognitionResult`** object will be empty. The biggest benefit of this approach is that it doesn't require an Internet connection since the grammar is stored locally.

The third and final way to load a grammar is by using another XML definition called **Speech Recognition Grammar Specification (SRGS)**. You can read more about the supported tags in the official documentation by W3C.

The following sample shows a custom grammar file:

```
<?xml version="1.0" encoding="utf-8" ?>

<grammar version="1.0" xml:lang="en" root="rootRule" tag-
format="semantics/1.0"
         xmlns="http://www.w3.org/2001/06/grammar"

xmlns:sapi="http://schemas.microsoft.com/Speech/2002/06/SRGSExtensions">

  <rule id="openAction">
    <one-of>
      <item>open</item>
      <item>load</item>
    </one-of>
  </rule>

  <rule id="fileWords">
    <one-of>
      <item> note </item>
      <item> reminder </item>
    </one-of>
  </rule>

  <rule id="rootRule">
    <ruleref uri="#openAction" />
    <one-of>
      <item>the</item>
      <item>a</item>
    </one-of>
    <ruleref uri="#fileWords" />
  </rule>

</grammar>
```

The file describes both the supported words and the correct order that should be used. The previous sample shows the supported commands to manage notes in an application, like "Open the note" or "Load a reminder," while a command like "Reminder open the" is not recognized.

Visual Studio 2012 offers built-in support for these files with a specific template called **SRGS Grammar** that is available when you right-click your project and choose **Add new item**.

Once the file is part of your project, you can load it using the **AddGrammarFromUri()** method of the **SpeechRecognizer** class that accepts as a parameter the file path expressed as a **Uri**, exactly as we've seen for VCD files. From now on, the recognition process will use the grammar defined in the file instead of the standard one, as shown in the following sample:

```
private async void OnStartRecordingWithCustomFile(object sender,
RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    recognizer.Grammars.AddGrammarFromUri("CustomGrammar", new Uri("ms-
appx:///CustomGrammar.xml"));
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Text != string.Empty)
    {
        RecordedText.Text = result.Text;
    }
    else
    {
        MessageBox.Show("Not recognized");
    }
}
```

Using text-to-speech (TTS)

Text-to-speech is a technology that is able to read text to users in a synthesized voice. It can be used to create a dialogue with users so they won't have to watch the screen to interact with the application.

The basic usage of this feature is really simple. The base class to interact with TTS services is **SpeechSynthesizer**, which offers a method called **SpeakTextAsync()**. You simply have to pass to the method the text that you want to read, as shown in the following sample:

```
private async void OnSpeakClicked(object sender, RoutedEventArgs e)
{
    SpeechSynthesizer synth = new SpeechSynthesizer();
    await synth.SpeakTextAsync("This is a sample text");
}
```

Moreover, it's possible to customize how the text is pronounced by using a standard language called **Synthesis Markup Language (SSML)**, which is based on the XML standard. This standard provides a series of XML tags that defines how a word or part of the text should be pronounced. For example, the speed, language, voice gender, and more can be changed.

The following sample is an example of an SSML file:

```
<?xml version="1.0"?>
```

```
<speak xmlns="http://www.w3.org/2001/10/synthesis"
       xmlns:dc="http://purl.org/dc/elements/1.1/"
       xml:lang="en"
       version="1.0">

  <voice age="5">This text is read by a child</voice>
  <break />
  <prosody rate="x-slow"> This text is read very slowly</prosody>

</speak>
```

This code features three sample SSML tags: **voice** for simulating the voice's age, **break** to add a pause, and **prosody** to set the reading speed using the **rate** attribute.

There are two ways to use an SSML definition in your application. The first is to create an external file by adding a new XML file in your project. Next, you can load it by passing the file path to the **SpeakSsmlFromUriAsync()** method of the **SpeechSynthesizer** class, similar to how we loaded the VCD file.

```
private async void OnSpeakClicked(object sender, RoutedEventArgs e)
{
    SpeechSynthesizer synth = new SpeechSynthesizer();
    await synth.SpeakSsmlFromUriAsync(new Uri("ms-appx:///SSML.xml"));
}
```

Another way is to define the text to be read directly in the code by creating a string that contains the SSML tags. In this case, we can use the **SpeakSsmlAsync()** method which accepts the string to read as a parameter. The following sample shows the same SSML definition we've been using, but stored in a string instead of an external file.

```
private async void OnSpeakClicked(object sender, RoutedEventArgs e)
{
    SpeechSynthesizer synth = new SpeechSynthesizer();

    StringBuilder textToRead = new StringBuilder();
    textToRead.AppendLine("<speak version=\"1.0\"");
    textToRead.AppendLine("
xmlns=\"http://www.w3.org/2001/10/synthesis\"");
    textToRead.AppendLine(" xml:lang=\"en\">");
    textToRead.AppendLine(" <voice age=\"5\">This text is read by a
child</voice>");
    textToRead.AppendLine("<prosody rate=\"x-slow\"> This text is read very
slowly</prosody>");
    textToRead.AppendLine("</speak>");

    await synth.SpeakSsmlAsync(textToRead.ToString());
}
```

You can learn more about the SSML definition and available tags in the official documentation provided by W3C.

Data sharing

Data sharing is a new feature introduced in Windows Phone 8 that can be used to share data between different applications, including third-party ones.

There are two ways to manage data sharing:

- **File sharing**: The application registers an extension such as .**`log`**. It will be able to manage any file with the registered extension that is opened by another application (for example, a mail attachment).
- **Protocol sharing**: The application registers a protocol such as **`log:`**. Other applications will be able to use it to send plain data like strings or numbers.

In both cases, the user experience is similar:

- If no application is available on the device to manage the requested extension or protocol, users will be asked if they want to search the Store for one that can.
- If only one application is registered for the same extension or protocol, it will automatically be opened.
- If multiple applications are registered for the same extension or protocol, users will be able to choose which one to use.

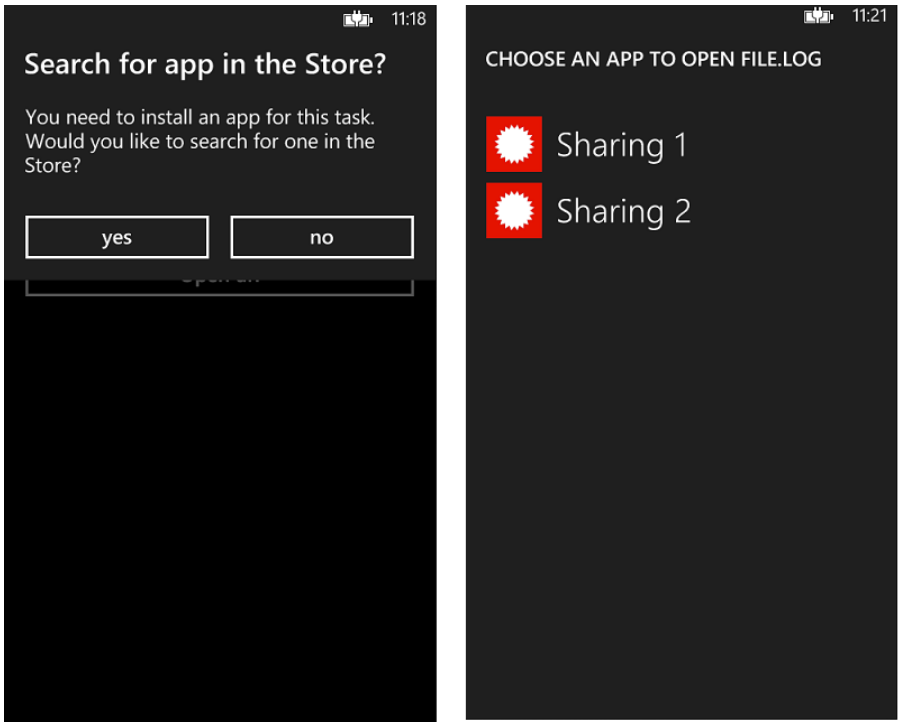

Figure 25: Data Sharing When a Compatible App Is Not on the Device (Left), and When Multiple Apps Are Compatible (Right)

Let's discuss how to support both scenarios in our application.

Note: There are some file types and protocols that are registered by the system, like Office files, pictures, mail protocols, etc. You can't override them; only Windows Phone is able to manage them. You can see a complete list of the reserved types in the MSDN documentation.

File sharing

File sharing is supported by adding a new definition in the manifest file that notifies the operating system which extensions the application can manage. As many other scenarios we've previously seen, this modification is not supported by the visual editor, so we'll need to right-click the manifest file and choose the **View code** option.

The extension is added in the **`Extensions`** section, which should be defined under the **`Token`** one:

```
<Extensions>
  <FileTypeAssociation Name="LogFile" TaskID="_default"
NavUriFragment="fileToken=%s">
    <Logos>
      <Logo Size="small">log-33x33.png</Logo>
      <Logo Size="medium">log-69x69.png</Logo>
      <Logo Size="large">log-176x176.png</Logo>
    </Logos>
    <SupportedFileTypes>
      <FileType ContentType="text/plain">.log</FileType>
    </SupportedFileTypes>
  </FileTypeAssociation>
</Extensions>
```

Every supported file type has its own **`FileTypeAssociation`** tag, which is identified by the **`Name`** attribute (which should be unique). Inside this node are two nested sections:

- **`Logos`** is optional and is used to support an icon to visually identify the file type. Three different images are required, each with a different resolution: 33 × 33, 69 × 69, and 176 × 176.The icons are used in various contexts, such as when the file is received as an email attachment.
- **`SupportedFileTypes`** is required because it contains the extensions that are going to be supported for the current file type. Multiple extensions can be added.

The previous sample is used to manage the **`.log`** file extension in our application.

When another application tries to open a file we support, our application is opened using a special URI:

```
/FileTypeAssociation?fileToken=89819279-4fe0-9f57-d633f0949a19
```

The **fileToken** parameter is a GUID that univocally identifies the file—we're going to use it later.

To manage the incoming URI, we need to introduce the **UriMapper** class we talked about in Chapter 3. When we identify this special URI, we're going to redirect the user to a specific page of the application that is able to interact with the file.

The following sample shows what the **UriMapper** looks like:

```
public class UriMapper: UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        string tempUri = HttpUtility.UrlDecode(uri.ToString());
        if (tempUri.Contains("/FileTypeAssociation"))
        {
            int fileIdIndex = tempUri.IndexOf("fileToken=") + 10;
            string fileId = tempUri.Substring(fileIdIndex);

            string incomingFileName =
                SharedStorageAccessManager.GetSharedFileName(fileId);

            string incomingFileType =
System.IO.Path.GetExtension(incomingFileName);

            switch (incomingFileType)
            {
                case ".log":
                    return new Uri("/LogPage.xaml?fileToken=" + fileId,
UriKind.Relative);
                default:
                    return new Uri("/MainPage.xaml", UriKind.Relative);
            }
        }

        return uri;
    }
}
```

If the starting **Uri** contains the **FileTypeAssociation** keyword, it means that the application has been opened due to a file sharing request. In this case, we need to identify the opened file's extension. We extract the **fileToken** parameter and, by using the **GetSharedFileName()** of the **SharedAccessManager** class (which belongs to the **Windows.Phone.Storage** namespace), we retrieve the original file name.

By reading the name, we're able to identify the extension and perform the appropriate redirection. In the previous sample, if the extension is **.log**, we redirect the user to a specific page of the application called **LogPage.xaml**. It's important to add to the **Uri** the **fileToken** parameter as a query string; we're going to use it in the page to effectively retrieve the file. Remember to register the **UriMapper** in the **App.xaml.cs** file, as explained in Chapter 3.

Tip: The previous UriMapper shows a full example that works when the application supports multiple file types. If your application supports just one extension, it's not needed to retrieve the file name and identify the file type. Since the application can be opened with the special URI only in a file sharing scenario, you can immediately redirect the user to the dedicated page.

Now it's time to interact with the file we received from the other application. We'll do this in the page that we've created for this purpose (in the previous sample code, it's the one called **LogPage.xaml**).

We've seen that when another application tries to open a **.log** file, the user is redirected to the **LogPage.xaml** page with the **fileToken** parameter added to the query string. We're going to use the **OnNavigatedTo** event to manage this scenario:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("fileToken"))
    {
        await
SharedStorageAccessManager.CopySharedFileAsync(ApplicationData.Current.Loca
lFolder, "file.log",

NameCollisionOption.ReplaceExisting,

NavigationContext.QueryString["fileToken"]);

    }
}
```

Again we use the **SharedStorageAccessManager** class, this time by invoking the **CopySharedFileAsync()** method. Its purpose is to copy the file we received to the local storage so that we can work with it.

The required parameters are:

- A **`StorageFolder`** object, which represents the local storage folder in which to save the file (in the previous sample, we save it in the root).
- The name of the file.
- The behavior to apply in case a file with the same name already exists (by using one of the values of the **`NameCollisionOption`** enumerator).
- The GUID that identifies the file, which we get from the **`fileToken`** query string parameter.

Once the operation is completed, a new file called **file.log** will be available in the local storage of the application, and we can start playing with it. For example, we can display its content in the current page.

How to open a file

So far we've seen how to manage an opened file in our application, but we have yet to discuss how to effectively open a file.

The task is easily accomplished by using the **`LaunchFileAsync()`** method offered by the **`Launcher`** class (which belongs to the **`Windows.System`** namespace). It requires a **`StorageFile`** object as a parameter, which represents the file you would like to open.

In the following sample, you can see how to open a log file that is included in the Visual Studio project:

```
private async void OnOpenFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile storageFile = await
Windows.ApplicationModel.Package.Current.InstalledLocation.GetFileAsync("fi
le.log");
    Windows.System.Launcher.LaunchFileAsync(storageFile);
}
```

Protocol sharing

Protocol sharing works similarly to file sharing. We're going to register a new extension in the manifest file, and we'll deal with the special URI that is used to launch the application.

Let's start with the manifest. In this case as well, we'll have to add a new element in the **`Extensions`** section that can be accessed by manually editing the file through the **View code** option.

```
<Extensions>
  <Protocol Name="log" NavUriFragment="encodedLaunchUri=%s" TaskID="_default"
/>
</Extensions>
```

The most important attribute is **Name**, which identifies the protocol we're going to support. The other two attributes are fixed.

An application that supports protocol sharing is opened with the following URI:

```
/Protocol?encodedLaunchUri=log:ShowLog?LogId=1
```

The best way to manage it is to use a **UriMapper** class, as we did for file sharing. The difference is that this time, we'll look for the **encodedLaunchUri** parameter. However, the result will be the same: we will redirect the user to the page that is able to manage the incoming information.

```
public class UriMapper : UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        string tempUri = System.Net.HttpUtility.UrlDecode(uri.ToString());

        if (tempUri.Contains("Protocol"))
        {
            int logIdIndex = tempUri.IndexOf("LogId=") + 6;
            string logId = tempUri.Substring(logIdIndex);

            return new Uri("/LogPage.xaml?LogId=" + logId,
UriKind.Relative);
        }

        return uri;
    }
}
```

In this scenario, the operation is simpler. We extract the value of the parameter **LogId** and pass it to the **LogPage.xaml** page. Also, we have less work to do in the landing page; we just need to retrieve the parameter's value using the **OnNavigatedTo** event, and use it to load the required data, as shown in the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("LogId"))
    {
        string logId = NavigationContext.QueryString["LogId"];
        MessageBox.Show(logId);
    }
}
```

How to open a URI

Similar to file sharing, other applications can interact with ours by using the protocol sharing feature and the **`Launcher`** class that belongs to the **`Windows.System`** namespace.

The difference is that we need to use the **`LaunchUriAsync()`** method, as shown in the following sample:

```
private async void OnOpenUriClicked(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri("log:ShowLog?LogId=1");
    await Windows.System.Launcher.LaunchUriAsync(uri);
}
```

A quick recap

In this chapter, we've examined various ways to integrate our application with the features offered by the Windows Phone platform:

- We started with the simplest integration available: launchers and choosers, which are used to demand an operation from the operating system and eventually get some data in return.
- We looked at how to interact with user contacts and appointments: first with a read-only mode offered by a new set of APIs introduced in Windows Phone 7.5, and then with the private contacts book, which is a contacts store that belongs to the application but can be integrated with the native People Hub.
- We briefly talked about how to take advantage of Kid's Corner, an innovative feature introduced to allow kids to safely use the phone without accessing applications that are not suitable for them.
- We learned how to use one of the most powerful new APIs added in Windows Phone 8: Speech APIs, to interact with our application using voice commands.
- We introduced data sharing, which is another new feature used to share data between different applications, and we can manage file extensions and protocols.

Chapter 8 Multimedia Applications

Using the camera

The camera is one of the most important features in Windows Phone devices, especially thanks to Nokia, which has created some of the best camera phones available on the market.

As developers, we are able to integrate the camera experience into our application so that users can take pictures and edit them directly within the application. In addition, with the Lens App feature we'll discuss later, it's even easier to create applications that can replace the native camera experience.

Note: To interact with the camera, you need to enable the ID_CAP_IS_CAMERA capability in the manifest file.

The first step is to create an area on the page where we can display the image recorded by the camera. We're going to use **`VideoBrush`**, which is one of the native XAML brushes that is able to embed a video. We'll use it as a background of a **`Canvas`** control, as shown in the following sample:

```
<Canvas Height="400" Width="400">
    <Canvas.Background>
        <VideoBrush x:Name="video" Stretch="UniformToFill">
            <VideoBrush.RelativeTransform>
                <CompositeTransform x:Name="previewTransform" CenterX=".5"
CenterY=".5" />
            </VideoBrush.RelativeTransform>
        </VideoBrush>
    </Canvas.Background>
</Canvas>
```

Notice the **`CompositeTransform`** that has been applied; its purpose is to keep the correct orientation of the video, based on the camera orientation.

Taking pictures

Now that we have a place to display the live camera feed, we can use the APIs that are included in the **`Windows.Phone.Media.Capture`** namespace. Specifically, the class available to take pictures is called **`PhotoCaptureDevice`** (later we'll see another class for recording videos).

The following code is a sample initialization:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    Size resolution =
PhotoCaptureDevice.GetAvailableCaptureResolutions(CameraSensorLocation.Back
).First();
     PhotoCaptureDevice camera = await
PhotoCaptureDevice.OpenAsync(CameraSensorLocation.Back, resolution);
    video.SetSource(camera);
    previewTransform.Rotation = camera.SensorRotationInDegrees;
}
```

Before initializing the live feed, we need to make two choices: which camera to use, and which of the available resolutions we want to use.

We achieve this by calling the **`GetAvailableCaptureResolutions()`** method on the **`PhotoCaptureDevice`** class, passing as parameter a **`CameraSensorLocation`** object which represents the camera we're going to use. The method will return a collection of the supported resolutions, which are identified by the **`Size`** class.

Tip: It's safe to use the previous code because every Windows Phone device has a back camera. If we want to interact with the front camera instead, it's better to check whether one is available first since not all the Windows Phone devices have one. To do this, you can use the `AvailableSensorLocation` property of the `PhotoCaptureDevice` class, which is a collection of all the supported cameras.

Once we've decided which resolution to use, we can pass it as a parameter (together again with the selected camera) to the **`OpenAsync()`** method of the **`PhotoCaptureDevice`** class. It will return a **`PhotoCaptureDevice`** object which contains the live feed; we simply have to pass it to the **`SetSource()`** method of the **`VideoBrush`**. As already mentioned, we handle the camera orientation using the transformation we've applied to the **`VideoBrush`**: we set the **`Rotation`** using the **`SensorRotationInDegrees`** property that contains the current angle's rotation.

Note: You may get an error when you try to pass a `PhotoCaptureDevice` object as a parameter of the `SetSource()` method of the `VideoBrush`. If so, you'll have to add the `Microsoft.Devices` namespace to your class, since it contains an extension method for the `SetSource()` method that supports the `PhotoCaptureDevice` class.

Now the application will simply display the live feed of the camera on the screen. The next step is to take the picture.

The technique used by the API is to create a sequence of frames and save them as a stream. Unfortunately, there's a limitation in the current SDK: you'll only be able to take one picture at a time, so you'll only be able to use sequences made by one frame.

```
private async void OnTakePhotoClicked(object sender, RoutedEventArgs e)
{
    CameraCaptureSequence cameraCaptureSequence =
camera.CreateCaptureSequence(1);

    MemoryStream stream = new MemoryStream();
    cameraCaptureSequence.Frames[0].CaptureStream =
stream.AsOutputStream();

    await camera.PrepareCaptureSequenceAsync(cameraCaptureSequence);
    await cameraCaptureSequence.StartCaptureAsync();

    stream.Seek(0, SeekOrigin.Begin);

    MediaLibrary library = new MediaLibrary();
    library.SavePictureToCameraRoll("picture1.jpg", stream);
}
```

The process starts with a **CameraCaptureSequence** object, which represents the capture stream. Due to the single-picutre limitation previously mentioned, you'll be able to call the **CreateCaptureSequence()** method of the **PhotoCaptureDevice** class only by passing **1** as its parameter.

For the same reason, we're just going to work with the first frame of the sequence that is stored inside the **Frames** collection. The **CaptureStream** property of the frame needs to be set with the stream that we're going to use to store the captured image. In the previous sample, we use a **MemoryStream** to store the photo in memory. This way, we can save it later in the user's Photos Hub (specifically, in the Camera Roll album).

Note: To interact with the `MediaLibrary` class you need to enable the ID_CAP_MEDIALIB_PHOTO capability in the manifest file

You can also customize many settings of the camera by calling the **SetProperty()** method on the **PhotoCaptureDevice** object that requires two parameters: the property to set, and the value to assign. The available properties are defined by two enumerators called **KnownCameraGeneralProperties**, which contains the general camera properties, and **KnownCameraPhotoProperties**, which contains the photo-specific properties.

Some properties are read-only, so the only operation you can perform is get their values by using the **GetProperty()** method.

In the following samples, we use the **SetProperty()** method to set the flash mode and **GetProperty()** to get the information if the current region forces phones to play a sound when they take a picture.

```
private void OnSetPropertiesClicked(object sender, RoutedEventArgs e)
{
    camera.SetProperty(KnownCameraPhotoProperties.FlashMode,
FlashMode.RedEyeReduction);
    bool isShutterSoundRequired =
(bool)camera.GetProperty(KnownCameraGeneralProperties.IsShutterSoundRequire
dForRegion);
}
```

Note that the **GetProperty()** method always returns a generic object, so you'll have to manually cast it according to the properties you're querying.

You can see a list of all the available properties in the MSDN documentation.

Using the hardware camera key

Typically, Windows Phone devices have a dedicated button for the camera, which can be used both to set the focus by half-pressing it, and to take the picture by fully pressing it. You are also able to use this button in your applications by subscribing to three events that are exposed by the **CameraButtons** static class:

- **ShutterKeyPressed** is triggered when the button is pressed.
- **ShutterKeyReleased** is triggered when the button is released.
- **ShutterKeyHalfPressed** is triggered when the button is half-pressed.

In the following sample, we subscribe to the **ShutterKeyReleased** event to take a picture and the **ShutterKeyHalfPressed** event to use the auto-focus feature.

```
public Camera()
{
    InitializeComponent();
    CameraButtons.ShutterKeyReleased += CameraButtons_ShutterKeyReleased;
    CameraButtons.ShutterKeyHalfPressed +=
CameraButtons_ShutterKeyHalfPressed;
}

async void CameraButtons_ShutterKeyHalfPressed(object sender, EventArgs e)
{
    await camera.FocusAsync();
}

void CameraButtons_ShutterKeyReleased(object sender, EventArgs e)
```

```
{
    //Take the picture.
}
```

Recording a video

The process to record a video is similar to the one we used to take a picture. In this case, we're going to use the **AudioVideoCaptureDevice** class instead of the **PhotoCaptureDevice** class. As you can see in the following sample, the initialization procedure is the same: we decide which resolution and camera we want to use, and we display the returned live feed using a **VideoBrush**.

Note: To record videos, you'll also need to enable the ID_CAP_MICROPHONE capability in the manifest file.

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    Size resolution =
AudioVideoCaptureDevice.GetAvailableCaptureResolutions(CameraSensorLocation
.Back).First();
    videoDevice = await
AudioVideoCaptureDevice.OpenAsync(CameraSensorLocation.Back, resolution);
    video.SetSource(videoDevice);
    previewTransform.Rotation = videoDevice.SensorRotationInDegrees;
}
```

Recording a video is even simpler since the **AudioVideoCaptureDevice** class exposes the **StartRecordingToStreamAsync()** method, which simply requires you to specify where to save the recorded data. Since it's a video, you'll also need a way to stop the recording; this is the purpose of the **StopRecordingAsync()** method.

In the following sample, the recording is stored in a file created in the local storage:

```
private AudioVideoCaptureDevice videoDevice;
private IRandomAccessStream stream;
private StorageFile file;

public VideoRecording()
{
    InitializeComponent();
}

private async void OnRecordVideoClicked(object sender, RoutedEventArgs e)
```

```
{
    file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("video.wmv",
CreationCollisionOption.ReplaceExisting);
    stream = await file.OpenAsync(FileAccessMode.ReadWrite);

    videoDevice.StartRecordingToStreamAsync(stream);
}

private async void OnStopRecordingClicked(object sender, RoutedEventArgs e)
{
    await videoDevice.StopRecordingAsync();
    await stream.FlushAsync();
}
```

You can easily test the result of the operation by using the **MediaPlayerLauncher** class to play the recording:

```
private void OnPlayVideoClicked(object sender, RoutedEventArgs e)
{
    MediaPlayerLauncher launcher = new MediaPlayerLauncher
         {
              Media = new Uri(file.Path, UriKind.Relative)
         };
    launcher.Show();
}
```

The SDK offers a specific list of customizable settings connected to video recording. They are available in the **KnownCameraAudioVideoProperties** enumerator.

Interacting with the media library

The framework offers a class called **MediaLibrary**, which can be used to interact with the user media library (photos, music, etc.). Let's see how to use it to manage the most common scenarios.

Note: In the current version, there's no way to interact with the library to save new videos in the camera roll, nor to get access to the stream of existing videos.

Pictures

The **MediaLibrary** class can be used to get access to the pictures stored in the Photos Hub, thanks to the **Pictures** collection. It's a collection of **Picture** objects, where each one represents a picture stored in the Photos Hub.

Note: You'll need to enable the ID_CAP_MEDIALIB_PHOTO capability in the manifest file to get access to the pictures stored in the Photos Hub.

The **Pictures** collection grants access to the following albums:

- Camera Roll
- Saved Pictures
- Screenshots

All other albums displayed in the People Hub that come from remote services like SkyDrive or Facebook can't be accessed using the **MediaLibrary** class.

Tip: The `MediaLibrary` class exposes a collection called `SavedPictures`, which contains only the pictures that are stored in the Saved Pictures album.

Every **Picture** object offers some properties to get access to the basic info, like **Name**, **Width**, and **Height**. A very important property is **Album**, which contains the reference of the album where the image is stored. In addition, you'll be able to get access to different streams in case you want to manipulate the image or display it in your application:

- The **GetPicture()** method returns the stream of the original image.
- The **GetThumbnail()** method returns the stream of the thumbnail, which is a low-resolution version of the original image.
- If you add the **PhoneExtensions** namespace to your class, you'll be able to use the **GetPreviewImage()** method, which returns a preview picture. Its resolution and size are between the original image and the thumbnail.

In the following sample, we generate the thumbnail of the first available picture in the Camera Roll and display it using an **Image** control:

Tip: To interact with the `MediaLibrary` class using the emulator, you'll have to open the Photos Hub at least once; otherwise you will get an empty collection of pictures when you query the `Pictures` property.

```
private void OnSetPhotoClicked(object sender, RoutedEventArgs e)
{
    MediaLibrary library = new MediaLibrary();
    Picture picture = library.Pictures.FirstOrDefault(x => x.Album.Name ==
"Camera Roll");
    if (picture != null)
    {
        BitmapImage image = new BitmapImage();
        image.SetSource(picture.GetThumbnail());
        Thumbnail.Source = image;
    }
}
```

With the **MediaLibrary** class, you'll also be able to do the opposite: take a picture in your application and save it in the People Hub. We've already seen a sample when we talked about integrating the camera in our application; we can save the picture in the camera roll (using the **SavePictureToCameraRoll()** method) or in the Saved Pictures album (using the **SavePicture()** method). In both cases, the required parameters are the name of the image and its stream.

In the following sample, we download an image from the Internet and save it in the Saved Pictures album:

```
private async void OnDownloadPhotoClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    Stream stream = await
client.GetStreamAsync("http://www.syncfusion.com/Content/en-
US/Home/Images/syncfusion-logo.png");
    MediaLibrary library = new MediaLibrary();
    library.SavePicture("logo.jpg", stream);
}
```

Music

The **MediaLibrary** class offers many options for accessing music, but there are some limitations that aren't present when working with pictures.

Note: You'll need to enable the ID_CAP_MEDIALIB_AUDIO capability in the manifest file to get access to the pictures stored in the Photos Hub.

The following collections are exposed by the **MediaLibrary** class for accessing music:

- **Albums** to get access to music albums.
- **Songs** to get access to all the available songs.
- **Genres** to get access to the songs grouped by genre.
- **Playlists** to get access to playlists.

Every song is identified by the **Song** class, which contains all the common information about a music track taken directly from the ID3 tag: **Album**, **Artist**, **Title**, **TrackNumber**, and so on.

Unfortunately, there's no access to a song's stream, so the only way to play tracks is by using the **MediaPlayer** class, which is part of the **Microsoft.XNA.Framework.Media** namespace. This class exposes many methods to interact with tracks. The **Play()** method accepts as a parameter a **Song** object, retrieved from the **MediaLibrary**.

In the following sample, we reproduce the first song available in the library:

```
private void OnPlaySong(object sender, RoutedEventArgs e)
{
    MediaLibrary library = new MediaLibrary();
    Song song = library.Songs.FirstOrDefault();
    MediaPlayer.Play(song);
}
```

One of the new features introduced in Windows Phone 8 allows you to save a song stored in the application's local storage to the media library so that it can be played by the native Music + Videos Hub. This requires the **Microsoft.Xna.Framework.Media.PhoneExtensions** namespace to be added to your class.

```
private async void OnDownloadMusicClicked(object sender, RoutedEventArgs e)
{
    MediaLibrary library = new MediaLibrary();
    SongMetadata metadata = new SongMetadata
    {
        AlbumName = "A rush of blood to the head",
        ArtistName = "Coldplay",
        Name = "Clocks"
    };
    library.SaveSong(new Uri("song.mp3", UriKind.RelativeOrAbsolute),
metadata, SaveSongOperation.CopyToLibrary);
}
```

The **SaveSong()** method requires three parameters, as shown in the previous sample:

- The path of the song to save. It's a relative path that points to the local storage.
- The song metadata, which is identified by the **SongMetadata** class. It's an optional parameter; if you pass **null**, Windows Phone will automatically extract the ID3 information from the file.
- A **SaveSongOperation** object, which tells the media library if the file should be copied (**CopyToLibrary**) or moved (**MoveToLibrary**) so that it's deleted from the storage.

Lens apps

Windows Phone 8 has introduced new features specific to photographic applications. Some of the most interesting are called **lens apps**, which apply different filters and effects to pictures. Windows Phone offers a way to easily switch between different camera applications to apply filters on the fly.

Lens apps are regular Windows Phone applications that interact with the Camera APIs we used at the beginning of this chapter. The difference is that a lens app is displayed in the **lenses** section of the native Camera app; when users press the camera button, a special view with all the available lens apps is displayed. This way, they can easily switch to another application to take the picture.

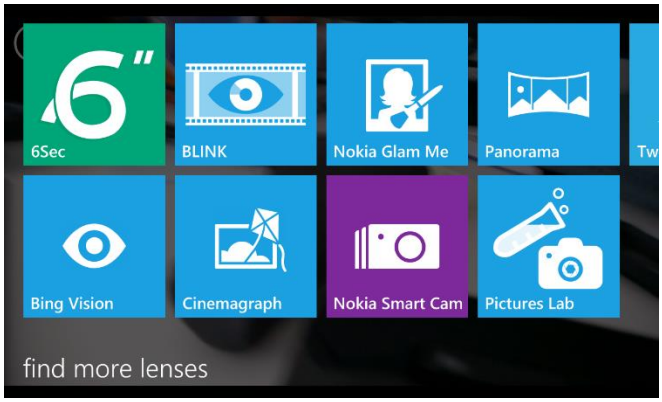

Figure 26: Lenses View in the Camera App

Integration with the lenses view starts from the manifest file, which must be manually edited by choosing the **View code** option in the context menu. The following code has to be added in the **Extension** section:

```
<Extensions>
  <Extension ExtensionName="Camera_Capture_App" ConsumerID="{5B04B775-356B-
4AA0-AAF8-6491FFEA5631}" TaskID="_default" />
</Extensions>
```

Every lens app needs a specific icon that is displayed in the lenses view. Icons are automatically retrieved from the **Assets** folder based on a naming convention. An icon must be added for every supported resolution using the conventions in the following table:

Resolution	Icon size	File name
480 × 800	173 × 173	Lens.Screen-WVGA.png
768 × 1280	277 × 277	Lens.Screen-WXGA.png
720 × 1280	259 × 259	Lens.Screen-720p.png

The **`UriMapper`** class is required for working with lens apps. In fact, lens apps are opened using a special URI that has to be intercepted and managed. The following code is a sample **`Uri`**:

```
/MainPage.xaml?Action=ViewfinderLaunch
```

When this **`Uri`** is intercepted, users should be redirected to the application page that takes the picture. In the following sample, you can see a **`UriMapper`** implementation that redirects users to a page called **Camera.xaml** when the application is opened from the lens view.

```
public class MyUriMapper : UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        string tempUri = uri.ToString();
        if (tempUri.Contains("ViewfinderLaunch"))
        {
            return new Uri("/Camera.xaml", UriKind.Relative);
        }
        else
        {
            return uri;
        }
    }
}
```

Support sharing

If you've developed an application that supports photo sharing such as a social network client, you can integrate it in the **Share** menu of the Photos Hub. Users can find this option in the Application Bar in the photo details page.

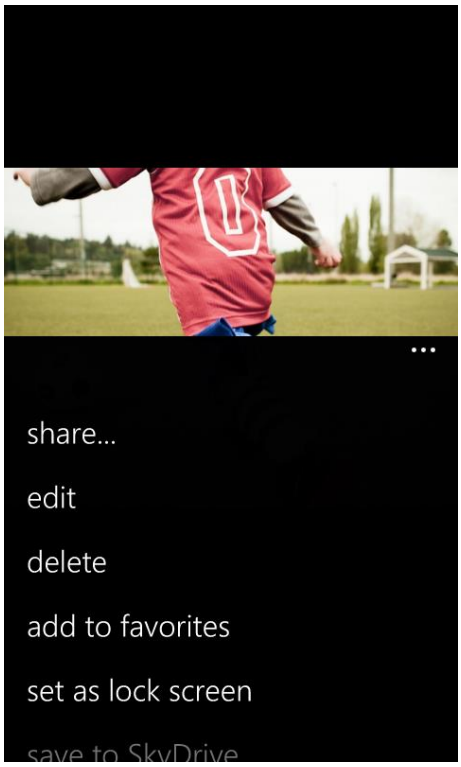

Figure 27: Share Option in the Picture Details Page

When users choose this option, Windows Phone displays a list of applications that are registered to support sharing. We can add our application to the list simply by adding a new extension in the manifest file, like we did to add lens support.

We have to manually add the following declaration in the **`Extensions`** section:

```
<Extensions>
  <Extension ExtensionName="Photos_Extra_Share" ConsumerID="{5B04B775-356B-
4AA0-AAF8-6491FFEA5632}" TaskID="_default" />
</Extensions>
```

When users choose our application from the list, it is opened with the following URI:

```
/MainPage.xaml?Action=ShareContent&FileId=%7B1ECC86A2-CDD7-494B-A9A8-
DBD9B4C4AAC7%7D
```

Again, we can use a **UriMapper** implementation to redirect users to our application's page that offers the sharing feature. It's also important to carry the **FiledId** parameter in this page; we're going to need it to know which photo has been selected by the user.

The following sample shows a **UriMapper** implementation that simply replaces the name of the original page (**MainPage.xaml**) with the name of the destination page (**SharePage.xaml**):

```
public class MyUriMapper: UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        string tempUri = uri.ToString();
        string mappedUri;

        if ((tempUri.Contains("SharePhotoContent")) &&
(tempUri.Contains("FileId")))
        {
            // Redirect to PhotoShare.xaml.
            mappedUri = tempUri.Replace("MainPage", "SharePage");
            return new Uri(mappedUri, UriKind.Relative);
        }

        return uri;
    }
}
```

After redirecting the user to the sharing page, we can use a method called **GetPictureFromToken()** exposed by the **MediaLibrary** class. It accepts the unique picture ID as a parameter and returns a reference to the **Picture** object that represents the image selected by the user.

The picture ID is the parameter called **FileId** that we received in the URI when the application was opened. In the following sample, you can see how we retrieve the parameter by using the **OnNavigatedTo** event which is triggered when the user is redirected to the sharing page, and use it to display the selected picture with an **Image** control.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("FileId"))
    {
        string fileId = NavigationContext.QueryString["FileId"];
        MediaLibrary library = new MediaLibrary();
        Picture picture = library.GetPictureFromToken(fileId);

        BitmapImage image = new BitmapImage();
        image.SetSource(picture.GetImage());
        ShareImage.Source = image;
```

```
    }
}
```

Other integration features

There are other ways to integrate our application with the Photos Hub. They all work the same way:

- A declaration must be added to the manifest file.
- The application is opened using a special **`Uri`** that you need to intercept with a **`UriMapper`** class.
- The user is redirected to a dedicated page in which you can retrieve the selected image by using the **`FileId`** parameter.

List the application as a photographic app

This is the simplest integration since it just displays the application in the Apps section of the Photos Hub. To support it, you simply have to add the following declaration in the manifest file:

```
<Extensions>
  <Extension ExtensionName="Photos_Extra_Hub" ConsumerID="{5B04B775-356B-
4AA0-AAF8-6491FFEA5632}" TaskID="_default" />
</Extensions>
```

Nothing else is required since this kind of integration will simply include a quick link in the Photos Hub. The application will be opened normally, as if it was opened using the main app icon.

Integrating with the edit option

Another option available in the Application Bar of the photo details page is called **edit**. When the user taps it, Windows Phone displays a list of applications that support photo editing. After choosing one, the user expects to be redirected to an application page where the selected picture can be edited.

The following declaration should be added in the manifest file:

```
<Extensions>
  <Extension ExtensionName="Photos_Extra_Image_Editor"
ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5632}" TaskID="_default" />
</Extensions>
```

When this feature is enabled, your application will be opened with the following URI:

```
/MainPage.xaml?Action=EditPhotoContent&FileId=%7B1ECC86A2-CDD7-494B-A9A8-
DBD9B4C4AAC7%7D
```

This is the **`Uri`** to intercept to redirect users to the proper page where you'll be able to retrieve the selected image by using the **`FileId`** parameter, like we did for the photo sharing feature.

Rich Media Apps

Rich media apps are applications that are able to take pictures and save them in the user's library. When users open one of these photos, they will see:

- Text under the photo with the message "**captured by**" followed by the app's name.
- A new option in the Application Bar called "**open in**" followed by the app's name.

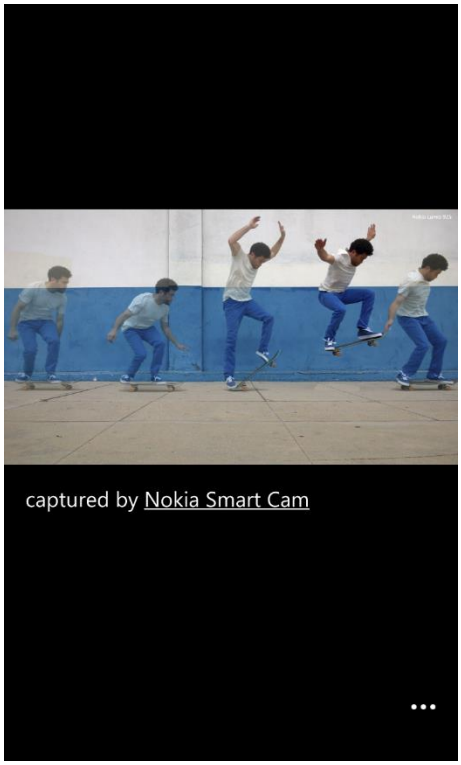

Figure 28: Photo Taken with a Rich Media App

This approach is similar to the sharing and editing features. The difference is that the rich media apps integration is available only for pictures taken within the application, while editing and sharing features are available for every photo, regardless of how they were taken.

The following declaration should be added in the manifest to enable rich media app integration:

```
<Extensions>
  <Extension ExtensionName="Photos_Rich_Media_Edit" ConsumerID="{5B04B775-
356B-4AA0-AAF8-6491FFEA5632}" TaskID="_default" />
</Extensions>
```

In this scenario, the application is opened with the following URI:

```
/MainPage.xaml?Action=RichMediaEdit&FileId=%7B1ECC86A2-CDD7-494B-A9A8-
DBD9B4C4AAC7%7D
```

As you can see, the URI is always the same; what changes is the value of the **`Action`** parameter—in this case, **`RichMediaEdit`**.

This is the URI you need to intercept with your **`UriMapper`** implementation. You'll need to redirect users to a page of your application that is able to manage the selected picture.

A quick recap

In this chapter, we've learned many ways to create a great multimedia application for Windows Phone by:

- Integrating camera features to take photos and record videos.
- Interacting with the media library to get access to pictures and audio.
- Integrating with the native camera experience to give users access to advanced features directly in the Photos Hub.

Chapter 9 Live Apps: Tiles, Notifications, and Multitasking

The multitasking approach

As we've seen in the application life cycle discussed in Chapter 3, applications are suspended when they are not in the foreground. Every running process is terminated, so the application can't execute operations while in the background.

There are three ways to overcome this limitation:

- **Push notifications**, which are sent by a remote service using an HTTP channel. This approach is used to send notifications to users, update a Tile, or warn users that something has happened.
- **Background agents**, which are services connected to our application that can run from time to time under specific conditions. These services can also be used for push notification scenarios—in this case, remote services are not involved—but they can also perform other tasks as long as they use supported APIs.
- **Alarms and reminders**, which display reminders to the user at specific dates and times.

Let's see in detail how they work.

Push notifications

Push notifications are messages sent to the phone that can react in many ways based on the notification type. There are three types of push notifications:

- **Raw notifications** can store any type of information, but they can be received only if the associated application is in the foreground.
- **Toast notifications** are the most intrusive ones, since they display a message at the top of the screen, along with a sound and a vibration. Text messages are a good example of toast notifications.
- **Tile notifications** can be used to update the application's Tile.

There are three factors involved in the push notification architecture:

- The Windows Phone application, which acts as a client to receive notifications.
- The server application, which can be a web application or a service, that takes care of sending the notifications. Usually, the server stores a list of all the devices that are registered to receive notifications.
- The **Microsoft Push Notification Service** (MPNS), which is a cloud service offered by Microsoft that is able to receive notifications from the server application and route them to the Windows Phone clients.

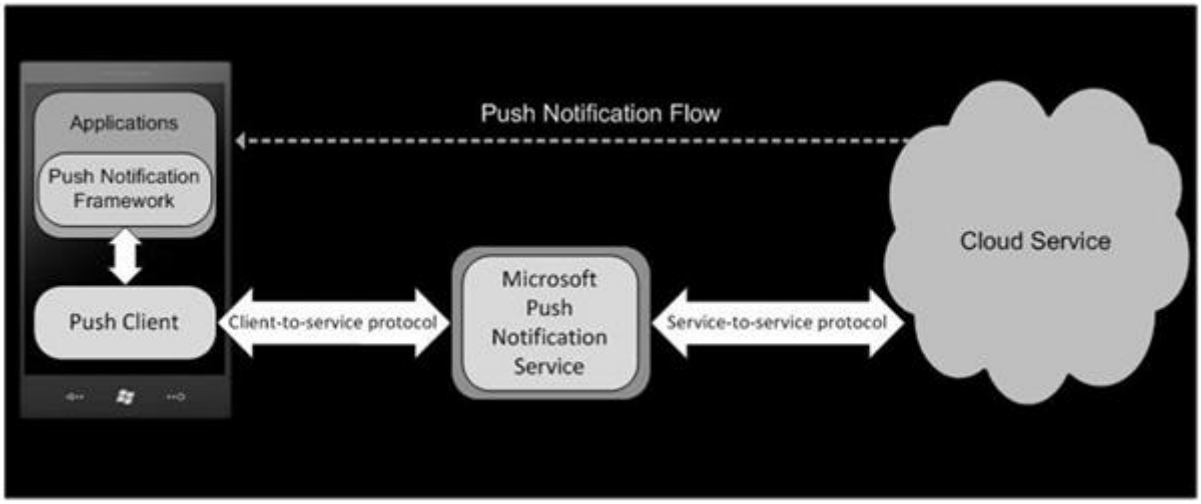

Figure 29: Microsoft's Push Notification Architecture

Every Windows Phone application receives push notifications using a **channel**, which is identified by a unique URI. The server application will send notifications to the registered clients by sending an XML string to this URI using a POST command. The MPNS will take care of routing the requests to the proper devices.

Here is a sample of a URI that represents a channel:

```
http://sn1.notify.live.net/throttledthirdparty/01.00/AAhsLicyiJgtTaidbJoSgm-
```

Note: MPNS usage is free, but limited to 500 notifications per day per single device. If you need to exceed this limitation, you have to buy a TLS digital certificate, which you'll need to submit during the certification process and to digitally sign your server's application. This way, you'll also be able to support SSL to encrypt the notification's channel.

Sending a notification: The server

As already mentioned, notifications are sent using an HTTP channel with a POST command. The benefit is that it relies on standard technology, so you'll be able to create a server application with any development platform.

The HTTP request that represents a notification has the following features:

- It's defined using XML, so the content type of the request should be **`text/xml`**.
- A custom header called **`X-WindowsPhone-Target`**, which contains the notification's type (toast, Tile, or raw).
- A custom header called **`X-NotificationClass`**, which is the notification's priority (we'll discuss this more in-depth later).

Let's see how the different push notifications are structured in detail.

Toast notifications

The following sample shows the XML needed to send a toast notification:

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification">
  <wp:Toast>
    <wp:Text1>Title</wp:Text1>
    <wp:Text2>Text</wp:Text2>
    <wp:Param>/MainPage.xaml?ID=1</wp:Param>
  </wp:Toast>
</wp:Notification>
```

There are three parameters to set:

- **wp:Text1** is the notification's title.
- **wp:Text2** is the notification's text.
- **wp:Param** is the optional notification deep link; when this is set, the application is opened automatically on the specified page with one or more query string parameters that can be used to identify the notification's context.

When you prepare the request to send over HTTP, the **X-WindowsPhone-Target** header should be set to **toast**, while the **X-NotificationClass** header supports the following values:

- **2** to send the notification immediately.
- **12** to send the notification after 450 seconds.
- **22** to send the notification after 900 seconds.

Figure 30: A Toast Notification

Tile notifications

Tile notifications are used to update either the main Tile or one of the secondary Tiles of the application. We won't describe the XML needed to send the notification here: Tiles are more complex than the other notification types since Windows Phone 8 supports many templates and sizes. We'll look at the XML that describes Tile notifications later in the Tiles section of the chapter.

To send a Tile notification, the **X-WindowsPhone-Target** header of the HTTP request should be set to **tile**, while the **X-NotificationClass** header supports the following values:

- **1** to send the notification immediately.
- **11** to send the notification after 450 seconds.
- **21** to send the notification after 900 seconds.

Raw notifications

Raw notifications don't have a specific XML definition since they can deliver any kind of data, so we can include our own definition.

To send a raw notification, the **X-WindowsPhone-Target** header of the HTTP request should be set to **raw**, while the **X-NotificationClass** header supports the following values:

- 3 to send the notification immediately.
- 13 to send the notification after 450 seconds.
- 23 to send the notification after 900 seconds.

Sending the request and managing the response

The following sample code shows an example of how to send a toast notification using the **HttpWebRequest** class, one of the basic .NET Framework classes for performing network operations:

```
string toastNotificationPayloadXml = "<?xml version=\"1.0\" encoding=\"utf-
8\"?>" +
                                     "<wp:Notification
xmlns:wp=\"WPNotification\">" +
                                     "<wp:Toast>" +
                                     "<wp:Text1> title </wp:Text1>" +
                                     "<wp:Text2> text </wp:Text2>" +
                                     "</wp:Toast> " +
                                     "</wp:Notification>";

byte[] payload = Encoding.UTF8.GetBytes(toastNotificationPayloadXml);

var pushNotificationWebRequest =
(HttpWebRequest)WebRequest.Create("http://sn1.notify.live.net/throttledthir
dparty/01.00/AAEqbi-
clyknR6iysF1QNBFpAgAAAAADAQAAAAQUZm52OkJCMjg1QTg1QkZDMkUxREQ");

pushNotificationWebRequest.Method = "POST";
pushNotificationWebRequest.ContentType = "text/xml";

var messageId = Guid.NewGuid();

pushNotificationWebRequest.Headers.Add("X-MessageID",
messageId.ToString());
```

```
pushNotificationWebRequest.Headers.Add("X-WindowsPhone-Target", "toast");
pushNotificationWebRequest.Headers.Add("X-NotificationClass", "2");

pushNotificationWebRequest.ContentLength = payload.Length;

using (var notificationRequestStream =
pushNotificationWebRequest.GetRequestStream())
{
    notificationRequestStream.Write(payload, 0, payload.Length);
}

using (var pushNotificationWebResponse =
(HttpWebResponse)pushNotificationWebRequest.GetResponse())
{
     //Check the status of the response.
}
```

The XML definition is simply stored in a **`string`**. We're going to change just the node values that store the notification's title and text. Then, we start to prepare the HTTP request by using the **`HttpWebRequest`** class. We add the custom headers, define the content's length and type (**`text/xml`**), and specify the method to use (**`POST`**).

In the end, by using the **`GetRequestStream()`** method, we get the stream location to write the request's content, which is the notification's XML. Then we send it by calling the **`GetResponse()`** method, which returns the status of the request. By analyzing the response we are able to tell whether or not the operation was successful.

The response's analysis involves the status code and three custom headers:

- The response's status code returns generic information that tells you whether the request has been received. It's based on the standard HTTP status codes. For example, **200 OK** means that the request has been successfully received, while **404 Not Found** means that the URI was invalid.
- The **`X-NotificationStatus`** header tells you if the MPNS has received the request using the values **`Received`**, **`Dropped`**, **`QueueFull`**, and **`Supressed`**.
- The **`X-DeviceConnectionStatus`** header returns the device status when the request is sent: **`Connected`**, **`Inactive`**, **`Disconnected`**, or **`TempDisconnected`**.
- The **`X-SubscriptionStatus`** header returns if the channel is still valid (**`Active`**) or not (**`Expired`**). In the second case, we shouldn't try to send it again, since it doesn't exist anymore.

The combination of these parameters will help you understand the real status of the operation. The MSDN documentation features descriptions of all the possible combinations.

It's important to correctly manage the notifications because MPNS doesn't offer any automatic retry mechanism. If a notification is not delivered, MPSN won't try again to send it, even if the operation failed for a temporary reason (for example, the device wasn't connected to the Internet). It's up to you to implement a retry mechanism based on the response.

PushSharp: A push notification helper library

As you can see, sending push notifications is a little bit tricky since it requires you to manually set headers, XML strings, etc. Some developers have worked on wrappers that hide the complexity of manually defining the notification by exposing high-level APIs so that you can work with classes and objects.

One of the most interesting wrappers is called PushSharp, which can be simply installed on your server project using NuGet. The biggest benefits of this library are:

- It's a generic .NET library that supports not only Windows Phone, but the most common platforms that use push notifications, like Windows Store apps, iOS, Android, and Blackberry. If you have a cross-platform application, it will make your life easier in managing a single-server application that is able to send notifications to different kinds of devices.
- It's totally compatible with Windows Phone 8, so it supports not only toast and raw notifications, but also all the new Tile templates and sizes.

The following sample shows how simple it is to send a toast notification using this library:

```
WindowsPhoneToastNotification notification = new
WindowsPhoneToastNotification();
notification.Text1 = "Title";
notification.Text2 = "Text";
notification.EndPointUrl =
"http://sn1.notify.live.net/throttledthirdparty/01.00/AQHcej5duTcJRqnn779so
TA1AgAAAAADAQAAAAQUZm52OkJCMjg1QTg1QkZDMkUxREQFBkxFR0FDWQ";
notification.NotificationClass=BatchingInterval.Immediate;
PushBroker broker = new PushBroker();
broker.RegisterWindowsPhoneService();
broker.QueueNotification(notification);
```

Every notification type is represented by a specific class, which exposes a property for every notification feature. In the previous sample, the **`WindowsPhoneToastNotification`** class offers properties to set the notification's title, text, and deep link.

The channel URI location to send the notification is set in the **`EndPointUrl`** property. Once everything is set, you can send it by creating a **`PushBroker`** object, which represents the dispatcher that takes care of sending notifications. First, you have to register for the kind of notifications you want to send. Since we're working with Windows Phone, we use the **`RegisterWindowsPhoneService()`** method. Then, we can queue the notification by simply passing it to the **`QueueNotification()`** method. It will be automatically sent with the priority you've set.

The approach is the same if you want to send a Tile. You have three different classes based on the Tile's template, **`WindowsPhoneCycleTileNotification`**, **`WindowsPhoneFlipTileNotification`**, and **`WindowsPhoneIconicTileNotification`**; or **`WindowsPhoneRawNotification`** for a raw notification.

In the end, the **PushBroker** class exposes many events to control the notification life cycle, like **OnNotificationSent** which is triggered when a notification is successfully sent, or **OnNotificationFailed** which is triggered when the sending operation has failed.

Receiving push notifications: The client

The base class that identifies a push notification channel is called **HttpNotificationChannel** and exposes many methods and events that are triggered when something connected to the channel happens.

Note: To receive push notifications you'll need to enable the ID_CAP_PUSH_NOTIFICATION capability in the manifest file.

Every application has a single unique channel, identified by a keyword. For this reason, it should be created only the first time the application subscribes to receive notifications; if you try to create a channel that already exists, you'll get an exception. To avoid this scenario, the **HttpNotificationChannel** class offers the **Find()** method, which returns a reference to the channel.

```
private void OnRegisterChannelClicked(object sender, RoutedEventArgs e)
{
    HttpNotificationChannel channel =
HttpNotificationChannel.Find("TestChannel");
    if (channel == null)
    {
        channel = new HttpNotificationChannel("TestChannel");
        channel.Open();
    }
}
```

In the previous sample, the channel is created only if the **Find()** method fails and returns a null object. The **HttpNotificationChannel** class exposes many methods to start interacting with push notifications; they should be called only if the channel doesn't already exist. In the sample we see the **Open()** method which should be called to effectively create the channel, and which automatically subscribes to raw notifications.

If we want to be able to receive toast and Tile notifications, we need to use two other methods offered by the class: **BindToShellToast()** and **BindToShellTile()**. The following sample shows a complete initialization:

```
private void OnRegisterChannelClicked(object sender, RoutedEventArgs e)
{
    HttpNotificationChannel channel =
HttpNotificationChannel.Find("TestChannel");
```

```
    if (channel == null)
    {
        channel = new HttpNotificationChannel("TestChannel");
        channel.Open();
        channel.BindToShellToast();
        channel.BindToShellTile();
    }
}
```

Beyond offering methods, the **HttpNotificationChannel** class also offers some events to manage different conditions that can be triggered during the channel life cycle.

The most important one is called **ChannelUriUpdated**, which is triggered when the channel creation operation is completed and the MPNS has returned the URI that identifies it. This is the event in which, in a regular application, we will send the URI to the server application so that it can store it for later use. It's important to subscribe to this event whether the channel has just been created, or already exists and has been retrieved using the **Find()** method. From time to time, the URI that identifies the channel can expire. In this case, the **ChannelUriUpdated** event is triggered again to return the new URI.

The following sample shows a full client initialization:

```
private void OnRegisterChannelClicked(object sender, RoutedEventArgs e)
{
    HttpNotificationChannel channel =
HttpNotificationChannel.Find("TestChannel");
    if (channel == null)
    {
        channel = new HttpNotificationChannel("TestChannel");
        channel.Open();
        channel.BindToShellToast();
        channel.BindToShellTile();
    }

    channel.ChannelUriUpdated += channel_ChannelUriUpdated;
}

void channel_ChannelUriUpdated(object sender, NotificationChannelUriEventArgs
e)
{
    MessageBox.Show(e.ChannelUri);
}
```

As you can see, the **ChannelUriUpdated** event returns a parameter with the **ChannelUri** property, which contains the information we need. In the previous sample, we just display the URI channel to the user.

There are two other events offered by the **HttpNotificationChannel** class that can be useful:

- **HttpNotificationReceived** is triggered when the application has received a raw notification.
- **ShellToastNotificationReceived** is triggered when the application receives a toast notification while it is open. By default, toast notifications are not displayed if the associated application is in the foreground.

The **HttpNotificationReceived** event receives, in the parameters, the object that identifies the notification. The content is stored in the **Body** property, which is a stream since raw notifications can store any type of data. In the following sample, we assume that the raw notification contains text and display it when it's received:

```
void Channel_HttpNotificationReceived(object sender,
HttpNotificationEventArgs e)
{
    using (StreamReader reader = new StreamReader(e.Notification.Body))
    {
        string message = reader.ReadToEnd();
        Dispatcher.BeginInvoke(() => MessageBox.Show(message));
    }
}
```

The **ShellNotificationReceived** event, instead, returns in the parameters a **Collection** object, which contains all the XML nodes that are part of the notification. The following sample shows you how to extract the title and the description of the notification, and how to display them to the user:

```
void Channel_ShellToastNotificationReceived(object sender,
NotificationEventArgs e)
{
    string title = e.Collection["wp:Text1"];
    string message = e.Collection["wp:Text2"];
    Dispatcher.BeginInvoke(() => MessageBox.Show(title + " " + message));
}
```

Managing errors

If something goes wrong when you open a notification channel, you can subscribe to the **ErrorOccurred** event of the **HttpNotificationChannel** class to discover what's happened.

The event returns a parameter that contains information about the error, like **ErrorType**, **ErrorCode**, **ErrorAdditionalData**, and **Message**.

The following list includes the most common conditions that can lead to a failure during the channel opening:

- To preserve battery life and performance, Windows Phone limits the maximum number of channels that are kept alive at the same time. If the limit has been reached and you try to open a new channel, you'll get the value **ChannelOpenFailed** as **ErrorType**.
- The received notification can contain a message which is badly formatted; in this case the **ErrorType** will be **MessageBadContent**.
- You can send too many notifications at the same time; in this case, they are rejected with the **NotificationRateTooHigh** error.
- To preserve battery power, notifications can be received only if the battery isn't critical; in this case, you'll get a **PowerLevelChanged** error.

The **ErrorAdditionalData** property can contain additional information about the error. For example, if you get a **PowerLevelChanged** error, you'll be informed of the current battery level (low, critical, or normal).

```
void channel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
{
    if (e.ErrorType == ChannelErrorType.PowerLevelChanged)
    {
        ChannelPowerLevel level = (ChannelPowerLevel)
e.ErrorAdditionalData;
        switch (level)
        {
            case ChannelPowerLevel.LowPowerLevel:
                MessageBox.Show("Battery is low");
                break;
            case ChannelPowerLevel.CriticalLowPowerLevel:
                MessageBox.Show("Battery is critical");
                break;
        }
    }
}
```

Background agents

Push notifications are the best way to interact with the user when the application is not running since they offer the best experience and, at the same time, preserve battery life. However, the experience is limited to notifications: you can't execute any other operation, like fetching data from a web service or reading a file from the local storage. Moreover, for certain scenarios in which you don't require instant notifications, creating the required server infrastructure can be too expensive. Think, for example, of a weather application: it's not critical that the forecast is updated immediately when the forecasts change.

For all these scenarios, Windows Phone 7.5 has introduced background agents, which are special services periodically executed by Windows Phone, even when the application is not running. There are two types of periodic background agents: periodic and audio. In the **New Project** section of Visual Studio, you'll find many templates for all the supported agent types. In this section we'll see how periodic agents work in detail.

Tip: Even if a background agent is a separate Visual Studio project, it shares the same resources with the foreground application. For example, they share the same local storage, so you're able to read data created by the application in the agent, and vice versa.

Agent limits

There are some limitations that background agents have to satisfy. The most important one is connected to timing, since agents can run only in a specific time frame for a limited amount of time. We'll discuss this limitation later since there are some differences according to the background agent type you're going to use.

The first limitation concerns supported **APIs**: only a limited number of APIs can be used in a background agent. Basically, all the APIs that are related to the user interface are prohibited since agents can't interact with the application interface. You can find the complete list of unsupported APIs in the MSDN documentation.

The second limitation is about **memory**: a background agent can't use more than 11 MB of memory, otherwise it will be terminated. It's important to highlight that during the testing process (when the Visual Studio debugger is attached), the memory limit will be disabled, and the background agent won't be terminated if it has used more than 11 MB. You'll have to test it in a real environment if you want to make sure the limit isn't reached.

The third and final limitation is about **timing**: a background agent is automatically disabled 14 days after it has been initialized by the connected application. There are two ways to overcome this limitation:

- The user keeps using the application; the agent can be renewed for another 14 days every time the application is opened.
- The agent is used to send notifications to update the main application's Tile or the lock screen; every time the agent sends a notification it will be automatically renewed for another 14 days.

It's important to keep in mind that if the background agent execution consecutively fails twice (because it exceeded the memory limit or raised an unmanaged exception), it's automatically disabled; the application will have to reenable it when it's launched.

Periodic agents

Periodic agents are used when you need to execute small operations frequently. They are typically executed **every 30 minutes** (the execution interval can sometimes be shortened to every 10 minutes to coincide with other background processes to save battery life), and they can run for **up to 25 seconds**. Users are able to manage periodic agents from the **Settings** panel and disable the ones they don't need. Periodic agents are automatically disabled if the phone is running in Battery Saver mode; they'll be automatically restored when sufficient battery power is available.

Periodic agents are identified by the **`PeriodicTask`** class, which belongs to the **`Microsoft.Phone.Scheduler`** namespace.

Resource intensive agents

Resource intensive agents have been created for the opposite scenario: long-running tasks that are executed occasionally. They can run for up to 10 minutes, but only if the phone is connected to a Wi-Fi network and an external power source.

These agents are perfect for tasks like data synchronization. In fact, they are typically executed during the night, when the phone charging. Other than the previous conditions, in fact, the phone shouldn't be in use. The lock screen should be activated and no other operations (like phone calls) should be performing.

Resource intensive agents are identified by the **`ResourceIntensiveTask`**, which is also part of the **`Microsoft.Phone.Scheduler`** namespace.

Creating a background agent

As already mentioned, background agents are defined in a project separate from the front-end application. Periodic agents share the same template and architecture, and the Windows Phone application will decide to register them as **`PeriodicTask`** or **`ResourceIntensiveTask`** objects.

To create a background agent, you'll have to add a new project to the solution that contains your Windows Phone application. In the **Add New Project** window you'll find a template called **Windows Phone Scheduled Task Agent** in the Windows Phone section.

The project already contains the class that will manage the agent; it's called **`ScheduledAgent`** and it inherits from the **`ScheduledTaskAgent`** class. The class already implements a method and an event handler.

The method, called **`OnInvoke()`**, is the most important one. It's the method that is triggered when the background agent is executed, so it contains the logic that performs the operations we need. The following sample shows how to send a toast notification from a background agent:

```
protected override void OnInvoke(ScheduledTask task)
{
    ShellToast toast = new ShellToast();
```

```
    toast.Title = "Title";
    toast.Content = "Text";
    toast.Show();

    NotifyComplete();
}
```

It's important to highlight the **NotifyComplete()** method, which should be called as soon as the agent has completed all the operations. It notifies the operating system that the task has completed its job and that the next scheduled task can be executed. The **NotifyComplete()** method determines the task's status. If it's not called within the assigned time—25 seconds for periodic tasks or 10 minutes for resource intensive tasks—the execution is interrupted.

There's another way to complete the agent's execution: **Abort()**. This method is called when something goes wrong (for example, the required conditions to execute the agent are not satisfied) and the user needs to open the application to fix the problem.

The event handler is called **UnhandledException** and is triggered when an unexpected exception is raised. You can use it, for example, to log the error.

The previous sample shows you how to send local toast notifications. A toast notification is identified by the **ShellToast** class. You simply have to set all the supported properties (**Title**, **Content**, and optionally **NavigationUri**, which is the deep link). In the end, you have to call the **Show()** method to display it.

Like remote notifications, local toasts are supported only if the application is in the background. The previous code works only inside a background agent. If it's executed by a foreground application, nothing happens.

Registering the agent

The background agent is defined in a separate project, but is registered in the application. The registration should be done when the application starts, or in the settings page if we give users the option to enable or disable it within the application.

The base class to use when working with background agents is **ScheduledActionService**, which represents the phone's scheduler. It takes care of registering all the background agents and maintaining them during their life cycle.

The first step is to define which type of agent you want to use. As previously mentioned, the background agent architecture is always the same; the type (periodic or resource intensive) is defined by the application.

In the first case you'll need to create a **PeriodicTask** object, and in the second case, a **ResourceIntensive** task object. Regardless of the type, it's important to set the **Description** property, which is text displayed to users in the Settings page. It's used to explain the purpose of the agent so users can decide whether or not to keep it enabled.

```
PeriodicTask periodicTask = new PeriodicTask("PeriodicTask");
periodicTask.Description = "This is a periodic task";

ResourceIntensiveTask resourceIntensiveTask = new
ResourceIntensiveTask("ResourceIntensiveTask");
resourceIntensiveTask.Description = "This is a resource intensive task";
```

In both cases, background agents are identified by a name, which is passed as a parameter of the class. This name should be unique across all the tasks registered using the **PhoneApplicationService** class; otherwise you'll get an exception.

The basic operation to add a task is very simple:

```
public void ScheduleAgent()
{
    ScheduledAction action = ScheduledActionService.Find("Agent");
    if (action == null || !action.IsScheduled)
    {
        if (action != null)
        {
            ScheduledActionService.Remove("Agent");
        }

        PeriodicTask task = new PeriodicTask("Agent");
        task.Description = "This is a periodic agent";
        ScheduledActionService.Add(task);
        #if DEBUG
        ScheduledActionService.LaunchForTest("Agent",
TimeSpan.FromSeconds(10));
        #endif
     }
}
```

The first operation checks whether the agent is already scheduled by using the **Find()** method of the **ScheduledActionService** class, which requires the task's unique name. This operation is required if we want to extend the agent's lifetime. If the agent does not exist yet or is not scheduled (the **IsScheduled** property is false), we first remove it from the scheduler and then add it since the **ScheduledActionService** class doesn't offer a method to simply update a registered task. The add operation is done using the **Add()** method, which accepts either a **PeriodicTask** or a **ResourceIntensiveTask** object.

Now the task is scheduled and will be executed when the appropriate conditions are satisfied. If you're in the testing phase, you'll find the **LaunchForTest()** method useful; it forces the execution of an agent after a fixed amount of time. In the previous sample, the agent identified by the name **PeriodicTask** is launched after five seconds. The **LaunchForTest()** method can also be executed in the **OnInvoke()** event inside the background agent, allowing you to easily simulate multiple executions.

In the previous sample you can see that we've used conditional compilation to execute the **`LaunchForTest()`** method only when the application is launched in debug mode. This way, we make sure that when the application is compiled in release mode for publication to the Windows Store, the method won't be executed; otherwise, you'll get an exception if the method is called by an application installed from the Store.

Managing errors

Background agents are good examples of the philosophy behind Windows Phone:

- Users are always in control; they can disable whatever background agents they aren't interested in through the Settings page.
- Performance and battery life are two crucial factors; Windows Phone limits the maximum number of registered background agents.

For these reasons, the agent registration process can fail, so we need to manage both scenarios. The following code shows a more complete sample of a background agent's initialization:

```
public void ScheduleAgent()
{
    ScheduledAction action = ScheduledActionService.Find("Agent");
    if (action == null || !action.IsScheduled)
    {
        if (action != null)
        {
            ScheduledActionService.Remove("Agent");
        }

        try
        {
            PeriodicTask task = new PeriodicTask("Agent");
            task.Description = "This is a periodic agent";
            ScheduledActionService.Add(task);
        }
        catch (InvalidOperationException exception)
        {
            if (exception.Message.Contains("BNS Error: The action is
disabled"))
            {
                // No user action required.
            }

            if (exception.Message.Contains("BNS Error: The maximum number
of ScheduledActions of this type have already been added."))
            {
                // No user action required.
            }
```

```
        }
    }
}
```

The difference in the previous sample is that the **`Add()`** operation is executed inside a **`try / catch`** block. This way, we are ready to catch the **`InvalidOperationException`** error that might be raised.

We can identify the scenario by the exception message:

- **BNS Error: The action is disabled**. The user has disabled the agent connected to our application in the Settings page. In this case, we have to warn the user to enable it again in the Settings page before trying to register it.
- **BSN Error: The maximum number of ScheduledActions of this type have already been added**. The user has reached the maximum number of agents allowed to be installed on phone. In this case, we don't have to do anything; Windows Phone will display a proper warning message.

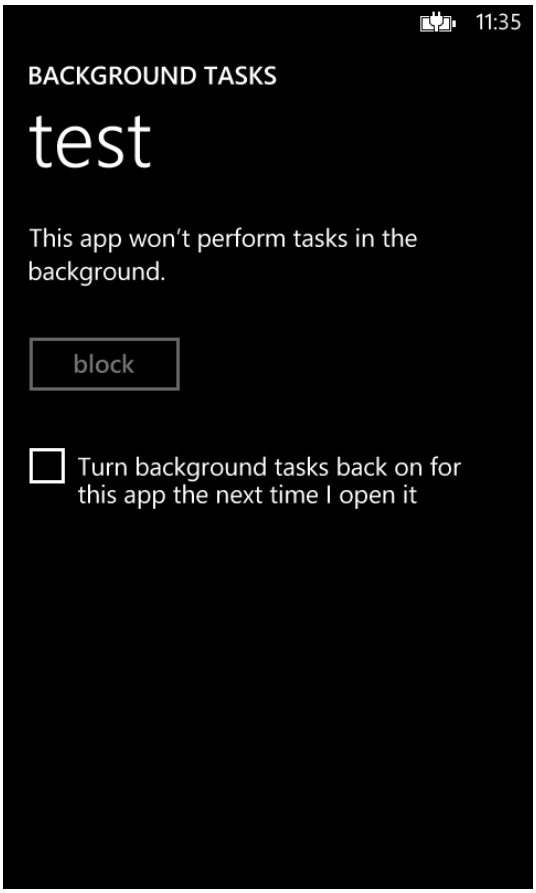

Figure 31: A Background Agent Disabled by the User in the Settings Page

Moreover, the **`ScheduledTask`** class (which is the base class that **`PeriodicTask`** and **`ResourceIntensiveTask`** inherit from) offers some properties for understanding the status of the last execution, such as **`LastScheduledTime`** which contains the date and time of the last execution, and **`LastExitReason`** which stores the last execution status.

Specifically, **`LastExitReason`** is very useful for knowing if the last execution completed successfully (**`Completed`**), if it exceeded the memory limit (**`MemoryQuotaExceeded`**) or the time limit (**`ExecutionTimeExceeded`**), or if an unhandled exception occurred (**`UnhandledException`**).

Background audio agent

There's a special kind of background agent that works differently than periodic agents: audio agents, which are used in audio-related applications to keep playing audio when the app is closed. The goal is to offer a similar experience to the native Music + Videos Hub; even when the app is not in the foreground, users are able to keep listening to their music library.

Figure 32: A Background Audio Player

Again, the background agent is defined in a different project than the foreground application. However:

- The agent doesn't need to be initialized in the foreground application using the **`ScheduledActionService`** class like we did for periodic agents.
- There aren't time limitations. The agent runs every time users interact with the music controls, and it never expires. The only limitation is that the triggered operation should complete within 30 seconds.

- There is a memory limitation, but the cap is higher: 20 MB (keep in mind that the memory limit isn't activated when the Visual Studio debugger is connected).
- In this scenario, the background agent is not just a companion, but the core of the application; it manages all interactions with the music playback, regardless of whether they occur in the foreground application or the native embedded player.

Interacting with the audio

The core class to reproduce background audio is called **`BackgroundAudioPlayer`**, which identifies the built-in Windows Phone audio player. There's just one instance of the player within the system, and it can't be shared. If users launch another application that uses a background audio agent (including the native Music + Videos Hub), it takes control over the audio reproduction. As we're going to see soon, the **`BackgroundAudioPlayer`** class is used both in the foreground app and in the background agent to interact with the music playback.

The audio tracks played by a background audio agent are represented by the **`AudioTrack`** class. Each track, other than the resource to play, contains all the metadata like the title, artist, and album title.

The track's path is set in the **`Source`** property, which can be either a remote file or a file stored in the local storage. However, most of the properties can be set directly when the **`AudioTrack`** object is created, like in the following sample:

```
AudioTrack track = new AudioTrack(new
Uri("http://www.windowsphonelounge.net/issues/wpl_issue_01.mp3",
UriKind.Absolute), "Episode 1", "Igor & Matteo", "Windows Phone Lounge",
null);
```

With the previous code, in addition to setting the source file, we also immediately set information like the title, the artist, and the album. A useful available property is called **`PlayerControls`**, which can be used to set which controls (Play, Pause, Forward, etc.) are available for the track. This way, if you're developing an application connected to an online radio, for example, you can automatically block options that are not supported (like the skip track button).

Creating the agent

Visual Studio offers two templates to create background audio agents: **Windows Phone Audio Playback Agent** and **Windows Phone Audio Streaming** agent. They share the same purpose; their difference is that the **Windows Phone Audio Streaming** agent is required for working with media streaming codecs that are not natively supported by the platform.

A background audio agent's project already comes with a class called **`AudioAgent`**, which inherits from the **`AudioPlayerAgent`** class. As we saw with periodic agents, the class automatically implements some methods that are used to interact with the agent. The most important ones are **`OnUserAction()`** and **`OnPlayStateChanged()`**.

OnUserAction() is triggered every time users manually interact with the music playback, such as pausing a track or pressing the skip track button in the foreground application or the background player.

The method returns some parameters that can be used to understand the context and perform the appropriate operations:

- A **BackgroundAudioPlayer** object, which is a reference to the background audio player.
- An **AudioTrack** object, which is a reference to the track currently playing.
- A **UserAction** object, which is the action triggered by the user.

The following sample shows a typical implementation of the **OnUserAction()** method:

```
protected override void OnUserAction(BackgroundAudioPlayer player,
AudioTrack track, UserAction action, object param)
{
    switch (action)
    {
        case UserAction.Pause:
        {
            player.Pause();
            break;
        }
        case UserAction.Play:
            {
                player.Play();
                break;
            }
        case UserAction.SkipNext:
            {
                //Play next track.
                break;
            }
        case UserAction.SkipPrevious:
            {
                //Play previous track.
                break;
            }
    }
    NotifyComplete();
}
```

Usually with a **switch** statement, you'll monitor every supported user interaction, which is stored in the **UserAction** object. Then, you respond using the methods exposed by the **BackgroundAudioPlayer** class. **Play** and **Pause** are the simplest states to manage; **SkipNext** and **SkipPrevious** usually require more logic, since you have to get the previous or next track to play in the list from your library.

Note that background audio agents also require the **`NotifyComplete()`** method execution as soon as we've finished to manage the operation; it should be called within 30 seconds to avoid termination.

The **`OnPlayStateChanged()`** method is triggered automatically every time the music playback state is changed, but not as a direct consequence of a manual action. For example, when the current track ends, the agent should automatically start playing the next track in the list.

The method's structure is very similar to the **`OnUserAction()`** method. In addition to a reference to the background player and the current track in this case, you'll get a **`PlayState`** object, which notifies you about what's going on.

The following sample shows a typical implementation of the method:

```
protected override void OnPlayStateChanged(BackgroundAudioPlayer player,
AudioTrack track, PlayState playState)
{
    if (playState == PlayState.TrackEnded)
        //Play next track.

    NotifyComplete();
}
```

Tip: Background audio agents are not kept in memory all the time, but instead are launched only when the music playback state changes. If you need to persist some data across the different executions, you'll need to rely on the local storage.

The foreground application

We've seen how all the main playback logic is managed directly by the background agent. The foreground application, in most of the cases, is just a visual frontend for the agent.

To understand the playback state (and to properly update the UI) we need to use, again, the **`BackgroundAudioPlayer`** class we've seen. The difference is that, in the foreground application, we need to use the **`Instance`** singleton to get access to it.

The methods exposed by the class are the same, so we can use it to play, pause, or change the music playback state (for example, if we want to connect these operations to input controls like buttons).

The **`BackgroundAudioPlayer`** exposes an important event called **`PlayStateChanged`**, which is triggered every time the playback state changes. We can use it to update the visual interface (for example, if we want to display the track currently playing).

The following sample shows how the **`PlayStateChanged`** event is used to change the behavior of the play/pause button and to display to some metadata about the currently playing track:

```
public partial class MainPage : PhoneApplicationPage
{
    private BackgroundAudioPlayer player;

    // Constructor
    public MainPage()
    {
        InitializeComponent();
        player = BackgroundAudioPlayer.Instance;
        player.PlayStateChanged += new
EventHandler(player_PlayStateChanged);
    }

    private void OnPlayClicked(object sender, RoutedEventArgs e)
    {
        player.Play();
    }

    void player_PlayStateChanged(object sender, EventArgs e)
    {
        if (player.PlayerState == PlayState.Playing)
            Dispatcher.BeginInvoke(() => btnPlay.Content = "Pause");
        else
            Dispatcher.BeginInvoke(() => btnPlay.Content = "Play");

        if (player.PlayerState == PlayState.Playing)
        {

            Dispatcher.BeginInvoke(() =>
            {
                txtTitle.Text = player.Track.Title;
                txtArtist.Text = player.Track.Artist;
                txtAlbum.Text = player.Track.Album;
            });
        }
    }
}
```

The previous code should be familiar; you have access to all the properties we've seen in the background agent, like **`PlayerState`** to identify the current playback state, or **`Track`** to identify the currently playing track. **`Track`** isn't just a read-only property. If we want to set a new track to play in the application, we can simply assign a new **`AudioTrack`** object to the **`Track`** property of the **`BackgroundAudioPlayer`** instance.

Alarms and reminders

Alarms and reminders are simple ways to show reminders to user sat a specified date and time, like the native Alarm and Calendar applications do.

They work in the same way. The APIs belong to the **Microsoft.Phone.Scheduler** namespace, and they inherit from the base **ScheduledNotification** class. There are some properties in common between the two APIs:

- **Content**: The reminder description.
- **BeginTime**: The date and time the reminder should be displayed.
- **RecurrenceType**: Sets whether it's a recurrent or one-time reminder.
- **ExpirationTime**: The date and time a recurrent reminder expires.

Every reminder is identified by a name, which should be unique across all the alarms and reminders created by the application. They work like background agents; their life cycle is controlled by the **ScheduledActionService** class, which takes care of adding, updating, and removing them.

Alarms are identified by the **Alarm** class and used when to show a reminder that doesn't have a specific context. Users will be able to snooze or dismiss it. A feature specific to alarms is that they can play a custom sound, which is set in the **Sound** property.

The following sample shows how to create and schedule an alarm:

```
private void OnSetAlarmClicked(object sender, RoutedEventArgs e)
{
    Alarm alarm = new Alarm("Alarm")
    {
        BeginTime = DateTime.Now.AddSeconds(15),
        Content = "It's time!",
        RecurrenceType = RecurrenceInterval.None,
        Sound = new Uri("/Assets/CustomSound.mp3", UriKind.Relative)
    };

    ScheduledActionService.Add(alarm);
}
```

The sample creates an alarm that is scheduled 15 seconds after the current date and time, and uses a custom sound that is an MP3 file inside the Visual Studio project.

Reminders, instead, are identified by the **Reminder** class and are used when the notification is connected to a specific context, in a similar way that calendar reminders are connected to an appointment.

The context is managed using the **NavigationUri** property, which supports a deep link. It's the page (with optional query string parameters) that is opened when users tap the reminder's title.

```
private void OnSetReminderClicked(object sender, RoutedEventArgs e)
{
    Reminder reminder = new Reminder("Reminder")
    {
        BeginTime = DateTime.Now.AddSeconds(15),
        Title = "Reminder",
        Content = "Meeting",
        RecurrenceType = RecurrenceInterval.None,
        NavigationUri = new Uri("/DetailPage.xaml?id=1", UriKind.Relative)
    };

    ScheduledActionService.Add(reminder);
}
```

The previous code schedules a reminder that opens a page called **DetailPage.xaml**. Using the navigation events described in Chapter 3, you'll be able to get the query string parameters and load the requested data. Notice also that the **`Reminder`** class offers a **`Title`** property, which is not supported by alarms.

Live Tiles

Live Tiles are, without a doubt, the most unique Windows Phone feature, and one you won't find on any other platform. They are called Live Tiles because they aren't simply shortcuts to open applications; they can be updated with local or remote notifications to display information without forcing users to open the application. Many kinds of applications take advantage of this feature, like weather apps that display the forecast, news apps that display the latest headlines, and movie apps that display upcoming movie titles.

Windows Phone 8 has introduced many new features regarding Tiles, like new templates and new sizes.

An application can use three different sizes for Tiles: small, medium, and wide. As developers, we'll be able to customize the Tile's content according to the size so that, for example, the wide Tile can display more info than the small Tile.

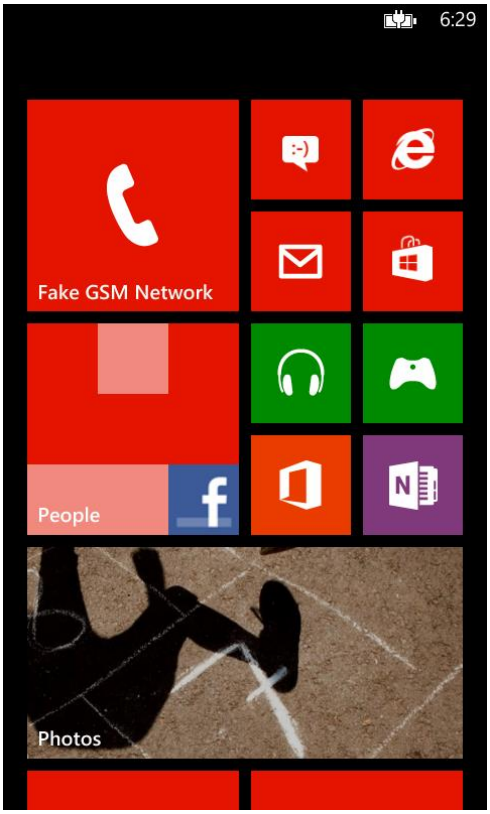

Figure 33: Various Tile Sizes

Windows Phone 8 has also introduced three different templates to customize a Tile: flip, cycle, and iconic. It's important to note that you can choose only one template for your application; it must be declared in the manifest file, in the **Application UI** section. Once you've set it, you won't be able to change it at run time, and all the Tiles you're going to create or update will have to use that template. In addition, you can choose the features (Tiles, pictures, etc.) to use for the main Tile in the **Application UI** section; this information will be used until a notification updates it.

In the following sections we'll examine every available template in detail. For each one we'll discuss the architecture and code needed to update it with a notification. For remote notifications, we'll see the required XML. For local notifications, we'll look at the APIs to use in the application or in a background agent.

In both cases, all the fields that define a Tile are optional. If you don't set some of them, those properties will simply be ignored. On the other hand, if a field that was previously set is not updated with a notification, the old value will be kept.

Flip template

Flip is the standard Windows Phone template, and the only one that was already available in Windows Phone 7. With this template you can display text, counters, and images on the front of the Tile. Periodically, the Tile will rotate or "flip" to show the opposite side, which can display different text or images.

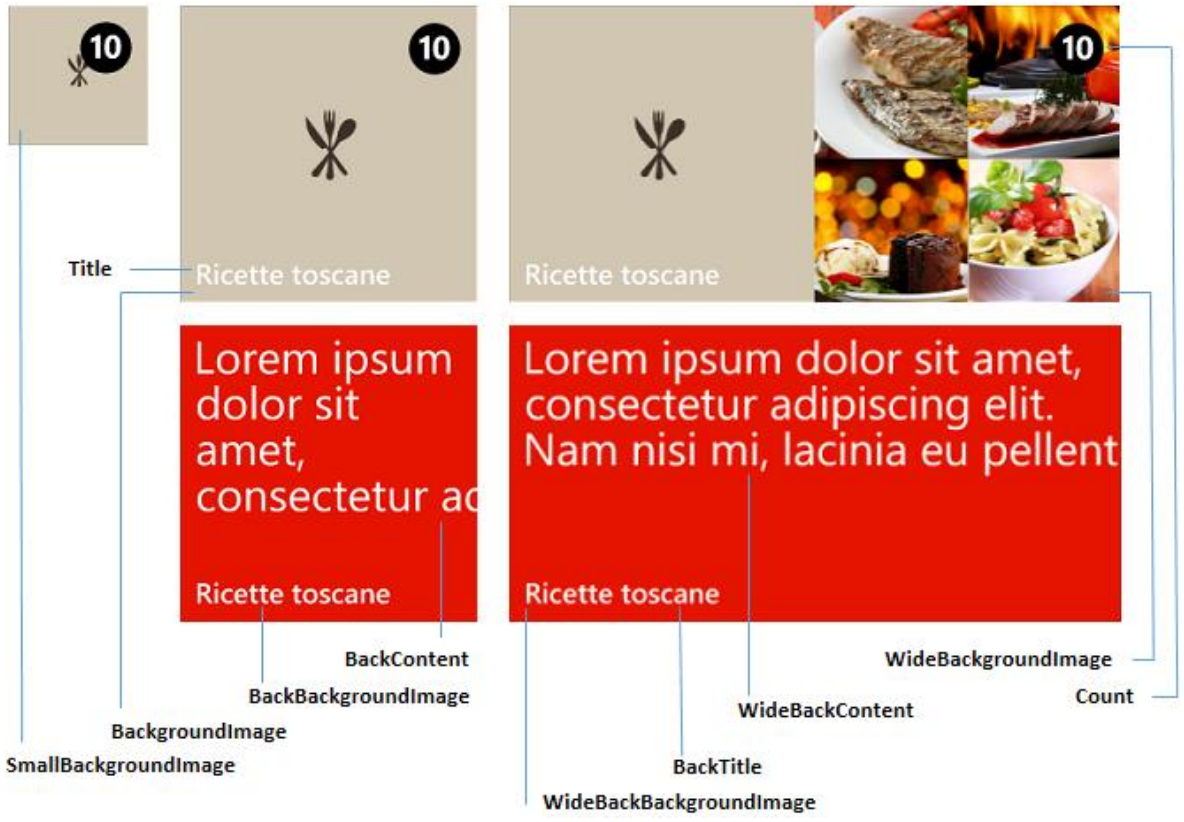

Figure 34: Anatomy of a Tile Using the Flip Template

As you can see from the previous figure, you can customize both front and rear sides of the Tile. If you want to include an image, you have to use one of the following sizes:

- Small: 159 × 159
- Medium: 336 × 336
- Wide: 691 × 336

A flip template Tile is identified by the **`FlipTileData`** class. The following sample shows how to use it to define a Tile that can be managed by code.

```
private void OnCreateFlipTileClicked(object sender, RoutedEventArgs e)
{
    FlipTileData data = new FlipTileData
    {
        SmallBackgroundImage = new
Uri("Assets/Tiles/FlipCycleTileSmall.png", UriKind.Relative),
        BackgroundImage = new Uri("Assets/Tiles/FlipCycleTileMedium.png",
UriKind.Relative),
        WideBackgroundImage = new
Uri("Assets/Tiles/FlipCycleTileLarge.png", UriKind.Relative),
        Title = "Flip tile",
        BackTitle = "Back flip tile",
```

```
        BackContent = "This is a flip tile",
        WideBackContent = "This is a flip tile with wide content",
        Count = 5
    };
}
```

The following code shows how the same Tile is represented using the XML definition needed for remote notifications:

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification" Version="2.0">
  <wp:Tile Id="[Tile ID]" Template="FlipTile">
    <wp:SmallBackgroundImage Action="Clear">[small Tile size
URI]</wp:SmallBackgroundImage>
    <wp:WideBackgroundImage Action="Clear">[front of wide Tile size
URI]</wp:WideBackgroundImage>
    <wp:WideBackBackgroundImage Action="Clear">[back of wide Tile size
URI]</wp:WideBackBackgroundImage>
    <wp:WideBackContent Action="Clear">[back of wide Tile size
content]</wp:WideBackContent>
    <wp:BackgroundImage Action="Clear">[front of medium Tile size
URI]</wp:BackgroundImage>
    <wp:Count Action="Clear">[count]</wp:Count>
    <wp:Title Action="Clear">[title]</wp:Title>
    <wp:BackBackgroundImage Action="Clear">[back of medium Tile size
URI]</wp:BackBackgroundImage>
    <wp:BackTitle Action="Clear">[back of Tile title]</wp:BackTitle>
    <wp:BackContent Action="Clear">[back of medium Tile size
content]</wp:BackContent>
  </wp:Tile>
</wp:Notification>
```

Notice the **`Action`** attribute that is set for many nodes. If you set it without assigning a value to the node, it will simply erase the previous value so that it reverts to the default.

Cycle template

The cycle template can be used to create a visual experience similar to the one offered by the Photos Hub. Up to nine pictures can cycle on the front side of the Tile.

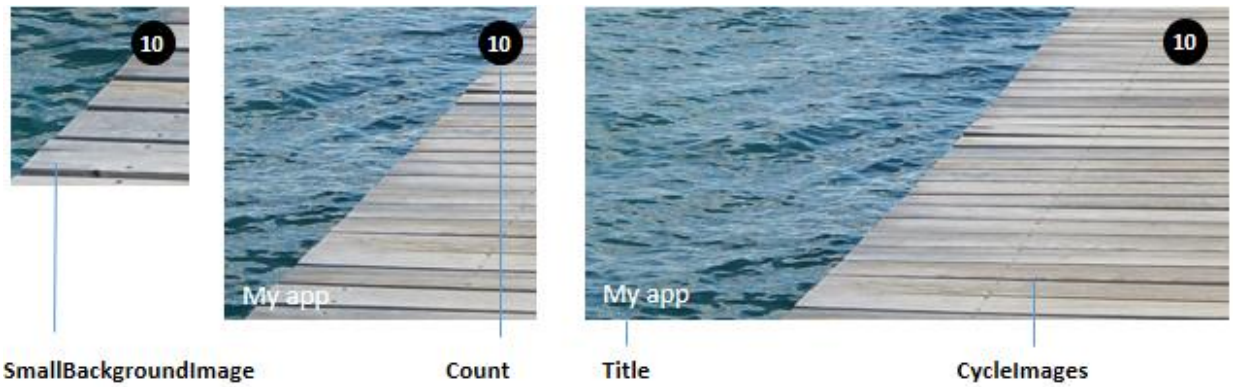

Figure 35: Anatomy of a Tile Using the Cycle Template

The cycle template offers fewer ways to customize the Tile than the other two templates since its focus is the images. The image sizes are the same as those used for the flip template:

- Small: 159 × 159
- Medium: 336 × 336
- Wide: 691 × 336

A cycle template is identified by the **`CycleTileData`** class, as shown in the following sample:

```
private void OnCreateCycleTileClicked(object sender, RoutedEventArgs e)
{
    CycleTileData data = new CycleTileData()
    {
        Count = 5,
        SmallBackgroundImage = new
Uri("Assets/Tiles/FlipCycleTileSmall.png", UriKind.Relative),
        Title = "Cycle tile",
        CycleImages = new List<Uri>
        {
            new Uri("Assets/Tiles/Tile1.png", UriKind.Relative),
            new Uri("Assets/Tiles/Tile2.png", UriKind.Relative),
            new Uri("Assets/Tiles/Tile3.png", UriKind.Relative)
        }
    };
}
```

The following XML can used to send remote notifications to update Tiles based on the cycle template:

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification" Version="2.0">
  <wp:Tile Id="[Tile ID]" Template="CycleTile">
```

```
    <wp:SmallBackgroundImage Action="Clear">[small Tile size
URI]</wp:SmallBackgroundImage>
    <wp:CycleImage1 Action="Clear">[photo 1 URI]</wp:CycleImage1>
    <wp:CycleImage2 Action="Clear">[photo 2 URI]</wp:CycleImage2>
    <wp:CycleImage3 Action="Clear">[photo 3 URI]</wp:CycleImage3>
    <wp:Count Action="Clear">[count]</wp:Count>
    <wp:Title Action="Clear">[title]</wp:Title>
  </wp:Tile>
</wp:Notification>
```

Iconic template

The iconic template is used to create Tiles that emphasize the counter. Many native applications such as Mail, Messaging, and Phone use this template. In this template, the counter is bigger and easier to see.

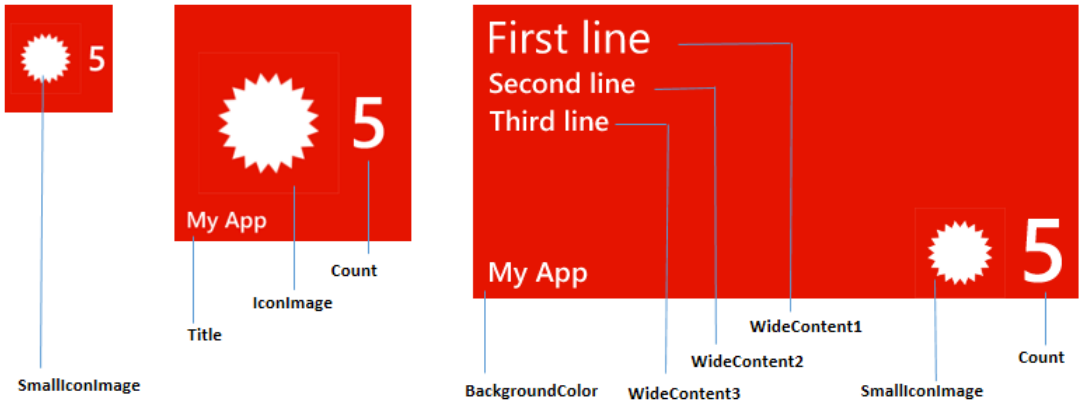

Figure 36: Anatomy of a Tile Using the Cycle Template

The iconic template features two main differences from the flip and cycle templates. The first is that full size images are not supported; instead, you can specify an icon image, which is displayed near the counter. There are just two images sizes required:

- Small and wide Tiles: 110 × 110
- Medium Tile: 202 × 202

The other difference is that it's possible to customize the background color (the only way to do this with the other templates is to use an image with the background color you prefer). If you don't set a background color, the template will automatically use the phone's theme.

An iconic Tile is represented by the **`IconicTileData`** template, as shown in the following sample:

```
private void OnCreateIconicTileClicked(object sender, RoutedEventArgs e)
{
    IconicTileData data = new IconicTileData()
    {
```

```
        SmallIconImage = new Uri("/Assets/Tiles/IconicTileSmall.png",
UriKind.Relative),
        IconImage = new Uri("/Assets/Tiles/IconicTileMediumLarge.png",
UriKind.Relative),
        Title = "My App",
        Count = 5,
        WideContent1 = "First line",
        WideContent2 = "Second line",
        WideContent3 = "Third line"
    };
}
```

The following sample is the XML representation for remote push notifications in a Tile that uses the iconic template:

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification" Version="2.0">
  <wp:Tile Id="[Tile ID]" Template="IconicTile">
    <wp:SmallIconImage Action="Clear">[small Tile size
URI]</wp:SmallIconImage>
    <wp:IconImage Action="Clear">[medium/wide Tile size URI]</wp:IconImage>
    <wp:WideContent1 Action="Clear">[1st row of content]</wp:WideContent1>
    <wp:WideContent2 Action="Clear">[2nd row of content]</wp:WideContent2>
    <wp:WideContent3 Action="Clear">[3rd row of content]</wp:WideContent3>
    <wp:Count Action="Clear">[count]</wp:Count>
    <wp:Title Action="Clear">[title]</wp:Title>
    <wp:BackgroundColor Action="Clear">[hex ARGB format
color]</wp:BackgroundColor>
  </wp:Tile>
</wp:Notification>
```

Working with multiple Tiles

The previous code, in addition to being supported in the application or in a background agent to update the main Tile, can also be used to create multiple Tiles—a feature introduced in Windows Phone 7.5. Secondary Tiles behave like the main ones: they can be updated by notifications and moved or deleted from the Start screen.

The difference is that secondary Tiles have a unique ID, which is the Tile's deep link. The main Tile always opens the application's main page, while secondary Tiles can open another page of the application and include one or more query string parameters to identify the context. For example, a weather application can create Tiles for the user's favorite cities, and every Tile will redirect the user to the forecast page for the selected city.

The base class to interact with Tiles is called **`ShellTile`**, which belongs to the **`Microsoft.Phone.Shell`** namespace.

Creating a secondary Tile is simple: you call the **`Create()`** method by passing the Tile's deep link and the Tile itself, using one of the classes we've seen before. The following sample shows how to create a secondary Tile using the flip template:

```
private void OnCreateFlipTileClicked(object sender, RoutedEventArgs e)
{
    FlipTileData data = new FlipTileData
    {
        SmallBackgroundImage = new
Uri("Assets/Tiles/FlipCycleTileSmall.png", UriKind.Relative),
        BackgroundImage = new Uri("Assets/Tiles/FlipCycleTileMedium.png",
UriKind.Relative),
        WideBackgroundImage = new
Uri("Assets/Tiles/FlipCycleTileLarge.png", UriKind.Relative),
        Title = "Flip tile",
        BackTitle = "Back flip tile",
        BackContent = "This is a flip tile",
        WideBackContent = "This is a flip tile with wide content",
        Count = 5
    };

    ShellTile.Create(new Uri("/MainPage.xaml?id=1", UriKind.Relative),
data, true);
}
```

When the application is opened using this Tile, you'll be able to understand the context and display the proper information using the **`OnNavigatedTo`** method and the **`NavigationContext`** class we used in Chapter 3.

Note: To avoid inappropriate usage of secondary Tiles, every time you create a new Tile the application will be closed to immediately display it to the user.

Deleting a secondary Tile requires working with the **`ShellTile`** class again. It exposes a collection called **`ActiveTiles`**, which contains all the Tiles that belong to the application, including the main one. It's sufficient to get a reference to the Tile we want to delete (using the deep link as an identifier) and call the **`Delete()`** method on it.

```
private void OnDeleteTileClicked(object sender, RoutedEventArgs e)
{
    Uri deepLink = new Uri("/MainPage.xaml?id=1", UriKind.Relative);
    ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(x =>
x.NavigationUri == deepLink);
    if (tile != null)
    {
        tile.Delete();
    }
}
```

The previous sample deletes the Tile identified by the deep link **`/MainPage.xaml?id=1`**. Unlike when the Tile is created, the application won't be closed.

Tip: Remember to always check that a Tile exists before removing it. Like every other Tile, in fact, users can also delete one on the main page by tapping and holding Tile and then tapping the Unpin icon.

Tiles can also be updated. Updates can be performed not only by the main application but also in the background by a background agent.

The approach is similar to the one we've seen for the delete operation. First we have to retrieve a reference to the Tile we want to update, and then we call the **`Update()`** method, passing the Tile object as a parameter. The following sample shows how to update a Tile that uses the flip template:

```
private void OnUpdateMainTileClicked(object sender, RoutedEventArgs e)
{
    FlipTileData data = new FlipTileData
    {
        Title = "Updated Flip tile",
        BackTitle = "Updated Back flip tile",
        BackContent = "This is an updated flip tile",
        WideBackContent = "This is an updated flip tile with wide content",
        Count = 5
    };
    Uri deepLink = new Uri("/MainPage.xaml?id=1", UriKind.Relative);
    ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(x =>
x.NavigationUri == deepLink);
    if (tile != null)
    {
        tile.Update(data);
    }
}
```

The **`Update()`** method can also be used to update the application's main Tile. It's always stored as the first element of the **`ActiveTiles`** collection, so it's enough to call the **`Update()`** method on it as in the following sample:

```
private void OnUpdateMainTileClicked(object sender, RoutedEventArgs e)
{
    FlipTileData data = new FlipTileData
    {
        Title = "Updated Flip tile",
        BackTitle = "Updated Back flip tile",
        BackContent = "This is an updated flip tile",
        WideBackContent = "This is an updated flip tile with wide content",
```

```
        Count = 5
    };
    ShellTile.ActiveTiles.FirstOrDefault().Update(data);
}
```

The previous code will always work, even if the main Tile is not pinned to the Start screen. If the user decides to pin it, the Tile will already be updated with the latest notification.

Tip: You can invite users to pin the main Tile on the Start screen, but you can't force it by code.

Interacting with the lock screen

Windows Phone 8 has introduced a new way for applications to interact with users, thanks to the lock screen support. There are two ways to interact with it:

- Display notifications in the same way the Messaging and Mail apps display the number of unread messages.
- Change the lock screen image; specifically, the application can become a lock screen provider and occasionally update the image using a background agent.

Let's see in detail how to support both scenarios.

Notifications

In the Settings page, users can choose up to five applications that are able to display counter notifications, and only one application that is able to display text notifications.

To support both scenarios in our application, we need to manually add a new declaration in the manifest file (remember to use the **View code** option in the context menu since it's not supported by the visual editor):

```
<Extensions>
  <Extension ExtensionName="LockScreen_Notification_IconCount"
ConsumerID="{111DFF24-AA15-4A96-8006-2BFF8122084F}" TaskID="_default" />
  <Extension ExtensionName="LockScreen_Notification_TextField"
ConsumerID="{111DFF24-AA15-4A96-8006-2BFF8122084F}" TaskID="_default" />
</Extensions>
```

The first extension is used to support counter notifications, while the second one is used for text notifications. You can declare just one of them or both, according to your requirements.

If you want to support counter notifications, there's another modification to apply to the manifest file: inside the **Tokens** section, you'll find the tags that define the main Tile's basic properties. One of them is called **DeviceLockImageURI**, which you need to set with the path of the image that will be used as an icon for the notifications.

```
<DeviceLockImageURI IsRelative="true"
IsResource="false">Assets/WinLogo.png</DeviceLockImageURI>
```

The image should have the following properties:

- The supported resolution is 38 × 38.
- It has to be in PNG format.
- It can contain only transparent or white pixels. No other colors are supported.

Once your application is set, you don't have to do anything special to display lock screen notifications. In fact, they are based on Tile notifications, so you'll be able to update both the Tile and the lock screen with just one notification.

- If your application supports counter notifications, you need to send a Tile notification with the number stored in the **Count** property.
- If your application supports text notifications, you need to send a Tile notification with the text stored in the **WideBackContent** property for the flip template or the **WideContent1** property for an iconic template. The cycle template doesn't support text notifications.

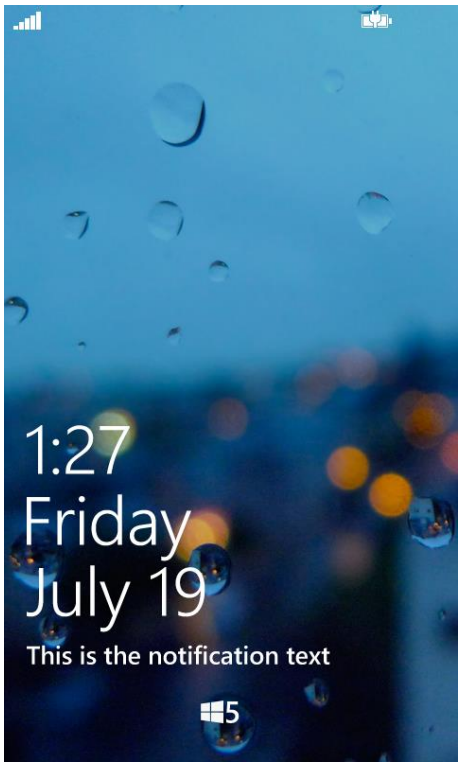

Figure 37: Lock Screen Notification with Counter and Text

Lock screen image

The starting point for supporting lock screen images is, again, the manifest file. The following sample is the declaration that should be added in the **Extensions** section:

```
<Extensions>
  <Extension ExtensionName="LockScreen_Background" ConsumerID="{111DFF24-
AA15-4A96-8006-2BFF8122084F}" TaskID="_default" />
</Extensions>
```

Starting now, your application will be listed as a wallpaper provider in the Settings page. If the user chooses your application as a provider, you'll be able to update the lock screen image, both from the foreground app and using a background agent.

The APIs allow you to check whether the application has already been set as a provider, or you can ask the user. If the application is set as a provider, you will be able to effectively change the wallpaper; otherwise you'll get an exception.

Two classes are part of the **Windows.Phone.System.UserProfile** namespace: **LockScreenManager** can be used to detect the current provider status, and **LockScreen** can effectively perform operations on the lock screen.

```
private async void OnSetWallpaperClicked(object sender, RoutedEventArgs e)
{
    Uri wallpaper = new Uri("ms-appx:///Assets/Wallpapers/Wallpaper1.jpg",
UriKind.RelativeOrAbsolute);

    bool isProvider = LockScreenManager.IsProvidedByCurrentApplication;
    if (isProvider)
    {
        LockScreen.SetImageUri(wallpaper);
    }
    else
    {
        LockScreenRequestResult lockScreenRequestResult = await
LockScreenManager.RequestAccessAsync();
        if (lockScreenRequestResult == LockScreenRequestResult.Granted)
        {
            LockScreen.SetImageUri(wallpaper);
        }
    }
}
```

The first step is to check if the current application is set as a provider by using the **`IsProvidedByCurrentApplication`** property of the **`LockScreenManager`** class. Otherwise, we ask for the user's permission by calling the **`RequestAccessAsync()`** method. In return, we receive the user's choice, which can be positive (**`LockScreenRequestResult.Granted`**) or negative (**`LockScreenRequestResult.Denied`**).

In both cases, only if the application has been set as provider can we effectively change the lock screen image using the **`SetImageUri()`** method, which requires the picture's path as a parameter. The picture can be either part of the project (as in the previous sample where we use the **`ms-appx:///`** prefix) or stored in the local storage (in this case, we have to use the **`ms-appdata:///Local/`** prefix). Remote images are not directly supported; they must be downloaded before using them as a lock screen background.

The previous code can also be used for the background agent. The difference is that you'll be able to check whether the application is set as a provider and, eventually, change the image. You won't be able to ask to the user for permission to use your app as a provider since background agents can't interact with the UI.

A quick recap

In this chapter, we have seen that live apps are one of the core concepts in Windows Phone development and, to properly create a quality experience, many factors are involved, like notifications, agents, and Tiles.

The following list details what we've learned:

- Push notifications are the best way to notify users of something even if the application is in the background. An app can either send toast notifications or update Tiles. We've seen how to create the required architecture for reach, both on the client side and the server side.
- Push notifications offer the best approach for optimizing battery life and performance, but they support limited scenarios and require a server application to send them. For this reason, Windows Phone has introduced background agents, which we can periodically execute to send notifications or carry out general purpose operations, even when the application is not in the foreground.
- Windows Phone offers a special background agent type called audio background agent that is used in audio playback scenarios. Applications are able to play audio even when they are not running, like the native Music + Videos Hub.
- Alarms and reminders are a simple way to show reminders to users when the application is not running.
- Live Tiles are one of the distinctive features of the platform. We've learned how to customize them by choosing between different templates and sizes.
- We've seen another new feature introduced in Windows Phone 8, lock screen support: applications are now able to interact with the lock screen by displaying notifications and changing the wallpaper.

Chapter 10 Distributing the Application: Localization, the Windows Phone Store, and In-App Purchases

Trial apps

One of the distinctive features of Windows Phone applications is that they support a trial mode, which enables them to be downloaded from the Windows Phone Store as a trial with limited features. Once users decide to buy the full application, they don't have to download it from scratch; the Store will simply download an updated certificate, which will unlock all the previously blocked features.

From the developer point of view, managing a trial is simple. The **`Windows.ApplicationModel.Store`** namespace contains a class called **`LicenseInformation`**, which offers a property called **`IsTrial`**. If its value is **`true`**, it means that the application has been installed in trial mode, so we have to lock the features we don't want enabled; otherwise, the user has purchased it, so all features can be made available.

It's up to you to choose the best trial experience for your application. For example, you can disable some features, limit the number of times the applications can be executed, or in a game, choose which levels to unlock.

```
LicenseInformation info = new LicenseInformation();
if (info.IsTrial)
{
    MessageBox.Show("Application is running in trial mode");
}
else
{
    MessageBox.Show("Application is running in full mode!");
}
```

Localization

One of the best ways to increase the sales and number of downloads of an application is to support multiple languages. The default Windows Phone 8 template in Visual Studio already supports localization by including a folder called **Resources**. It contains all resource files, which are simple XML files with a special **.resx** extension.

Resource files are simply a collection of values (the translated text) associated with a specific key which is used in the application to refer to that text. According to the user's language, the application will automatically use the proper resource file.

The standard template creates a file called **AppResources.resx** inside the **Resources** folder, which refers to the standard language (usually English).

Supporting a new language is easy. Just right-click your project in the **Solution Explorer** and choose **Properties**. The supported cultures box will display all the available languages. When you add new languages by selecting them in the list, Visual Studio will automatically create a new **AppResources** file named **AppResources.xx-yy.resx** in the Resources folder, where **xx-yy** is the culture code of the selected language (for example, if you've added Italian, it will create a file named **AppResources.it-IT.resx**).

Visual Studio offers a useful visual editor for working with resource files. It displays all the values inside a table, where you can easily define a key, its value, and an optional comment. To access it, simply double-click on a resource file in the **Resources** folder.

In addition to offering a standard resource file, the Windows Phone template also includes a class called **`LocalizedStrings`**, which acts as a wrapper between the localization file and the XAML. You can find it defined as a global resource in the **App.xaml** file:

```
<Application.Resources>
    <local:LocalizedStrings xmlns:local="clr-namespace:Localizzazione"
x:Key="LocalizedStrings"/>
</Application.Resources>
```

Thanks to this wrapper, you'll be able to access resources directly in XAML. You don't have to add it directly in the XAML every time you need to display text in the user interface; instead you'll add in the resource files and then connect them to your XAML with the following syntax:

```
<TextBlock Text="{Binding Source={StaticResource LocalizedStrings},
Path=LocalizedResources.SampleText}" />
```

Thanks to the **`LocalizedStrings`** class, we can access every value in the resource file simply by using the **`LocalizedResources.MyKey`** syntax, where **`MyKey`** is the key that identifies the text that you want to display.

If you want to access the resource string directly from code, you'll have to use the **`AppResources`** singleton class, as shown in the following sample:

```
private void OnShowMessageClicked(object sender, RoutedEventArgs e)
{
    MessageBox.Show(AppResources.SampleText);
}
```

The Multilingual App Toolkit

Microsoft has created a useful Visual Studio extension called Multilingual App Toolkit, which can be downloaded here. The tool doesn't change how localization works; it will always be based on resource files, which are accessed by the application using the **`LocalizedString`** class.

The following list details some of the main benefits of the Multilingual App Toolkit:

- One of the most time-consuming aspects of working with localization is manually copying all the new values you have added in the basic language to all other resource files. The Multilingual App Toolkit will do this for you. During the build process, it will copy all the new keys added to the **AppResources.resx** file to all the other resource files.
- It offers a better visual interface for managing translations. You'll be able to immediately identify the new keys to translate and set a different translation status.
- It supports Bing services to automatically translate sentences. Obviously, you can't completely rely on automatic translations, but they can be a good start for your localization process.
- It's able to automatically generate pseudo language, which is a way to immediately identify untranslated resources and have a better idea of how much space text occupies in the UI.

After you've installed the toolkit, you'll have to enable it for your project in the **Tools** menu of Visual Studio by selecting the **Enable multilingual app toolkit** option. After enabling it, Visual Studio will work with **.xlf** files instead of **.resx** files (except for the main language). During the compilation process, the toolkit will generate all the required **.resx** files for you.

To access the toolkit visual interface, just double-click an **.xlf** file and you'll see the complete list of resources. An icon will notify you of the status of each resource. If the resource has not yet been translated, a red circle is displayed. If the resource has been translated but requires revision, a yellow circle is displayed. If the translation has been approved, a green circle is displayed.

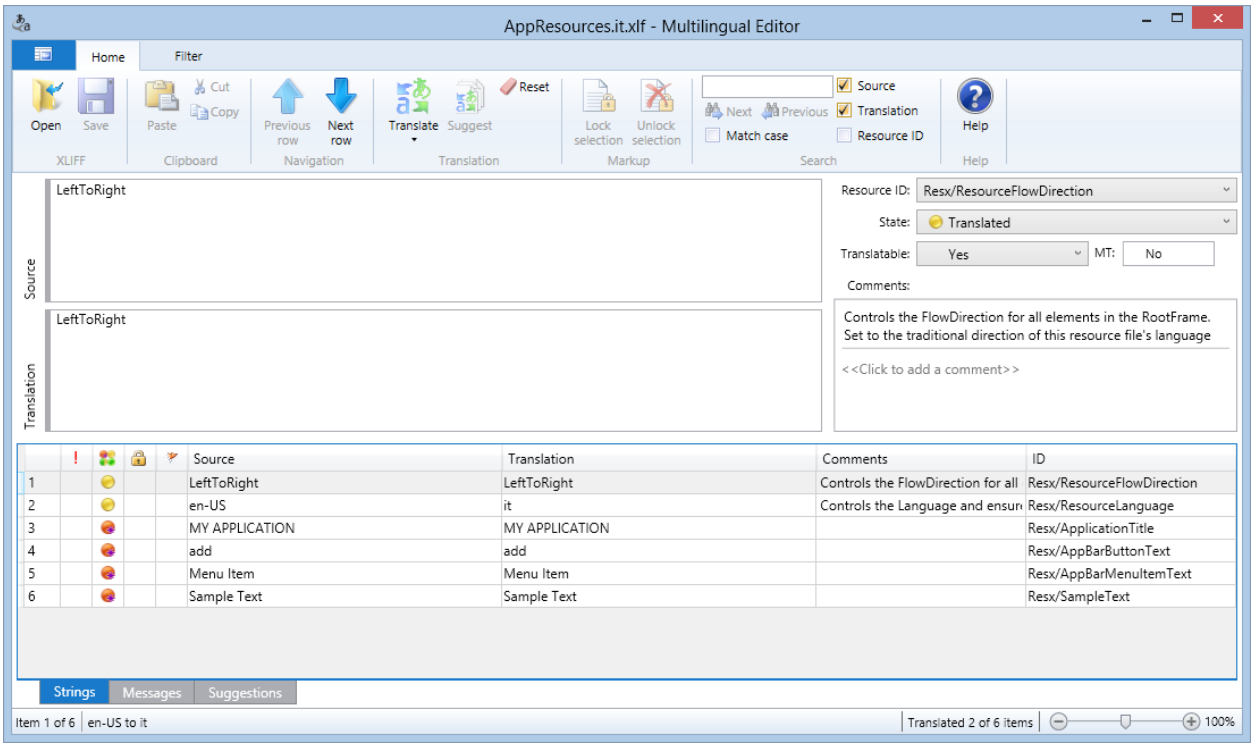

Figure 38: Multilingual App Toolkit

Forcing a language

Windows Phone will automatically pick the resource file that matches the phone's language. If an appropriate resource file is missing (because the phone's language is not supported by your application), the default will be used.

You can change this behavior, for example, if you want to give users the option to choose the language they prefer, regardless of the phone's language. To do this, you'll need to set a different culture code when the **App** class (declared in the **App.xaml.cs** file) is created:

```
public App()
{
    CultureInfo culture = new CultureInfo("it-IT");
    Thread.CurrentThread.CurrentCulture = culture;
    Thread.CurrentThread.CurrentUICulture = culture;

}
```

After you've defined a **CultureInfo** object by passing the culture code as a parameter, you can assign it to two different properties of the **Thread.CurrentThread** object:

- **CurrentCulture** is the application's culture, which is used to define features like date and time formats, currency, etc.
- **CurrentUICulture** is the user interface culture, which is used to understand which resource file to pick.

This approach is also required if you want to use the pseudo language generated by the Multilingual App Toolkit during the testing process. Since pseudo language is not an official language supported by the platform, you'll have to force it using the **qps-ploc** culture code, as shown in the following sample:

```
public App()
{
    CultureInfo culture = new CultureInfo("qps-ploc");
    Thread.CurrentThread.CurrentCulture = culture;
    Thread.CurrentThread.CurrentUICulture = culture;

}
```

Using pseudo language is a great way to identify text that has yet to be translated and test whether your application's layout is able to manage long text. What typically happens when developing an application is that you test it with only a couple languages, forgetting that a word that looks very short in English, for example, can be very long when it's translated to German.

The Store experience

As mentioned in the first chapter, the Windows Phone Store is the only way to distribute applications to your users unless you're pursuing enterprise distribution. You'll need a paid developer account, which can be purchased from https://dev.windowsphone.com/en-us/join. The fee is $19 per year, unless you're a student subscribed to the DreamSpark program, in which case access is free. A similar benefit is granted to MSDN subscribers: in the benefits page, you can get a token that allows you to register for free.

Other than the annual fee, Microsoft applies a revenue-sharing approach: 30 percent of the application's price is kept by Microsoft, while the other 70 percent is given to the developer. Of course, this sharing doesn't apply if the app is free.

Once your developer account has been activated and your application is ready, you can submit it to the portal at dev.windowsphone.com. During the process, you'll have to define the application marketing features, like price, metadata, and distribution type. After that, the submission is completed, and the certification process starts. Applications are not automatically published to the Store; first they need to pass a certification process that validates the application both from a technical and content point of view.

The technical tests make sure that the application offers a good experience to users: it doesn't crash, it's fast and responsive, and the user interface is not confusing.

The manual tests check the content of the application. Some types of content like pornography, excessive violence, etc., are not allowed and will lead to a certification failure.

When you start the submission process, you'll see a list of steps to follow to complete the process. Let's briefly look at each step.

Step 1: App info

The first step of the publishing process is used to set some basic information, like the app's category, price tier, and market distribution. Your application will automatically be distributed in all the supported countries at the price you've chosen. The price will be automatically converted into each country's currency. In this step, you can choose to automatically exclude countries like China, where the content policies are stricter. The certification process also offers an optional **Market selection and custom pricing** step, which offers deeper customization: you can choose to distribute the applications only in specific countries and customize the price for each country.

The other important option to set during this step is the distribution channel. There are three ways to distribute a Windows Phone application:

- **The public store**: The app can be discovered and downloaded by any user.
- **Hidden**: The app is still available in the public store, but it can't be discovered by users. The only way to find it is by using the direct link that is generated when the app is published.
- **Beta**: The application can't be discovered by users, and moreover, only authorized users will be allowed to download it. Users are authorized with the Microsoft account they registered the phone with. You'll be able to include up to 10,000 users during the submission process. This distribution channel was created to support beta testing; in this scenario, however, the application won't actually be tested, but will be available to the selected users within two hours of submitting the app. A beta application automatically expires 90 days after it's been submitted for the first time, regardless of later updates.

Step 2: Upload and describe your XAP

The second step requires more time to complete, since you'll have to provide all the application information that will be displayed in the Windows Phone Store.

The first step is to upload the XAP file. The XAP is the package produced by Visual Studio when you compile your project, and it contains all the required files for the application to run. You'll find it inside the **bin** folder of your Visual Studio project. Remember to always compile the application in release mode; otherwise it won't be accepted.

Once you've uploaded the XAP, the portal will automatically display a recap of the application's features, like the supported resolutions, the required capabilities, and so on. You'll also have to set the version number of the application you're uploading.

The portal will also automatically detect the languages you support, which will be displayed in a drop-down menu called **Language details**. You'll have to set the application's metadata (title, description, and keywords) for every supported language. This information will be displayed in the Store when the user opens your application's page.

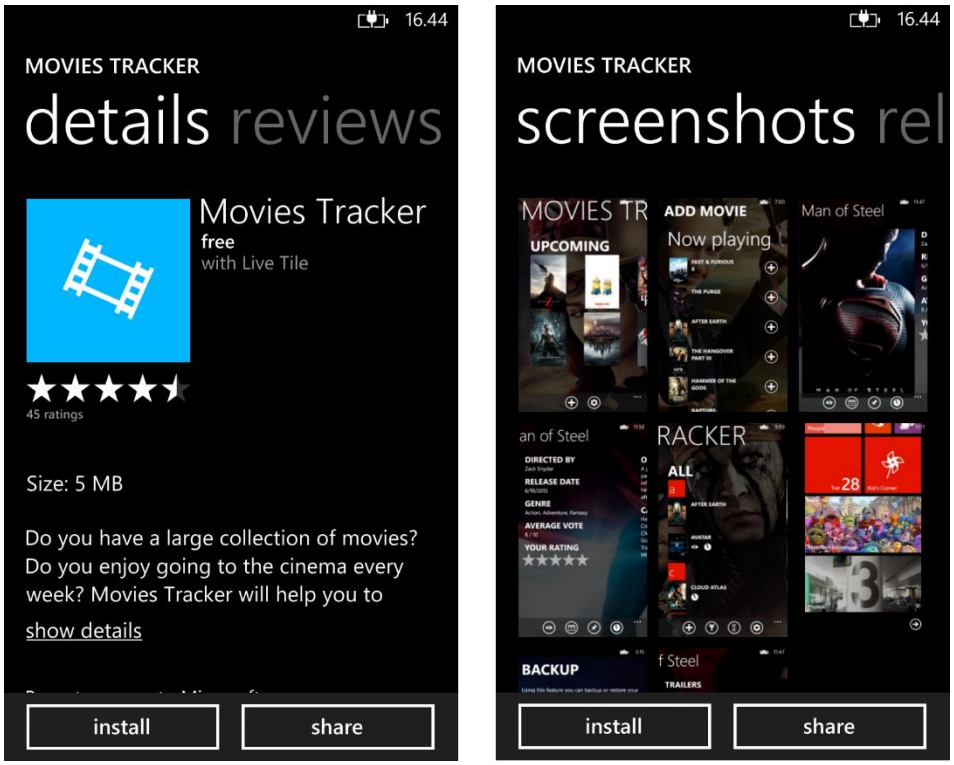

Figure 39: Information about the App Displayed in the Store

You will also need to upload at least one screenshot (eight are allowed) for every language and supported resolution, plus the application icon. They will be displayed in the screenshots section of the Store. To take screenshots, you can use one of the tools available in the emulator.

Once you've completed all the required steps, the portal will show you a recap of the submission for your confirmation.

Managing the application's life cycle

Once your application has been certified, you'll have access to many reports that will help you understand how well your application is performing. There are download reports, sales reports, and crash reports.

If the application doesn't pass the certification process, you'll find in your dashboard a PDF file containing the full report. It will tell you in detail what went wrong and what you should do to fix the identified problems.

In the end, of course, you can also update your application. In this case you'll simply have to repeat all the submission steps, although all the fields will already be filled with the old information and metadata. The process is also the same if you just want to change information like the price, description, and screenshots. The submission only has to be certified if you've changed any information that needs to be validated. If you only change the price, the update is immediately published.

In-app purchases

A different way to make money with Windows Phone apps is to support in-app purchases. In addition to buying the app from the Store, users can also make purchases within the application itself.

In-app purchases have always been allowed, but Windows Phone 8 has introduced specific APIs and Microsoft backend support. Previously, the developer was in charge of creating the server infrastructure, managing payments, and connecting the client.

Now, you can simply rely on the Store infrastructure. Users will pay using the same credit card they used to purchase apps or music from the Store, and the portal will allow you to create one or more products to buy within the application. Also in this case, the revenue-sharing approach will be applied. If you want to avoid it, you're free to implement your own in-app purchase infrastructure; Microsoft doesn't force developers to use its services.

It's important to highlight that in-app purchasing through Microsoft services is allowed only for virtual products (like new features, game levels, etc.); it can't be used to buy physical goods.

Two kinds of products are supported by Windows Phone:

- **Durables** are products that can be purchased just once, like application features, level packages, etc.
- **Consumables** are products that can be purchased again after they've been consumed, like virtual coins.

Defining a product

Products are defined in the portal. In the application's page, the **Products** section offers an option to add new in-app products. Defining a product is similar to submitting an application: you have to set some basic properties like the name, price, and metadata, and then you submit it.

There are two key properties:

- The product identifier, which is a unique ID that you use in your application to refer to the product.
- The product type, which can be consumable or durable.

Interacting with products

Once you've defined all the properties, you can start working with the products in your application. Probably, the first requirement is to display the list of available products users can buy. This goal is achieved using the **CurrentApp** class that belongs to the **Windows.ApplicationModel.Store** namespace.

```
private async void OnListStuffClicked(object sender, RoutedEventArgs e)
{
    ListingInformation listing = await
CurrentApp.LoadListingInformationAsync();
    List<ProductListing> productListings =
listing.ProductListings.Values.ToList();

    Purchases.ItemsSource = productListings;
}
```

The **CurrentApp** class exposes the **LoadListingInformationAsync()** method, which returns a **ListingInformation** object that stores all the information about the available products.

Products are stored inside the **ProductListings** collection. In the previous sample, we display them to the user using a **LongListSelector** control, which has the following definition:

```
<phone:LongListSelector x:Name="Purchases"
SelectionChanged="OnSelectedPurchaseChanged">
    <phone:LongListSelector.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" Margin="15, 0, 0, 20">
                <TextBlock Text="{Binding Path=Name}" Margin="0, 0, 20, 0"
/>
                <TextBlock Text="{Binding Path=FormattedPrice}" />
            </StackPanel>
        </DataTemplate>
    </phone:LongListSelector.ItemTemplate>
</phone:LongListSelector>
```

Every **ProductListing** object contains the same property we've assigned to the product in the Store. In the previous sample, we show the name (**Name**) and price (**FormattedPrice**) of the product.

Once you have the product list, you have to manage the purchase process. Again, we need to use the **CurrentApp** class, which offers the **RequestProductPurchaseAsync()** method. As a parameter, we're going to pass the **ProductListing** object selected by the user.

```
private async void OnSelectedPurchaseChanged(object sender,
SelectionChangedEventArgs e)
{
```

```
    ProductListing selectedPurchase = Purchases.SelectedItem as
ProductListing;
    await
CurrentApp.RequestProductPurchaseAsync(selectedPurchase.ProductId, true);
}
```

Once a product is purchased, you can check its status by using the **CurrentApp.LicenseInformation.ProductLicenses** collection. It contains all the supported products with the related license status. Every product is identified by a key, which is the unique identifier we assigned when we created it in the portal.

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    if
(CurrentApp.LicenseInformation.ProductLicenses.ContainsKey("CoolProduct"))
    {
        ProductLicense license =
CurrentApp.LicenseInformation.ProductLicenses["CoolProduct"];
        if (license.IsActive)
        {
            //Unlock the feature.
        }
        else
        {
            //Lock the feature.
        }
    }
}
```

In the previous sample, we can determine if the product with the **CoolProduct** identifier has been purchased by checking the value of the **IsActive** property. The operation is performed when the page is loaded: if the product has been purchased, we unlock the related feature, otherwise we'll wait for the user to purchase it.

For a consumable product, the purchase process is the same. The only difference is that, after it's been consumed, we need to report it so that it can be "unlocked" to be purchased again.

We can report it by calling the **ReportProductFullfillment()** method of the **CurrentApp** class, which requires as a parameter the **ProductLicense** object that identifies the product.

```
private void OnConsumeButtonClicked(object sender, RoutedEventArgs e)
{
    var licenses = CurrentApp.LicenseInformation.ProductLicenses;
    if (licenses.ContainsKey("CoolProductConsumable"))
    {
        ProductLicense productLicense = licenses["CoolProductConsumable"];
```

```
            CurrentApp.ReportProductFulfillment(productLicense.ProductId);
        }
    }
```

Testing in-app purchases

Unfortunately, testing an in-app purchase is not very easy. Since products have to be defined in the portal, you'll have to submit your application before being able to test the purchase process.

One work-around is to publish the application as beta; since the app doesn't need to be certified, it will be immediately available for testing. The downside is that it's hard to properly test it if something goes wrong since you can't debug it using Visual Studio as you normally would with any other application.

For this reason, Microsoft has provided a testing library called **MockIAP**. Its purpose is to "mock" the real in-app purchase APIs so that the operations are not executed against the real Microsoft service, but use fake products that are defined within the application.

MockIAP can be downloaded from MSDN and added to your solution. The APIs it offers are the same as those exposed by the native SDK; the only difference is that they belong to the **`MockIAPLib`** namespace instead of the **`Windows.ApplicationModel.Store`** namespace.

There are two things to do to start using the MockIAP library. The first is to add some conditional compilations directives so that when the application is compiled in debug mode (typically during testing) it will use the mock library. When it's compiled in release mode, it will use the real Microsoft services.

The following code sample shows how the namespace declaration will look in your page:

```
#if DEBUG
    using MockIAPLib;
    using Store = MockIAPLib;
#else
    using Windows.ApplicationModel.Store;
#endif
```

The second step is to define the products we need to use. Since the application won't be connected to the real services, we need to duplicate in the code the products we've defined in the portal.

The following code shows a sample initialization:

```
private void SetupMockIAP()
{
    MockIAP.Init();
```

```
    MockIAP.RunInMockMode(true);
    MockIAP.SetListingInformation(1, "en-US", "This is a sample app", "1",
"SampleApp");
    ProductListing p = new ProductListing
    {
        Name = "Cool product",
        ProductId = "CoolProduct",
        ProductType = Windows.ApplicationModel.Store.ProductType.Durable,
        Description = "A cool product",
        FormattedPrice = "10.00 €",
        Tag = string.Empty
    };

    MockIAP.AddProductListing("CoolProduct", p);

    ProductListing p2 = new ProductListing
    {
        Name = "Cool product consumable",
        ProductId = "CoolProductConsumable",
        ProductType =
Windows.ApplicationModel.Store.ProductType.Consumable,
        Description = "A cool consumable product",
        FormattedPrice = "5.00 €",
        Tag = string.Empty
    };

    MockIAP.AddProductListing("CoolProductConsumable", p2);
}
```

We create two products: a durable one identified by the key **`CoolProduct`**, and a consumable one identified by the key **`CoolProductConsumable`**. Every product is identified by the **`ProductListing`** class (the same class we used with the real services), and we can use it to define all the product properties that are usually retrieved from Microsoft services like the name, type, price, and so on.

We add each product using the **`AddProductListing()`** method of the **`MockIAP`** class. After adding the products, we can use the standard APIs for in-app purchases. Operations will be performed locally with the fake products instead of the real services, but the behavior will be exactly the same. We'll be able to list, buy, and consume the available products.

A quick recap

When we talk about mobile applications, development isn't the only aspect we need to consider. If we want to be successful, we also have to distribute and promote our application. In this chapter, we've discussed:

- How to localize the application so that it appeals to a wider audience.
- How the Windows Phone Store and the certification process work. Submitting the app is a crucial step since we have to make many important marketing decisions like the app's price, the countries it will available in, etc.
- How to increase revenue by supporting in-app purchases.