More Windows 8.1 Succinctly

by Matteo Pagani

Syncfusion | Technology Resource Portal

More Windows 8.1 Succinctly

By
Matteo Pagani

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Rajen Kishna
Copy Editor: Courtney Wright, Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, 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.

Introduction

The *Succinctly* series aims to provide valuable information in short, standalone volumes. Occasionally, a topic will prove so expansive that a single volume is not enough to adequately address the most important points. In *Windows 8.1 Succinctly*, you were guided through the process of developing apps for both Windows 8.1 and Windows Phone 8.1. Now that you know how to build functioning apps, it's time to get them into the hands of users.

In *More Windows 8.1 Succinctly*, you will learn how to publish apps, work with the network, and much more. By the end of the book, you will be ready to send your app into the world for users to enjoy. If you have not yet read *Windows 8.1 Succinctly*, we strongly recommend you do so to ensure you are ready for these advanced topics. If you have read the previous e-book and are ready to push your skills even further, proceed to Chapter 1.

Chapter 1 Publishing an Application on the Store

In this chapter, in addition to learning how to submit an application to the Store, we'll see also how to improve our application so that it can be more interesting and inspiring to the user.

Localization

An effective strategy to make our application more popular is to properly support multiple languages. Windows and Windows Phone devices are used all over the world and by all kinds of users, so we can't take for granted that our users are able to read and speak English. Windows Store apps offer built-in support for localization. The difference from the traditional approach is that we won't hard-code our texts in the XAML or in code—we'll save them in separate files, one for every supported language. Every file (which, under the hood, are XML files) contains a list of resources, with a key (the unique identifier) and the value (the real text localized in the specific language). Every time we'd like to display a text to the user, we're going to add a reference to the key that matches the text that we want to display.

The first step to manage localization is to add a file for each language we want to support. This scenario is implemented using a naming convention based on the folder's name. All the resource files need to be included in a folder in the project (it could also be in the Shared project, if you're developing a Universal app), usually called **strings**. Inside this folder, you'll need to create a subfolder with the culture code for each language you want to support.

You can find a list of supported codes in the MSDN documentation: http://msdn.microsoft.com/en-us/library/windows/apps/hh202918(v=vs.105).aspx. It's important to highlight that you're not forced to use the full culture code (like en-US or it-IT); you can also use the short version (like **en** or **it**) if you want to support all the variants of the same culture with one unique resource file.

Inside each folder, you'll need to create a resource file; you'll find a specific template in Visual Studio, called **Resources File (.resw).** The default name assigned is **Resources.resw**.

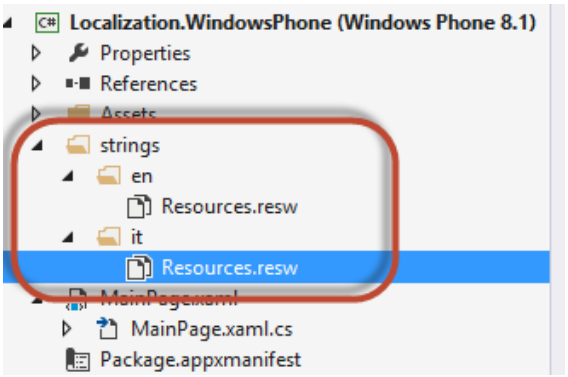


Figure 1: A project that supports two languages: English and Italian

If you double-click on the **Resources.resw** file, Visual Studio will open the Visual Editor, which will display a list of rows and three columns:

- **Name**, which is the resource identifier key.
- **Value,** which is the localized text.
- **Comment,** which is an optional comment that makes it easier to understand where the resource is used.

Resources are strictly connected to the XAML controls that will display the value; consequently, the Name follows a specific convention. The first part of the key is the unique resource identifier; it's up to you to choose what's best for your scenario. For example, it could be a label like **`Title`**. The second part of the key, separated by the first part with a period, is the name of the control's property we want to handle with this resource. For example, if we want to display the value of the resource in the **`Text`** property of a **`TextBlock`** control, we should define it as **`Title.Text`**.

How can we connect the resource to its control? By using a special XAML property called **`x:Uid`**, which is supported by every XAML control. We need to set it with the unique identifier of the resource, which is the first part of the key (before the period). For example, if we want to connect the resource **`Title.Text`** to display its value in a **`TextBlock`** control, we need to define it in the following way:

```
<TextBlock x:Uid="Title" />
```

Another common requirement is to use a resource in code. For example, if we need to display a pop-up message using the **`MessageDialog`** class, we need to access the resource in another way, since a pop-up message can't be defined in XAML. To achieve this goal, we need to use the **`ResourceLoader`** class, which offers a method called **`GetString()`**. As the parameter, we need to pass the full name that identifies the resource. The application will automatically retrieve the resource based on the current language.

```
private async void OnShowMessageClicked(object sender, RoutedEventArgs e)
{
    ResourceLoader loader = new ResourceLoader();
    string resource = loader.GetString("Title/Text");
    MessageDialog dialog = new MessageDialog(resource);
    await dialog.ShowAsync();
}
```

As you can see, there's an important difference to highlight: when you call a resource from code, you need to use the forward slash (*/*) instead of the period as a separator between the first and the second part of the key. In the sample, to retrieve the value of the resource with the name **`Title.Text`**, we pass as the parameter the value **`Title/Text`**.

Localizing images

Another common requirement is to support different images according to the user's language. In this case, we can use a naming convention, similar to the one we saw in Chapter 4, to manage the different scale factors. We can add a **.lang-** suffix to the image name, followed by the culture code, to make it visible only when the device is used with the specified language. For example, if we have an image called **logo.png** and we want to have two versions, one for English and one for Italian, we would have to add two images to the project: one with name **logo.lang-en.png**, and another with name **logo.lang-it.png**.

This naming convention is completely transparent to the developer. In XAML or in code, we'll just need to reference the image using the base name (**logo.png**), and the operating system will automatically pick the proper one, according to the language. Here's an example:

```
<Image Source="/Assets/logo.png" />
```

Manage the default language

The default application language is set in the manifest file, by a field called **Default language** in the **Application** section. It's the default language that is used when we don't provide a specific resource file for the current language. By default, it's **en-US**, and it's a good approach to keep the setting this way; since English is one of the most widespread languages in the world, it's likely that our users will know it, even if we don't support their native language.

However, there are some scenarios where it makes sense to use another default language. For example, local applications that are distributed only in a specific country (like a newsreader app connected to a local magazine). In this case, we can change the **Default language** value with another culture code.

Unlike in Windows Phone 8.0, it's not required anymore to specify, in the manifest file, all the languages supported by the application; the list of available languages will be automatically detected by the subfolders that have been created in the **strings** folder.

Translating the application's name

We can also translate the application's name, in case we want our app to have a localized name, based on the user's language. To achieve this goal, we can add a new string in each resource file. This time, we can give the resource the name we prefer, without following any convention.

Then, we can open the manifest file and, in the **Application** section, change the **Display name** field by setting the resource name with the prefix **`ms-resource:`**. For example, if we've created a resource with the key **`ApplicationName`**, we should use as value for the field **`ms-resource:ApplicationName`**.

The trial mode

Another way to make our paid application more interesting to the user is to provide a trial mode; this way, the user will be able to try it before purchasing it. The most interesting advantage of the trial mode is that, unlike in other platforms, you won't have to publish two different applications. The trial will be implemented in the full application and you'll be able to detect, in code, in which mode it's running. If the user purchases your app, they won't have to download it again from scratch; the Store will just download a new certificate that will unlock all the features.

There are two ways to manage a trial:

- **Manually:** We have an API that returns a **`Boolean`** that simply tells us if the application is running in trial mode. With this information, we can implement the trial in the way we prefer; we could block some features, display a pop-up message every time the app is launched, display some advertising, etc.
- **Time trial:** When you submit your application in the Store, you'll be able to set an expiration date for the trial, and the app will be automatically blocked once the trial period is ended. This mode is available only on Windows, since the Windows Phone Dev Center doesn't allow you to set an expiration date during the submission.

Both ways are managed by a class called **`LicenseInformation`**, which is a property of the **`CurrentApp`** class that belongs to the **`Windows.ApplicationModel.Store`** namespace. There are two properties that we can use to manage the trial:

- **`IsTrial`** is a **`Boolean`** value that returns if the application is running in trial mode or not.
- **`IsActive`** is a **`Boolean`** value that returns if the application is expired or not. If we want to discover the expiration date, we can access to the **`ExpirationDate`** property.

The following sample code shows a pop-up message with the current status of the application:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    LicenseInformation license = CurrentApp.LicenseInformation;

    if (!license.IsActive)
    {
        MessageDialog dialog = new MessageDialog("The application is
expired");
        await dialog.ShowAsync();
    }
    else if (license.IsTrial)
    {
        MessageDialog dialog = new MessageDialog("The application is in trial
mode");
        await dialog.ShowAsync();
    }
}
```

Testing the trial mode

The **CurrentApp** class is connected to the information that is reported by the Store; consequently, the only way we would have to test the trial mode would be to submit the app to the Store. However, the Windows Runtime offers a simulator class that we can use to test the trial mode: it's called **CurrentAppSimulator**, and it offers the same properties and methods of the **CurrentApp** class. The difference is that, instead of retrieving the information from the Store, it will load them from a local configuration file.

Since the **CurrentAppSimulator** class is similar to the **CurrentApp** class, using it is very simple. Just replace, in your code, the **CurrentApp** references with the **CurrentAppSimulator** references, like in the following sample.

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    LicenseInformation license = CurrentAppSimulator.LicenseInformation;

    if (!license.IsActive)
    {
        MessageDialog dialog = new MessageDialog("The application is
expired");
        await dialog.ShowAsync();
    }
    else if (license.IsTrial)
    {
        MessageDialog dialog = new MessageDialog("The application is in trial
mode");
        await dialog.ShowAsync();
    }
}
```

The first time you use this class, a new file called **WindowStoreProxy.xml** will be created in the local storage, inside the folder **Microsoft/Windows Store/ApiData**. You can read Chapter 5 to understand how to get access to this file in the local storage both on Windows and Windows Phone.

This XML file contains a set of parameters to configure the app status. Here is a sample configuration file:

```
<?xml version="1.0" encoding="utf-16" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>00000000-0000-0000-0000-000000000000</AppId>
      <LinkUri>
        http://apps.microsoft.com/webpdp/app/00000000-0000-0000-0000-
        000000000000
      </LinkUri>
```

```
      <CurrentMarket>en-US</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="en-us">
        <Name>AppName</Name>
        <Description>AppDescription</Description>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </App>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
      <ExpirationDate>2015-01-19T05:00:00.00Z</ExpirationDate>
    </App>
  </LicenseInformation>
</CurrentApp>
```

The section we need to use to test the trial mode is inside the **`LicenseInformation`** block, which contains an **`App`** section with a set of elements that matches the properties we've previously seen as exposed by the **`CurrentApp`** class (which are **`IsActive`**, **`IsTrial`** and **`ExpirationDate`**). We can edit this file and save it; from now on, the **`CurrentAppSimulator`** class will use these properties to return the application's status. The previous sample defined an application with a trial mode that expires on January 19th 2015 at 5 AM.

Tip: Before publishing the application, remember to replace the CurrentAppSimulator references with the CurrentApp class; otherwise, your code won't work. Alternatively, you can use the conditional compilation to use the CurrentAppSimulator class in Debug mode and the CurrentApp class in Release mode.

Note: If you try to use the previous sample code that sets the ExpirationDate on Windows Phone, you will notice that the code will work fine. The APIs are indeed supported by the platform, but since the submission process doesn't support this scenario, you won't be able to use it in a published application.

Purchasing the application

The application can simply be purchased from the Store. When the user opens the application's page and they have already downloaded the trial, they will see a Buy button that will start the purchase procedure. You can also trigger the purchase process also from the application; this way, the user will never have to leave your application to purchase it. This goal is achieved by calling the **`RequestAppPurchaseAsync()`** method of the **`CurrentApp`** class:

```
private async void OnBuyAppClicked(object sender, RoutedEventArgs e)
{
    await CurrentApp.RequestAppPurchaseAsync(true);
}
```

If you decide to use this approach, it's important to track when the trial status changes. If we're managing the trial mode manually, we need to disable it as soon as the purchasing process is completed. To support this requirement, the **`LicenseInformation`** class offers an event called **`LicenseChanged`**, which is triggered when a trial application has been purchased and turned into a full version. The following sample code uses this event to simply display a message to the user; in a real application, you would unlock the features that were previously unlocked.

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    LicenseInformation license = CurrentApp.LicenseInformation;
    license.LicenseChanged += license_LicenseChanged;
}

private async void license_LicenseChanged()
{
    MessageDialog dialog = new MessageDialog("The application has been
purchased");
    await dialog.ShowAsync();
}
```

You can simulate this feature by using the **`CurrentAppSimulator`** class. You can, in fact, customize the **`App`** section of the **WindowsStoreProxy.xml** file to set up the Store information, including the price, like in the following sample.

```
<App>
  <AppId>00000000-0000-0000-0000-000000000000</AppId>
  <LinkUri>
    http://apps.microsoft.com/webpdp/app/00000000-0000-0000-0000-
    000000000000
  </LinkUri>
  <CurrentMarket>it-IT</CurrentMarket>
  <AgeRating>3</AgeRating>
  <MarketData xml:lang="en-us">
    <Name>Sample app</Name>
```

```
    <Description>Description</Description>
    <Price>2.00</Price>
    <CurrencySymbol>€</CurrencySymbol>
    <CurrencyCode>EUR</CurrencyCode>
  </MarketData>
</App>
```

The previous sample simulates an application priced at 2 euro. When we use the **`RequestAppPurchaseAsync()`** method of the **`CurrentAppSimulator`** class, Windows and Windows Phone will display a special screen to simulate the purchase process. We will be able to choose one of the supported statuses, which are **S_OK** (purchase completed with success), **E_INVALID**, **E_FAIL** or **E_OUTOFMEMORY** (different error conditions).

In-app purchase

Another way to monetize our application is to support in-app purchases: instead of making a paid application, we can make a free one (or a paid one, but at a cheaper price) that gives the user the option to purchase new features or items within the application itself. For example, a photo-editing application can give access to a basic set of filters, and then allow the user to buy more of them using in-app purchase.

Microsoft supports the in-app purchase feature directly with the Store, so you won't need to manage payments yourself. The user will make in-app purchases by using the same credit card connected to their account, and the experience will be very similar to the one offered when they buy an app. For the developer, the revenue-sharing mechanism is the same: Microsoft will keep 30 percent of the price, and the developer gets the rest. However, you are not forced to use Microsoft services to manage in-app purchases: if you already have your own payment system, you can freely use it without sharing the income with Microsoft.

When you use the Microsoft services, you can define two types of purchases:

- **Durable:** Once the product is purchased, it becomes user's property, and it will be maintained even if he uninstalls the application or changes his device. It's typically used when the purchase is used to unlock an application feature (a new set of levels for a game, the advertising removal, etc.).
- **Consumable:** These products can be consumed and purchased multiple times. They are typically used in games to manage virtual money or temporary features (like a power up).

The available products are defined in the Windows Dev Center during the submission process, in the **Services** step. In the Windows Phone Dev Center, instead, there's a specific section called **Products**, where you can add new products by clicking on the **Add in-app product** button.

One important feature to highlight about in-app purchases is that they are strictly connected to the Universal App concept. If you're going to publish a Universal project and enable identity sharing (so you'll have the same application on both the Windows and the Windows Phone stores), the in-app purchases will be shared among the two platforms. This way, if the user has purchased a product on her smartphone, she will also find it available on her tablet or computer. However, since the stores are still separate, you will have to define the products on both Dev Centers.

Every product is identified by a set of parameters:

- A name
- A unique identifier, which is used in the application to reference it
- The type (durable or consumable)
- The expiration date: If we set it, the product will be no longer purchasable after that date.
- The language
- The price
- The countries where it's available.
- A title and description for every supported language.

Once you've defined the products, we're going to use the same **`CurrentApp`** class we've previously seen to manage the trial experience.

Retrieving the list of available products

The **`CurrentApp`** class offers a method called **`LoadListingInformationAsync()`**, which gives access to some information about the application. One of them is **`ProductListing`**, which is a collection of all the available in-app purchases products. Every element is identified by the **`ProductListing`** class, which contains all the information about the product that we've defined on the portal, like the price (**`FormattedPrice`**) or name (**`Name`**). The following sample code shows how to retrieve the list and display it to the user using a **`ListView`** control:

```
private async void OnListProductsClicked(object sender, RoutedEventArgs e)
{
    ListingInformation information = await
CurrentApp.LoadListingInformationAsync();
    List<ProductListing> products =
information.ProductListings.Values.ToList();
    Products.ItemsSource = products;
}
```

The next sample, instead, shows how to define the **`ListView`** control to display to the user the name and the price of each product:

```
<ListView x:Name="Products" SelectionChanged="Products_OnSelectionChanged">
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
```

```
                <TextBlock Text="{Binding Path=Name}" Margin="12, 0, 12, 0"
/>
                <TextBlock Text="{Binding Path=FormattedPrice}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Purchasing a product

The product purchase is made by using another method of the **CurrentApp** class, **RequestProductPurchaseAsync()**. As the parameter, we need to pass the unique product identifier we've defined in the portal. The following sample shows how to manage the **ItemClick** event of the previous **ListView** control that is triggered when the user has tapped on one of the products:

```
private async void Products_OnItemClicked(object sender,
SelectionChangedEventArgs e)
{
    Guid productTransactionId;
    ProductListing selectedProduct = e.ClickedItem as ProductListing;
    PurchaseResults result = await
CurrentApp.RequestProductPurchaseAsync(selectedProduct.ProductId);
    switch (result.Status)
    {
        case ProductPurchaseStatus.Succeeded:
        {
            productTransactionId = result.TransactionId;
            MessageDialog dialog = new MessageDialog("The product has been
succesfully purchased");
            await dialog.ShowAsync();
            break;
        }
        case ProductPurchaseStatus.AlreadyPurchased:
        {
            MessageDialog dialog = new MessageDialog("The product has already
been purchased");
            await dialog.ShowAsync();
            break;
        }
        case ProductPurchaseStatus.NotFulfilled:
        {
            MessageDialog dialog = new MessageDialog("The product hasn't been
fulfilled yet and it's still valid");
            await dialog.ShowAsync();
            break;
```

```
        }
    }
}
```

The method returns a **PurchaseResults** object, which offers a property called **Status**. It's a **ProductPurchaseStatus** enumerator, with a value for each different scenario. In the previous sample, we manage three situations:

- **Succeeded**: The product has been purchased with success.
- **AlreadyPurchased**: The user is trying to buy a durable product they already purchased.
- **NotFullfilled**: The user is trying to buy a consumable product before that the application has marked it as fulfilled.

It's up to us to manage these responses in the proper way; in the previous sample, we just display a message to the user. In a real application, when we receive the **Succeeded** response, we would download or unlock the purchased feature.

Managing a consumable product

In the previous sample code, we saved in a variable the value of a property of the **PurchaseResults** class called **TransactionId**. It's the unique identifier of the transaction, and it's really important to manage consumable products. Once the product has been fulfilled, we need to notify the Store of this operation, so that we can unlock it and allow the user to buy it again. This operation is performed using the **ReportConsumableFullfilmentAsync()** method of the **CurrentApp** class, which requires, in addition to the product identifier, the **TransactionId** we've previously saved.

```
private async void OnUseConsumableClicked(object sender, RoutedEventArgs e)
{
    await CurrentApp.ReportConsumableFulfillmentAsync("ProductId",
productTransactionId);
}
```

After using this code, if we tried to re-purchase the consumable product, the operation would be successful, and we would get a **Succeeded** status.

Checking the product's status

When we use in-app purchases, typically our application has a set of features that are locked. Consequently, when the app starts, we need to check the product's status, so that we can unlock the features that the user has purchased. We can get this information from the **ProductLicenses** collection, which is a property of the **CurrentApp** class.

This collection includes an item for each product; just see if the **IsActive** property is set to **true** to understand if it has been purchased or not. The following code checks, when the page is loaded, if the product identified by the name **MyProduct** has been purchased:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    LicenseInformation license = CurrentApp.LicenseInformation;
    ProductLicense product = license.ProductLicenses["MyProduct"];
    if (product.IsActive)
    {
        //unlock the feature
    }
}
```

Testing in-app purchase

To test in-app purchases, we can use the same approach we've seen to test the trial mode. The products to buy will be defined in the WindowsStoreProxy.xml file stored in the local storage of the application. Instead of the **CurrentApp** class, we'll need to use the **CurrentAppSimulator** class. The following configuration file shows how to simulate two products:

```
<?xml version="1.0" encoding="utf-16" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>00000000-0000-0000-0000-000000000000</AppId>
      <LinkUri>
        http://apps.microsoft.com/webpdp/app/00000000-0000-0000-0000-
000000000000
      </LinkUri>
      <CurrentMarket>en-US</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="en-us">
        <Name>AppName</Name>
        <Description>AppDescription</Description>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </App>
    <Product ProductId="RemoveAdvertising" LicenseDuration="0"
ProductType="Durable">
      <MarketData xml:lang="en-us">
        <Name>Remove advertising</Name>
        <Price>1.00</Price>
        <CurrencySymbol>€</CurrencySymbol>
        <CurrencyCode>EUR</CurrencyCode>
      </MarketData>
    </Product>
    <Product ProductId="PowerUp" LicenseDuration="0"
```

```
ProductType="Consumable">
      <MarketData xml:lang="en-us">
        <Name>Power up</Name>
        <Price>1.00</Price>
        <CurrencySymbol>€</CurrencySymbol>
        <CurrencyCode>EUR</CurrencyCode>
      </MarketData>
    </Product>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>false</IsTrial>
    </App>
  </LicenseInformation>
</CurrentApp>
```

After the **App** section, we find a series of **Product** items; each of them identifies a product that can be purchased in the application. The most important information is set using attributes, like **ProductId** (the product identifier), **LicenseDuration** (we can set it to 0 if it doesn't expire) and **ProductType**, which can be **Durable** or **Consumable**.

Inside every **Product** section, we have a tag called **MarketData**, which contains all the information about the product purchase, and will be displayed to the user when they try to buy the app. When you trigger a purchase using the simulator, the experience will be similar to the trial one: a pop-up window will allow you to simulate one of the supported statuses of the purchase process.

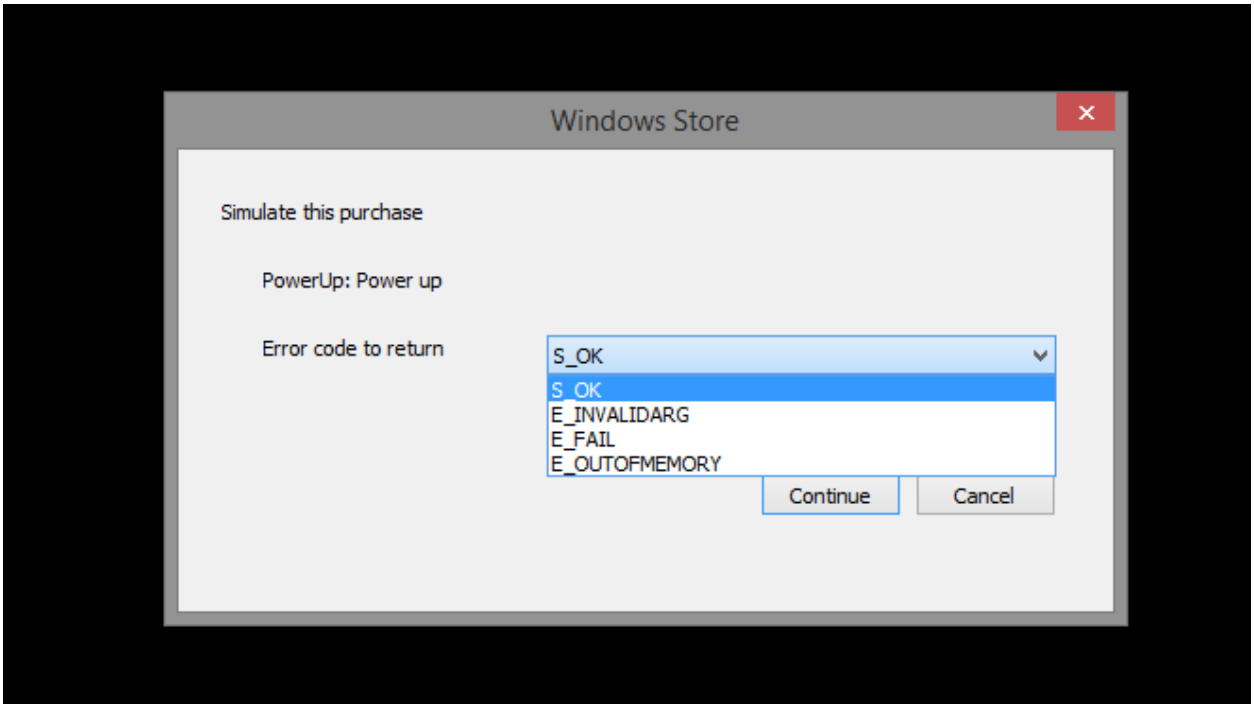


Figure 2: Simulating an in-app purchase in a Windows application

You can also simulate if a product has been already purchased or not, by adding a new **Product** item in the **LicenseInformation** section of the XML file, like in the following sample:

```
<LicenseInformation>
  <App>
    <IsActive>true</IsActive>
    <IsTrial>true</IsTrial>
    <ExpirationDate>2016-01-19T05:00:00.00Z</ExpirationDate>
  </App>
  <Product ProductId="RemoveAdvertising">
    <IsActive>true</IsActive>
  </Product>
</LicenseInformation>
```

By using this configuration, the product identified by the **RemoveAdvertising** id will be already active. If we want to test, instead, a consumable product, we need to add a new section called **ConsumableInformation** under the **LicenseInformation** section, like in the following sample:

```
<ConsumableInformation>
  <Product ProductId="PowerUp" TransactionId="00000000-0000-0000-0000-
000000000000" Status="Active" />
</ConsumableInformation>
```

Publishing the application on the Store

We are reaching the final stage of our project: we are ready to publish it to the Store and share it with all the Windows and Windows Phone users. As already mentioned in the beginning of the book, Universal Windows apps are not a kind of application, but a special Visual Studio template that has been created to allow developers to share as much code as possible between the two platforms. However, in the end we're going to get two different packages, since there are two different Stores: one for Windows, and one for Windows Phone. Consequently, in this last part of the book, we'll see the publishing process in separate sections, according to the type of application you're submitting.

Sharing the identity between two applications

Even if we need to publish two different applications on two different Stores, we'll still have the chance to connect them. This feature is called identity sharing, and it has many advantages:

- **Purchase sharing**: If the application has been purchased on one of the two platforms, the user won't have to pay again to download it on the other one.

- **In-app purchases sharing**: In-app purchases are shared, so that a product bought on one platform is available for free on the other one.
- **Roaming storage synchronization**: In Chapter 5, we learned how to use the roaming storage, where content is automatically synchronized across different devices. If the applications are sharing the same identity, this feature is extended to different platforms.
- **Using a single push notification channel**: When two applications are sharing the same identity, we can use the same channel to send a push notification to both platforms, by using the mechanism we learned in Chapter 10.

All these features are managed with the Microsoft account, which should be configured the same as the primary account for all the user's devices, whether they are smartphones, tablets, or computers. To enable identity sharing, you'll have to reserve the same name for both applications. When you start to submit a new Windows or Windows Phone application, the first step will ask you to choose a name for the application, which should be unique on the Store (this requirement was only recently introduced for Windows Phone, so you may find apps with the same name).

In the first submission step, you can reserve a new name, or choose, from a dropdown menu, one that you've already reserved for another application. When you choose the name of an application that has been already published on another platform, the portal will show you a warning that the applications will be linked and the identity shared. It's important to make sure that we're connecting the right apps before pressing the **Associate app** button, since it's not possible to unlink two linked apps.

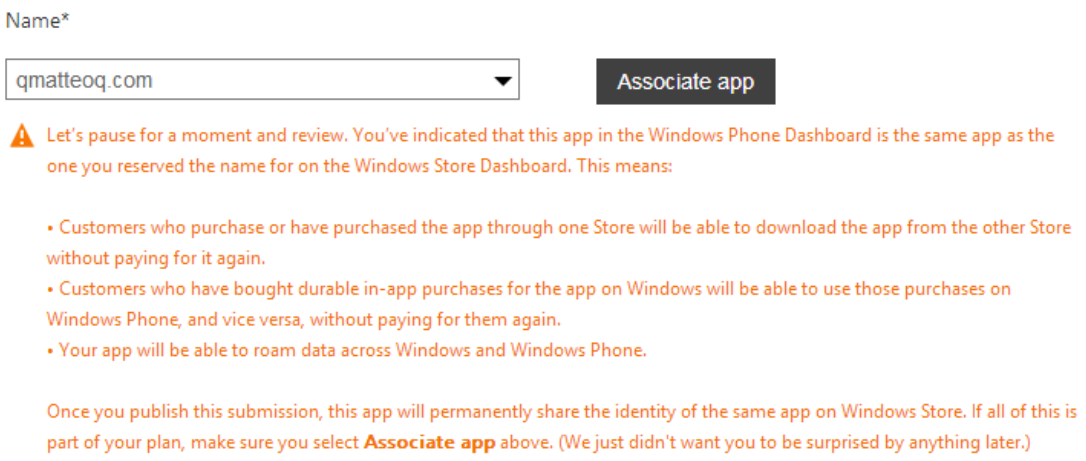
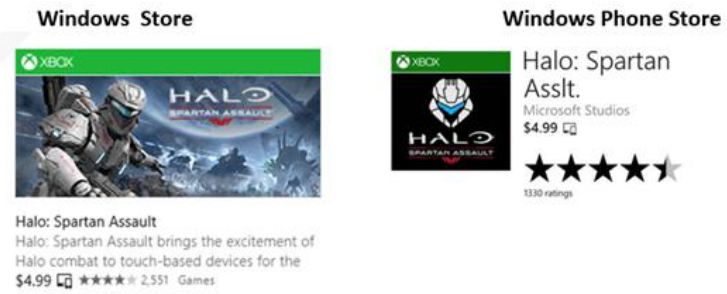


Figure 3: The warning message that is displayed when you try to link two applications

When two apps are published as universal and sharing an identity, they will be displayed on the Store with a special icon near the price, as you can see in the following image:



How to create a developer account

To publish applications, you need to create a developer account, which is shared between both Stores. A single account can be used to publish apps both for Windows and Windows Phone. The account's price is $19 for individual developers and $99 for companies. It's a lifetime subscription, which has to be paid only once. In the past, the Store required you to pay the fee every year to maintain your apps, but now this requirement has been removed.

There are two exceptions:

- Students can register for a program called DreamSpark (http://www.dreamspark.com), which gives free access to all the Microsoft development tools, including the full Visual Studio versions. Among the benefits, you get access to a code that can be used to register as a Windows Developer for free.
- Professional developers who have a MSDN subscription can get a similar benefit (depending on their subscription): a token that can be used to create a developer account for free.

You can start the procedure to create a developer account at http://s.qmatteoq.com/DevRegistration. You will be asked for a set of personal information, together with the Publisher name, which is the name that will be displayed on the Store. You can pay the subscription fee using a credit card or PayPal, and the account will be immediately activated at the end of the procedure. If you're planning to create a company account, however, the procedure requires more time: once the registration is complete, a Microsoft partner will contact you to validate your identity. This procedure will verify that the company really exists and that you are authorized to operate on behalf of it.

Once the account is ready, there are two important sections to complete if you want to publish paid apps: Payout and Tax.

- The Payout section is required to set up how we want to be paid by Microsoft. We can set up a bank account or a PayPal account. Microsoft will pay us every time we reach the income amount of $200.
- The second section is required to properly manage the taxes payout, to avoid being taxed by both the U.S. and your country. If you're unsure about this section, I suggest you contact a business consultant for help.

The Dashboard

Both portals offer a main section called Dashboard, which gives you quick access to all the Dev Center sections. You can start a new submission, see the details about the already published apps, and get statistics about download numbers and the number of sold apps. The main access to the Dev Center is found at http://dev.windows.com. After you've logged in clicked on the Dashboard link, you will be asked which Dev Center you want to access.

The certification process

The Windows and Windows Phone stores are controlled ecosystems; before the application is made available to the user, it needs to pass a certification process, which will check that it follows a set of guidelines and requirements. An application is evaluated using the following parameters:

- **Technical**: The application needs to be stable and fast; it needs to run correctly on all the devices, regardless of the available memory and the CPU speed; and it needs to properly support different resolutions and form factors.
- **Contents**: There are some contents that are not allowed in a Windows Store app, like pornographic images, excessive violence, racism, religious discrimination, etc.
- **Guidelines**: In this book, we've learned that when we develop an application, there are some guidelines that we need to follow.

Certification times are not fixed: in most cases, only automatic tests are performed, which means that the application will be certified in a few hours. However, it can happen also that the application is picked up for a deeper testing; in this case, it can take up to five business days. If the certification process is successfully completed, the application will be published to the Store (unless you've opted for a manual publishing, as we'll see later). In case of failure, you will receive a detailed report with all the problems that have been found, and the exact steps to reproduce them.

How to generate the package to publish

Visual Studio offers a way to generate the package that you're going to publish to the Store. The option is called **Create App Packages**, and it's available in the **Store** menu, which is displayed when you right-click in Solution Explorer on the Windows or Windows Phone project, or in the top menu bar in Visual Studio. This option will trigger a wizard procedure, which will ask you if you want to create a package to publish on the Store. If you choose No, you will create a package that can only be manually deployed to a phone or to another PC or tablet.

If you choose the first option, the next step will require you to log in with the Microsoft Account that is connected to your developer account. After that, the wizard will show you the list of all the names you've already reserved (or you can reserve a new one).

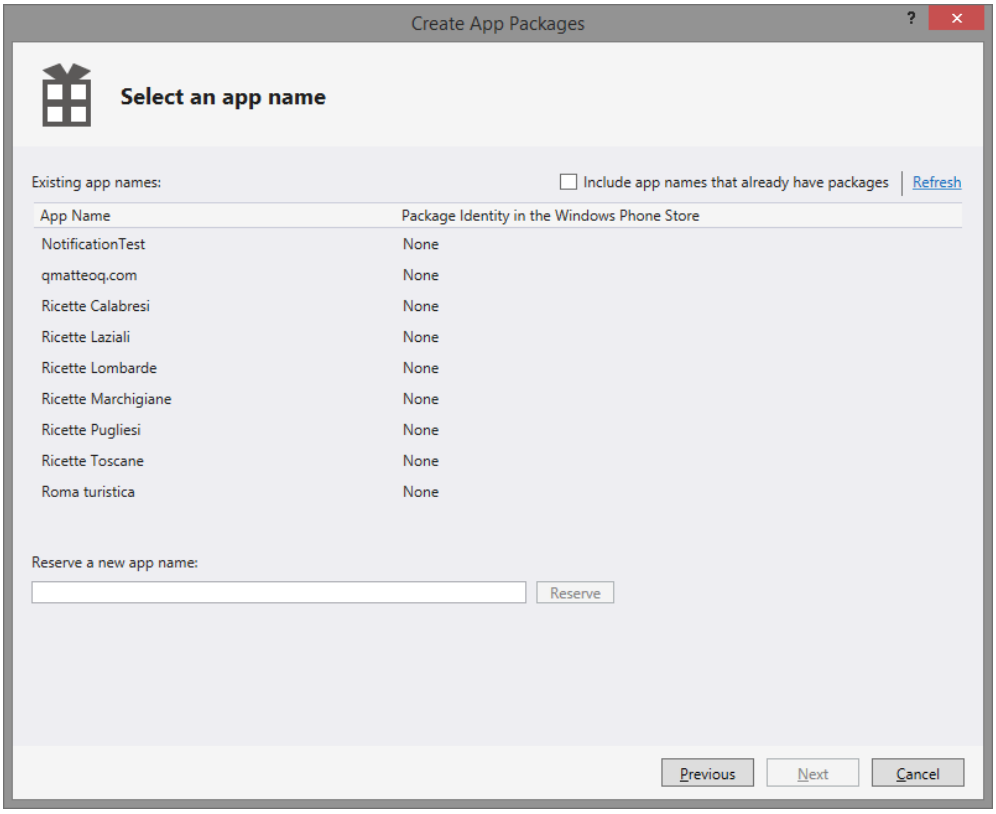


Figure 4: The option to link a package to a specific application on the Dev Center

The next step will ask you for information about the package, such as:

- The path where you want to save it.
- The version number of the application (which always needs to be higher than the version currently published on the Store, if we're creating an update).
- If you want to generate an app bundle: When this feature is turned on, the downloaded package will contain just the base application. All the other resources (images, localization files, etc.) will be downloaded separately, according to the configuration of the device. For example, if you're installing the application on an English phone with 720p resolution, the Store will download only the English resources and the 720p images.
- The supported configurations: If it's a Windows Phone app, you'll only be able to generate a neutral or an ARM package, since there aren't smartphones with X86 processors on the market. However, in most cases it's enough to create a Neutral package: only in some scenarios will you required to create a specific package for each architecture (like when you use a native library in your project, like SQLite).

Now the procedure is complete. After you click the **Create** button, the wizard will build the package, by providing an .appxupload or .appxbundle file to upload during the submission process. After the generation is complete, we have the chance to run the Windows App Certification Kit, which will launch a series of basic tests that will help us make sure that the certification process won't fail (at least from a technical point of view).

Publishing a Windows Phone app

The submission process of a Windows Phone app is made up of five steps, two of which are required. Let's see the various steps in detail.

Step 1: App info

This section defines the base information about the application, like the price, name, and category. The first information to define is the name, as we already highlighted when we talked about publishing a Universal Windows app and sharing the identity.

The other required information is:

- The application's category and, optionally, a sub category.
- The price.
- Whether our application supports a trial mode.
- The countries where the application will be released. By default, the **Distribute to all available markets at the base price tier** option is enabled, which will make the application available in every market supported by the platform. We can also choose to automatically exclude China, since it has more strict rules when it comes to content (for example, you can't publish an application that uses Twitter or Facebook, since they are not allowed in China).

The distribution channels for a Windows Phone app

In the first step, you'll have the chance to define which distribution channel to use to distribute the app. Usually, this section is hidden and the public distribution is automatically selected. There are three ways to distribute a Windows Phone app:

- **Public Store**: It's the standard approach; the application is publicly available on the Store and can be found and installed by any user.
- **Hide from users browsing or searching the Store**: The application is still published on the Store, but the user won't be able to search for it or it won't appear in the Store rankings. Users will be able to install the application only by using the direct Store link.
- **Beta**: This channel is used when we want to get feedback for our application, and we want to distribute it to a selected numbers of testers. As with the Hide option, the application won't be visible in the Store; you'll be able to find it only with the direct link. In addition, not all the users will be able to download it. We will have to provide a list of allowed people by specifying the Microsoft Account registered with their devices. The step offers a specific area for this purpose, where we can enter all the accounts separated by a semi-colon. If an unauthorized user will tries to install the application, the Store will display an error message. An important feature of the beta submission is that it doesn't require certification; it only needs to pass the technical tests. Since we're publishing the app for testing purposes, it makes sense that it's not fully completed yet, and thus, it can have some bugs.

It's important to highlight that the previous distribution channels are available only on Windows Phone apps. At the time of writing, the beta distribution is not available for Windows apps.

The last option we can set, using the Publish section, is when we want to publish the app. We can publish immediately after the certification process has been passed, or we can manually publish it (for example, because we want to start a marketing campaign first).

Step 2: Upload and describe your package

This section will ask the developer to publish the application package we've previously generated, and to fill all the information that is displayed to the user on the Store, so that it can better understand the purpose of our application.

At the beginning, this step will be empty. First, we have to upload the .appxupload or .appxbundle package that we've generated with Visual Studio, by clicking the **Add new** button. After the upload is complete, the Store will start to analyze it and will immediately display a set of information retrieved from the package, like the version number, the languages and resolutions supported, the declared capabilities, etc.

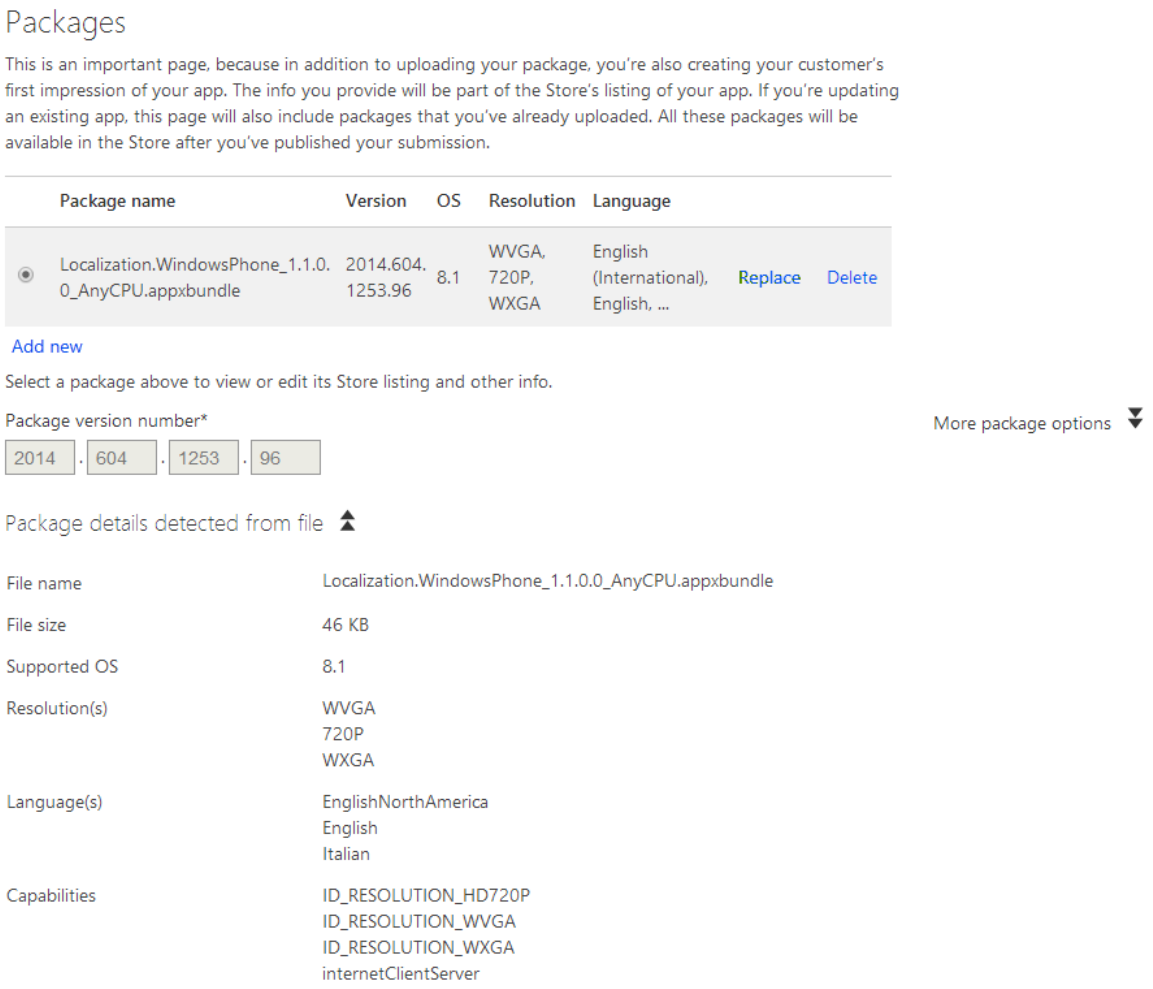


Figure 5: The package features detected by the Store

The second part of the section will ask for all the metadata that describe the application, which needs to be specified for every supported language. You'll find a dropdown menu that you can use to choose the language you're going to define.

The requested information is:

- **Description**: The text displayed to the user that describes our application. The text can have a maximum length of 2,000 chars.
- **Update description**: If we're submitting an update, we can describe what's changed in this new version. Also in this case, the maximum length is 2,000 chars.
- **Keywords**: We can include up to five keywords, which are used on the Store to make it easier for to users to search for your application.

The last section collects all the images that are displayed on the Store:

- **App tile icon** is the application's logo. The resolution is 300x300.
- **Promotional images**: If the application is chosen to be featured and promoted on the Store, Microsoft will pick one of these images. It requires three different images: 1000x800 (used when the image is placed as Store background), 358x358, and 158x173 (used when the app is highlighted in the first page of the Store).
- **Screenshots**: These are real images captured from your application, which show the features and the visual layout. You can upload up to eight screenshots; you can upload a set of screenshots for each supported resolution, or you can upload just the WXGA screenshots (with resolution 768x1280) and the Store will resize them for the other resolutions.

Optional steps

The submission procedure also offers three optional steps:

- **Add in-app advertising:** This step is used if you want to display advertising in your app using PubCenter, which is Microsoft's advertising provider. This step will give you the credentials required to properly register the advertising control in your pages.
- **Market selection and custom pricing:** We've seen how, in Step 1, we were able to choose the price and the countries where we will sell our application. The same price is applied in every country and automatically converted based on the local currency. With this step, we can choose to distribute the application only in some countries, or to sell it in some of them at a different price than the base one we've chosen.
- **Map services:** We've already talked about this step in Chapter 7. It's required if we're using the map control in our application, since it provides the required credentials to register it.

Submitting the application

Once we've filled all the steps, we can press the **Review and submit** button to be redirected to a summary page that will display all the main features of the application we're going to publish. If we don't find any issues, we can press the **Submit** button to confirm and start the certification process. You will receive an email with the certification response as soon as the process is complete.

Updating the application

If you want to release an update for your application, the procedure is the same as we've described before. The difference is that, instead of starting a new submission process, we'll have to press the **Update** button in the dashboard. We'll be redirected to the list of the submission steps, and all the information we previously entered will be already filled.

To submit the update, we have to repeat the previous steps and change the information that has been modified. For example, we can upload a new package to replace the existing one, or we can change the price or the description. Then, we need to submit it again to start the certification process. If the update didn't involve a new package version (for example, you've just changed the price), the process will be completed very quickly and the modification will be available almost immediately.

Publishing a Windows app

From a conceptual point of view, publishing a Windows application is not very different from the process we've seen to publish a Windows Phone app. Most of the steps are similar, and we will have to upload our package and define the metadata that describe it.

The procedure is split into eight steps.

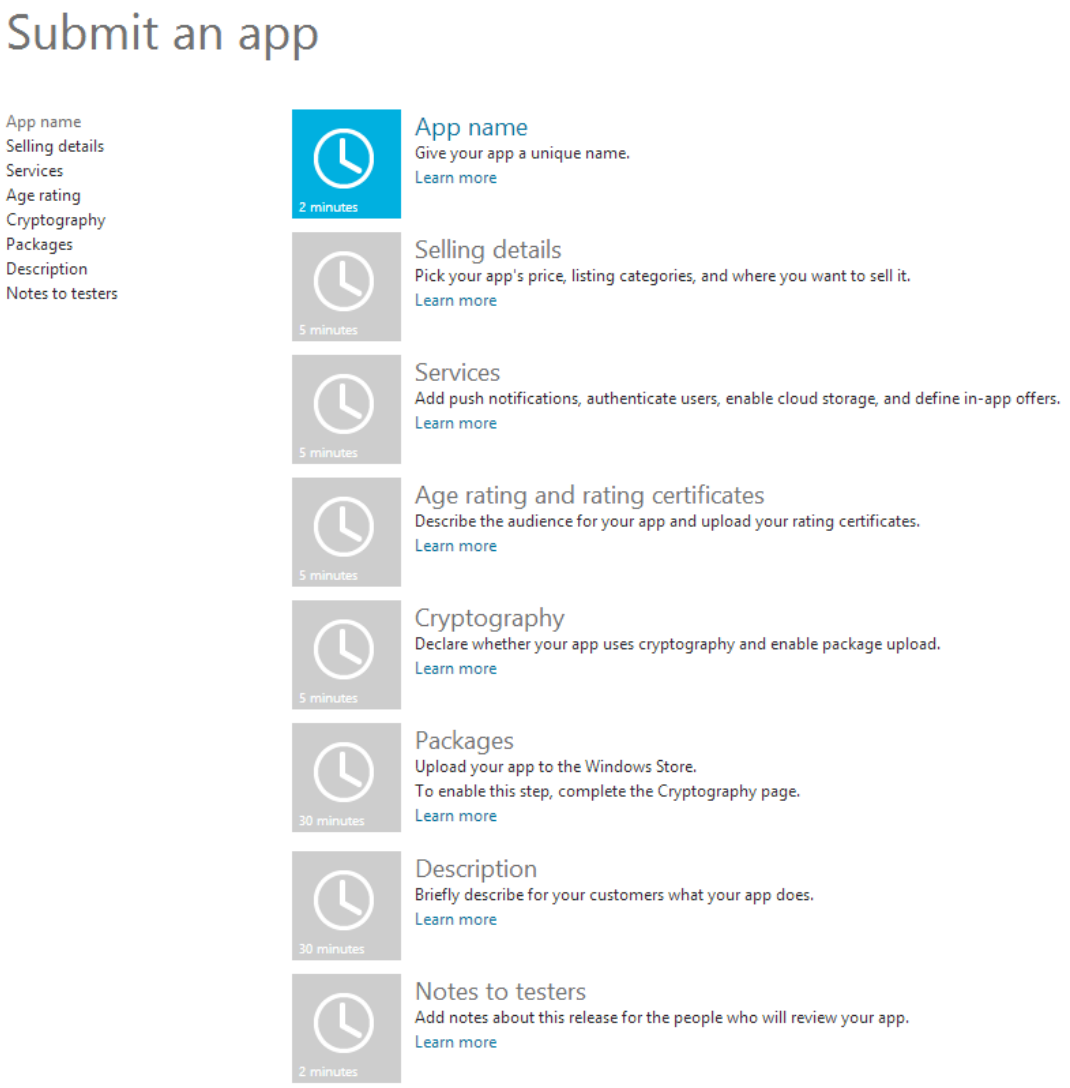


Figure 6: The steps to publish a Windows application

Step 1: App name

This step is similar to the one we've seen for Windows Phone: we will to choose the application's name, by selecting an already reserved name, or by reserving a new one. If we choose an application we have already published on Windows Phone, we'll be able to link them and enable identity sharing.

Step 2: Selling details

In this section, we can define the distribution options, which are:

- The price.
- If we support trial mode: If the answer is yes, we can decide to set an expiration date.

- If the application allows in-app purchases: If the answer yes, we can choose if we want to use the Microsoft services (in this case, we'll be redirected to the next step to define the products), or if we want to rely on our payment services.
- The markets where we will sell the application.
- The release date: The default option is **Release my app soon as it passes certification**, but we can also schedule a specific date and time.
- The category and the optional subcategory.
- The hardware requirements.
- If we support accessibility, which means that the application has been specifically designed to be easy to use for people with disabilities.

Step 3: Services

We've already seen this step in Chapter 10, when we introduced push notifications. In this section, we can set up the push notification services (by using Mobile Services provided by Azure, the cloud Microsoft solution, or by retrieving the authentication tokens to be used in our backend) and define the in-app purchase products.

Step 4: Age rating

In this section, we need to define the suggested minimum age for our application's users, based on the content we provide. This step is especially important for games, since we need to upload a certificate (called Game Definition File) provided by a certification authority, which certifies the suitable age for our application. In addition, some countries (like Brazil or Russia) require an additional set of certificates. You can read the documentation at http://s.qmatteoq.com/GameRatings to get more details about how to obtain these certificates.

Step 5: Cryptography

To be compliant with the US laws about cryptography, we need to declare if our application uses any algorithm to encrypt data like a digital signature, NTLM or SSL protocols, DRM, etc.

Step 6: Packages

This section is very important, since it allows developers to upload the .appxupload or .appxbundle packages that have been created by Visual Studio. You can simply drag and drop the files in the page or manually look for them on your computer by clicking the **Browse to files** button. Once you've chosen the file to upload, you'll need to wait for the operation to be completed before moving on. This section can be used to upload packages both for Windows 8 and Windows 8.1. Since this book is about Universal Windows apps (which are based on Windows 8.1), we won't describe how to manage Windows 8 packages.

Step 7: Description

This is another important step, since it requires the developer to define all the information that will be displayed on the Store to the user. This section is split into two parts: the first one is unique, while the second one is different according to whether we're uploading a Windows 8 or a Windows 8.1 application. As we've already stated, since this book is about Universal Windows app, we will talk only about the procedure to describe a Windows 8.1 application.

Here is the list of information requested:

- **Description:** The text that describes the application. The maximum length is 10,000 chars.
- **App features:** We can specify up to 20 features of our application that we want to highlight.
- **Screenshots:** We can upload up to nine screenshots that display how the application looks. The minimum required resolution is 1366x768. For every screenshot, we also need to include a description, with a maximum length of 200 chars.
- **Notes:** It's a text (maximum length 1,500 chars) which describes the new features added in this version. It's especially useful when you're submitting an update for an existing application.
- **Recommended hardware:** You can include a list of 11 features that describe the minimum hardware requirements for the application.
- **Keywords:** You can add up to seven keywords that are used to improve the app discoverability on the Store.
- **Copyright and trademark info:** The information about who has the legal rights to the application's content.
- **Promotional images:** A set of optional images that are used if your application is chosen to be promoted on the Store.
- **Website:** The website connected to the application.
- **Support contact info:** The developer's mail address, to get support in case of issues.
- **Privacy policy:** If we are publishing an application that uses an Internet connection, we need to provide a privacy policy, both within the app and on a website. This field contains the URL of the privacy policy page.

Step 8: Notes to testers

The last step contains only a text field, where we can specify some notes to the person that is going to test the application. For example, if the application is connected to a protected service, we need to provide the credentials to access to it.

Submitting the application

By pressing the **Review release info** button, you'll see a recap of all the information we've included for the submission. Once we've verified that everything is correct, we can submit it by pressing the **Submit for certification** button. The certification process will start, and you'll get an email notification as soon as it's completed.

Updating the application

The procedure to update a Windows application is the same we've seen for Windows Phone: when we press the **Update** button in the dashboard, we'll be redirected to the list of eight steps. This time, the previously inserted information will be already filled in.

You'll only have to change the information that you want to modify For example, if you're submitting an update, you'll have to access to the Packages section and replace the existing package with the new one. Alternatively, if you want to change the description, you can go directly into Step 7. In any case, a new submission will be sent to the Store; if there are important changes (like a new package), a new certification process will be triggered.

Deploying an application without using the Store

In some scenarios (like for testing or internal distribution in a company), you may have the requirement to deploy the application without publishing it on the Store. The first step to proceed with the manual deployment is to create a special package: you need to use, again, the **Create App Packages** option in the **Store** menu that is displayed when you right-click in Visual Studio on a Windows or Windows Phone project. However, this time, we will to answer **No** to the question **Do you want to build packages to upload to the Windows / Windows Phone Store?** This way, the procedure will create a special package that we won't be able to publish on the Store.

Deploying on Windows

The Visual Studio wizard will create a folder next to the .appxupload/.appxbundle package, which contains a set of additional files. To manually deploy the application, we'll need to copy this folder on another device and installing it by using the PowerShell script that is included in the folder. The script will take of retrieving the proper certificate, activating the developer license (the user will be required to login with his Microsoft Account, in the same way we do when we create our first project in Visual Studio) and installing the application.

To proceed with the deployment you'll have to right-click on the **Add-AppDevPackage.ps1** file included in the folder and choose the **Run with PowerShell** option: the script will be executed and it will install the application for you.

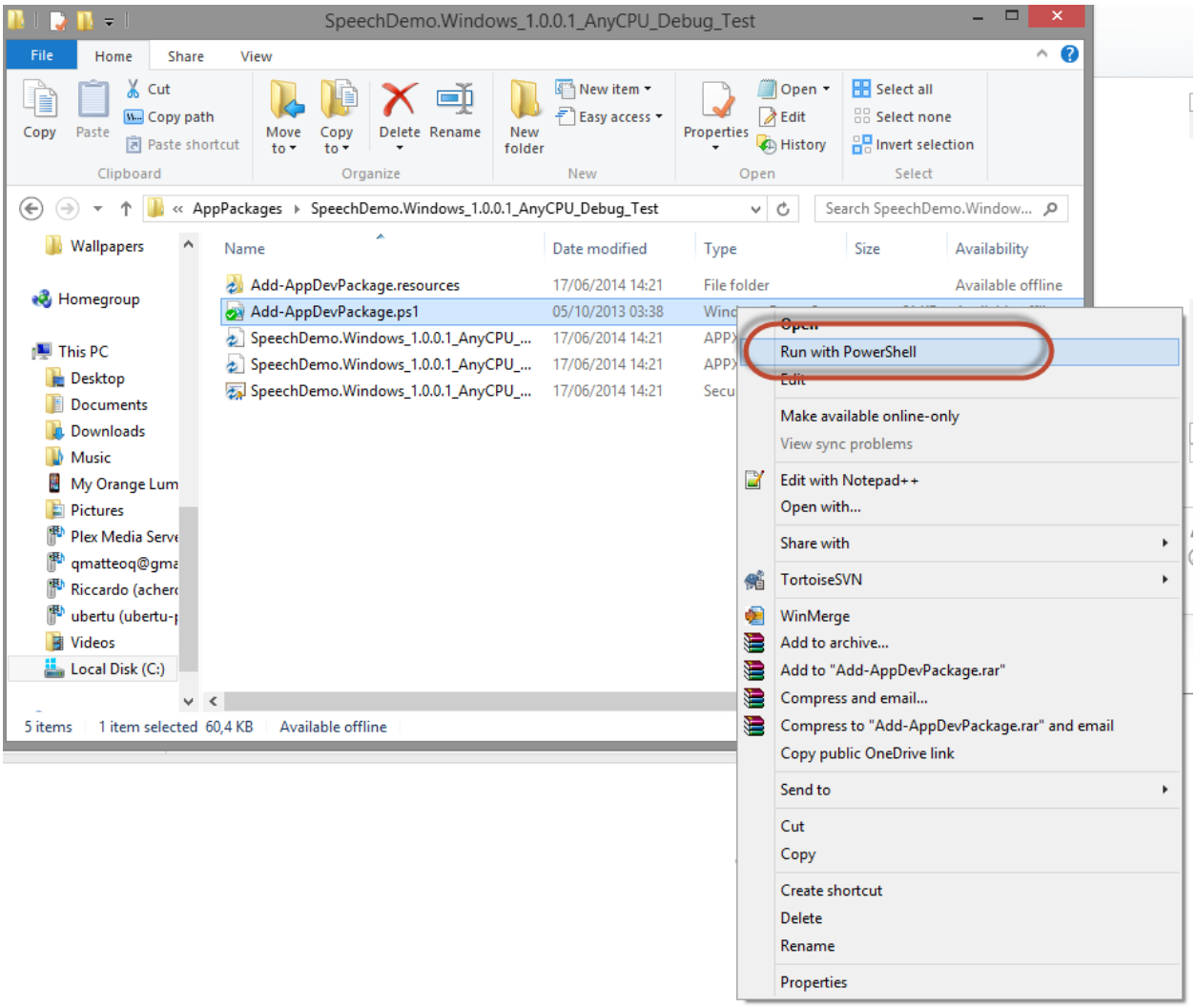


Figure 7: The option to manually deploy a Windows application on a tablet or a computer

In addition, the computer or the tablet has to be allowed to install applications without using the Store: on a computer with a Visual Studio 2013 installation, this policy is automatically applied. However, you can manually apply it by editing a key in the Windows registry. Open the Start screen, write **regedit** and launch the editor: now look for a key called **HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows**. It will contain a value called **AllowAllTrustedApps**: set it to 1 and close the editor.

Deploying on Windows Phone

Manual deployment on Windows Phone devices is allowed only for unlocked devices (see Chapter 1 for more details): you can't manually deploy apps on a regular phone.

The deployment is performed with a tool, installed with Visual Studio 2013 Update 2, called **Windows Phone Application Deployment 8.1** that you can find in the list of installed applications on your computer. It's very simple to use: from the **Target** dropdown menu you can choose the target device where to deploy the app (it can be a real device connected to the computer or one of the available emulators). Then you just need to click on the **Browse** button and look for the .appxupload or .appxbundle package generated by Visual Studio. In the end, you have to click the **Deploy** button to perform the operation.

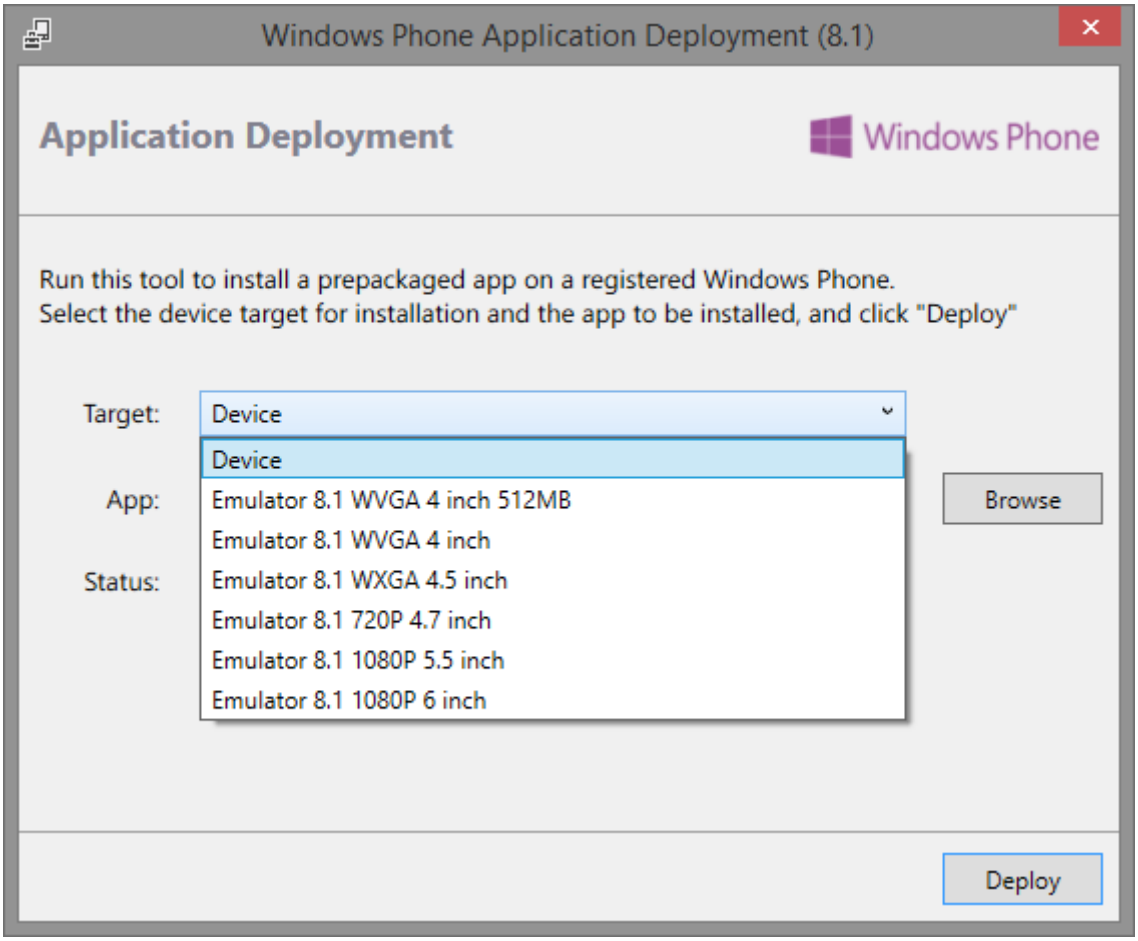


Figure 8: The Windows Phone deployment tool

Chapter 2 Interacting with the Network

Detecting Connectivity

To avoid unexpected errors in your app, it is always important to check the connectivity before performing any network operation, especially when it comes to mobile scenarios. It can happen that the user finds himself or herself in an area without network coverage so we can't take network connectivity for granted. The following code sample shows how to determine whether or not an Internet connection is available:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    ConnectionProfile profile = NetworkInformation.GetInternetConnectionProfile();
    NetworkConnectivityLevel connectivityLevel =
profile.GetNetworkConnectivityLevel();
    if (connectivityLevel == NetworkConnectivityLevel.InternetAccess)
    {
        NetworkConnection.Text = "Internet connection available";
    }
    else if (connectivityLevel == NetworkConnectivityLevel.LocalAccess)
    {
        NetworkConnection.Text = "Internet connection not available";
    }
}
```

The Windows Runtime supports multiple connection profiles (identified by the **`ConnectionProfile`** class). Each of them represents one of the connections that are available in the device. By calling the **`GetInternetConnectionProfile()`** method offered by the **`NetworkInformation`** class, we can get a reference to the connection that provides Internet connectivity.

Then, by calling the **`GetNetworkConnectityLevel()`** method on this profile, we can determine whether or not an Internet connection is available. This method will return a **`NetworkConnectivityLevel`** enumator, which can have two values: **`InternetAccess`** (which means that we have connectivity) and **`LocalAccess`** (which means that there is no Internet connection). In the previous sample, we just display a message to the user according to the connection status. In a real case scenario, we would always check that the **`NetworkConnectivityLevel`** is equal to **`InternetAccess`** before performing any network operation.

Another way to properly manage the Internet connection is by using the **NetworkStatusChanged** event, which is exposed by the **NetworkInformation** class. This event is triggered every time the status of the connection changes and it is useful to automatically perform an operation as soon as we have network connectivity. For example, a newsreader app could use this event to automatically load the latest news as soon as it detects an Internet connection. The following sample shows how to use this event:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        NetworkInformation.NetworkStatusChanged +=
NetworkInformation_NetworkStatusChanged;
    }
    private void NetworkInformation_NetworkStatusChanged(object sender)
    {
        ConnectionProfile profile =
NetworkInformation.GetInternetConnectionProfile();
        NetworkConnectivityLevel connectivityLevel =
profile.GetNetworkConnectivityLevel();
        Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            if (connectivityLevel == NetworkConnectivityLevel.LocalAccess)
            {
                MessageError.Visibility = Visibility.Visible;
            }
            else if (connectivityLevel == NetworkConnectivityLevel.InternetAccess)
            {
                MessageError.Visibility = Visibility.Collapsed;
                //load the data
            }
        });
    }
}
```

In the page constructor, we subscribe to the **NetworkStatusChanged** event. When it is triggered, we use the same code we previously saw to retrieve the current connection status and to check if we have Internet connectivity. In the sample, we have a **TextBlock** control in the page with a message that warns the user that the connection is missing. The message is displayed (by setting its **Visibility** to **Visible**) when the connection is not available. When the connection is back, we hide the message and load the data from the Internet.

Performing Network Operations: HttpClient

The Windows Runtime has introduced a new and powerful class called **`HttpClient`** that offers many methods to perform the most common network operations such as downloading or uploading a file or interacting with a web service. The **`HttpClient`** class is strictly tied to the HTTP commands; for every command offered by the HTTP protocol (such as GET, POST, or PUT), you will find a specific method to issue that command.

Note: The Windows Runtime class offers two `HttpClient` classes; the old one is part of the `Microsoft.Net.Http` namespace and it has been added in Windows 8. The new one is part of the `Windows.Web.Http` namespace and it has been introduced in Windows 8.1. In this chapter, we will focus on the new one.

Download Operations

Typically, download operations are performed by using a GET command. Consequently, you can use the **`GetAsync()`** method offered by the **`HttpClient`** class. To simplify the developer's work, the **`HttpClient`** class exposes some methods to download the most common data types such as **`GetStringAsync()`** (to download text data such as XML or JSON) or **`GetBufferAsync()`** (to download binary data such as an image).

Downloading textual content is simple: just pass to the **`GetStringAsync()`** method the URL of the text to download:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
Uri("http://feeds.feedburner.com/qmatteoq", UriKind.Absolute));
    Result.Text = result;
}
```

The method simply returns the downloaded string. In the sample, it is displayed on the page by using a **`TextBlock`** control.

Otherwise, if you need to download a binary content, you can use the **`GetBufferAsync()`** method, like in the following sample:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    IBuffer buffer = await client.GetBufferAsync(new
Uri("http://www.website.com/image.png", UriKind.Absolute));
    StorageFile storageFile = await
```

```
ApplicationData.Current.LocalFolder.CreateFileAsync("picture.png",
CreationCollisionOption.ReplaceExisting);

    await FileIO.WriteBufferAsync(storageFile, buffer);
}
```

By using the **WriteBufferAsync()** method of the **FileIO** class we learned to use in Chapter 5, we can save an image downloaded from the web to the local storage of the app.

The **HttpClient** class also offers a way to have more control over the download operation by using a generic method called **GetAsync()**. In this case, you will not get the resource's content directly in return but a **HttpResponseMessage** object instead, which contains all of the properties that define the HTTP response (such as headers, status, etc.). The following code shows the same previous sample (downloading an image and saving it to the store) but performed with generic method:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    try
    {
        HttpClient client = new HttpClient();
        HttpResponseMessage response = await client.GetAsync(new
Uri("http://www.website.com/image.png", UriKind.Absolute));
        response.EnsureSuccessStatusCode();
        IBuffer buffer = await response.Content.ReadAsBufferAsync();
        StorageFile storageFile = await
ApplicationData.Current.LocalFolder.CreateFileAsync("picture.png",
CreationCollisionOption.ReplaceExisting);
        await FileIO.WriteBufferAsync(storageFile, buffer);
    }
    catch (Exception exc)
    {
        Error.Text = exc.Message;
    }
}
```

The main differences with the previous code are:

- Since the **GetAsync()** method is generic, we do not directly receive in return the content of the resource but, rather, a generic **HttpResponseMessage** object with the response. (To get the real content, we need to access the **Content** property, which offers some methods to read it based on the type. In this sample, since we are downloading an image (which is a binary content), we call the **ReadAsBufferAsync()** method.)
- We can see one of the advantages of using the generic method: the **HttpResponseMessage** object offers a method called **EnsureSuccessStatusCode()**, which raises an exception in case something went wrong during the operation. (For example, the resource was not available so the request returned a 404 error. This way,

we make sure that we can properly catch any error before performing any operation with the content.)

Upload Operations

Upload operations are usually performed with a POST HTTP command. Consequently, we can use the **PostAsync()** method offered by the **HttpClient** class. To define the content of the request that we want to send, the Windows Runtime offers a generic interface called **IHttpContent**. In the run time, you can find many objects that already implement this class such as **HttpStreamContent** (to send a binary file), **HttpStringContent** (to send a text) or **HttpMultipartFormDataContent** (to send a web form). The following sample shows how to send a file to a web 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();
    IHttpContent content = new HttpStreamContent(accessStream.GetInputStreamAt(0));
    await client.PostAsync(new Uri("http://wp8test.azurewebsites.net/api/values",
UriKind.Absolute), content);
}
```

By using the APIs we learned in Chapter 5, we retrieve the stream with the content of an image stored in the local storage. Then, since we are dealing with a file (i.e., binary content) we create a new **HttpStreamContent** object, passing as parameter the content's stream. In the end, we simply call the **PostAsync()** method passing as parameters the URLs to where you want to perform the upload and to the content you want to send.

Managing the Progress

A common requirement when you are developing an app that can download or upload big files is to track the progress of the operation. This way, the user can have visual feedback on what is going on and how much time is left until the operation is completed. To achieve this goal, we can use the generic methods offered by the **HttpClient** class, which give us access to an event called **Progress**, which is triggered every time a new batch of data has been downloaded or uploaded. Let's take a look at the following sample:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    try
    {
        HttpClient client = new HttpClient();
        var operation = client.GetAsync(
```

```
        new Uri("http://www.website.com/image.png", UriKind.Absolute));

        operation.Progress = (result, progress) =>
        {
            Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
            {
                if (progress.TotalBytesToReceive.HasValue)
                {
                    double bytesReceived = progress.BytesReceived;
                    double totalBytesToReceive = progress.TotalBytesToReceive.Value;
                    double percentage = (bytesReceived / totalBytesToReceive) * 100;
                    Progress.Value = percentage;
                }
            });
        };

        HttpResponseMessage response = await operation;
        response.EnsureSuccessStatusCode();
        IBuffer buffer = await response.Content.ReadAsBufferAsync();
        StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("picture.png");
        await FileIO.WriteBufferAsync(file, buffer);
    }
    catch (Exception exc)
    {
        Error.Text = exc.Message;
    }
}
```

As you can see, the biggest difference with the previous code is that we are invoking the **`GetAsync()`** method without using the **`await`** prefix. This way, instead of immediately starting the download, we simply get a reference to the Task. This way, we are able to subscribe to the **`Progress`** event, which provides in the parameters some useful properties to understand the status of the operation:

- If it is a download operation, we can use the properties called **`BytesReceived`** (which contains the amount of bytes previously downloaded) and **`TotalBytesToReceive`** (which contains the size of the file to download in bytes)
- If it is an upload operation, we can use the properties called **`BytesSent`** (which contains the amount of bytes previously sent) and **`TotalBytesToSend`** (which contains the size of the file to upload in bytes)

In the event handler, thanks to this information, we are able to calculate the percentage of the operation status and, similar to the previous sample, we can assign it to the **`Value`** property of a **`ProgressBar`** control to visually display the status to the user. Please note that we have used the **`Dispatcher`** to perform this operation; the **`Progress`** event is, in fact, executed on a separate thread while the **`ProgressBar`** control is managed by the UI thread.

Once we have defined the **Progress** event, we can effectively start the network operation by invoking the **operation** object with the **await** prefix. From this point, the code will proceed as usual; when the operation is completed, we get the content of the downloaded file and can store it in the local storage.

Performing Download and Upload Operations in Background

All of the previous code samples have a limitation, which is connected to the app's life cycle that we learned about in Chapter 4. When the app is suspended, all of the active operations are terminated, including network transfers. If your app uses the network connectivity just to download small files (such as a response from a web service in XML or JSON), it is not a big deal. Typically, the user will not close the app until the data is downloaded, otherwise he or she will not be able to properly use the app. However, this consideration no longer applies when we are talking about apps that can download big amounts of data such as music or videos.

For these scenarios, the Windows Runtime has introduced some classes that are able to keep a download or an upload operation alive even when the app is suspended. Even if these operations are directly managed by the OS by using a separate process, they still belong exclusively to the app. Our Windows Store app will just be able to manage its download and upload operations.

Downloading a File in Background

To download a file in background, you need to use the **BackgroundDownloader** class, which creates a download operation starting from a source URL (i.e, the file to download) and a destination StorageFile (i.e., where you want to store the file). In fact, it is important to highlight that, unlike the **HttpClient** class, the **BackgroundDownloader** class is able to download a file only directly to storage; it can't keep it in memory since the download can terminate when the app is not running.

Here is a background download sample:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    StorageFile file =
    await
    ApplicationData.Current.LocalFolder.CreateFileAsync("demo.zip",
    CreationCollisionOption.ReplaceExisting);
    Uri uri = new
Uri("http://www.communitydays.it/content/downloads/2014/w807_demo.zip",
UriKind.Absolute);
    BackgroundDownloader downloader = new BackgroundDownloader();
    DownloadOperation download = downloader.CreateDownload(uri, file);
    await download.StartAsync();
}
```

After we create a **BackgroundDownloader** object, we can define the download by calling the **CreateDownload()** method, which requires as parameters the source URL (the file to download) and a **StorageFile** object, which represents the file where you want to write the downloaded content. In the previous sample, we created a file called **demo.zip** in the local storage in which we want to save our content. The download operation is queued by using the **StartAsync()** method. From now on, the OS will be in charge of managing it for us.

Since you are going to use this class especially to download big files, it is more important than ever to display the status to the user by using a progress bar. We can do this by using an approach similar to the one we used with the **HttpClient** class, as you can see in the following sample:

```
private async void OnStartDownloadClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("demo.zip",
CreationCollisionOption.ReplaceExisting);
    Uri uri = new
Uri("http://www.communitydays.it/content/downloads/2014/w807_demo.zip",
UriKind.Absolute);
    BackgroundDownloader downloader = new BackgroundDownloader();
    DownloadOperation download = downloader.CreateDownload(uri, file);
    Progress<DownloadOperation> progress = new Progress<DownloadOperation>();
    progress.ProgressChanged += progress_ProgressChanged;
    await download.StartAsync().AsTask(progress);
}

void progress_ProgressChanged(object sender, DownloadOperation e)
{
    double bytesReceived = e.Progress.BytesReceived;
    double totalBytes = e.Progress.TotalBytesToReceive;
    double percentage = (bytesReceived / totalBytes) * 100;
    Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        Progress.Value = percentage;
    });
}
```

We create a new **Progress<DownloadOperation>** object that exposes an event called **ProgressChanged**, which is triggered every time a new batch of data is downloaded. The operation performed in the event handler is the same as the one we saw for the **HttpClient** class. We simply calculate the progress percentage and display it by using a **ProgressBar** control. In the end, we call, as usual, the **StartAsync()** method to start the operation but with a difference: we convert it into a **Task** by using the **AsTask()** method, which accepts as parameter the **Progress** object we previously created. This way, the download will not just start but it will also trigger the **ProgressChanged** event every time the progress of the download changes.

Uploading a File in Background

The upload operation is very similar to the download one, as you can see in the following code:

```
private async void OnStartUploadClicked(object sender, RoutedEventArgs e)
{
    BackgroundUploader uploader = new BackgroundUploader();
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("demo.zip");
    Uri destination = new Uri("http://www.qmatteoq.com/upload.aspx",
UriKind.Absolute);
    UploadOperation operation = uploader.CreateUpload(destination, file);
    await operation.StartAsync();
}
```

The only difference is that, in this case, we use the **BackgroundUploader** class that offers the **CreateUpload()** method to define the upload operation, which requires the file to reach the destination URL and the file to send (represented, as usual, by a **StorageFile** object). However, we also have the chance to send multiple files with one single operation, as you can see in the following sample:

```
private async void OnStartUploadClicked(object sender, RoutedEventArgs e)
{
    List<BackgroundTransferContentPart> files = new
List<BackgroundTransferContentPart>();
    FileOpenPicker picker = new FileOpenPicker();
    var selectedFiles = await picker.PickMultipleFilesAsync();
    if (selectedFiles != null)
    {
        foreach (StorageFile selectedFile in selectedFiles)
        {
            BackgroundTransferContentPart part = new BackgroundTransferContentPart();
            part.SetFile(selectedFile);
            files.Add(part);
        }
    }
    BackgroundUploader uploader = new BackgroundUploader();
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("demo.zip");
    Uri destination = new Uri("http://www.qmatteoq.com/upload.aspx",
UriKind.Absolute);
    UploadOperation operation = await uploader.CreateUploadAsync(destination, files);
    await operation.StartAsync();
}
```

We are using the **FileOpenPicker** object that we learned to manage in Chapter 5. Thanks to the **PickMultipleFileAsync()** method, the user is able to select multiple files from his or her phone that are returned to the app in a collection. For every file in the collection, we create a new operation part, identified by the **BackgroundTransferContentPart** class. We add the file simply by calling the **SetFile()** method and then we add the part to a collection we previously defined.

In the end, we can use an override of the **CreateUploadAsync()** method of the **BackgroundUploader** class which accepts (instead of a single file) a collection of **BackgroundTransferContentPart** objects. The work is done. By calling the **StartAsync()** method, we will start the upload of all of the specified files.

Managing the App's Termination

As you have learned in Chapter 4, the app can be terminated in case of low resources by the OS. In this case, since the app is suspended, the background operations are also terminated. Consequently, when the app is reactivated, we will need to manually reconnect the background operations to the apps since they have not been kept in memory. For this reason, the **BackgroundDownloader** and **BackgroundUploader** classes offers a method that retrieves all of the download or upload operations issued by the apps. Let's take a look at a download sample:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    var downloads = await BackgroundDownloader.GetCurrentDownloadsAsync();
    foreach (DownloadOperation download in downloads)
    {
        Progress<DownloadOperation> progress = new Progress<DownloadOperation>();
        progress.ProgressChanged += progress_ProgressChanged;
        await download.AttachAsync().AsTask(progress);
    }
}
private void progress_ProgressChanged(object sender, DownloadOperation e)
{
    double bytesReceived = e.Progress.BytesReceived;
    double totalBytes = e.Progress.TotalBytesToReceive;
    double percentage = (bytesReceived / totalBytes) * 100;
    Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        Progress.Value = percentage;
    });
}
```

By using the **GetCurrentDownloadsAsync()** method, we retrieve the list of all of the download operations that belong to the app. Then, for each of them, we create a new **Progress<DownloadOperation>** object, we subscribe to the **ProgressChanged** event, and we call the **AttachAsync()** method to perform the reconnection. You will notice that the **ProgressBar** control will not start from the beginning but, instead, from the last known position before the app was terminated.

Of course, you can also use the same approach for upload operations. In this case, we will need to use the **GetCurrentUploadsAsync()** method of the **BackgroundUploader** class.

Interacting with Services

Services are one of the most widely used approaches when it comes to designing a software solution. In the past, typically the apps (websites, desktop apps, etc.) and the backend with the data (such as a database) were tightly connected. The app established a direct connection with the backend and took care of performing the required operations such as retrieving some data or inserting new information. This approach does not fit well in the mobile world since it requires a stable and fast connection between the app and the backend (which can't be taken for granted on a tablet or smartphone).

Services are a way to decouple this tight connection since they act as an intermediary between the backend and the client. The service will take care of performing the operations on the backend and prepare the data so that it can be easily consumed by the mobile app. In addition, this approach is useful when we need to support multiple apps. Since the service outputs the data by using standard technologies (such as XML or JSON), they can be accessed from any app whether or not it is a website or desktop app or a mobile app for Windows, Windows Phone, iOS or Android.

REST Services

Today, the most common approach to creating web services for mobile apps is by using Representational State Transfer (REST) services. They are popular for two main reasons:

- They rely on the HTTP protocol so the operations are defined by using the standard commands of the protocol such as GET or POST
- They return the data by using standard formats such as XML or JSON

Consequently, virtually any technology is able to interact with web services. No matter which platform and language you have chosen to write your mobile app in, you will always have the chance to perform HTTP requests and to parse XML or JSON data.

When it comes to Windows Store apps, the easiest way to interact with REST services is by using the **HttpClient** class we previously saw since it already offers a method for every HTTP command. For example, we are going to use the **GetAsync()** method to retrieve some data from the service or the **PostAsync()** one to send a new item that needs to be stored on the backend. Since REST services return the data by using XML or JSON, we can reuse the knowledge we acquired in Chapter 5 about serialization. Thanks to the **DataContractSerializer** and **DataContractJsonSerializer** classes, we are able to automatically convert the plain data into a collection of objects, which we can manipulate in our Windows Store app. And vice versa: when we need to send some data to the service, we need to convert our objects into a XML or JSON structure.

For example, let's say that we need to interact with a REST service, which is connected to a database that contains a set of customers. One of the operations offered by the service is to return a list of people, like in the following sample:

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

As we did in Chapter 5, the first thing we will need now is a class that maps this JSON data:

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

By combining what we have learned at the beginning of this chapter about the **HttpClient** class and the usage of the **DataContractJsonSerializer** class in Chapter 5, it should be easy to understand the next code sample:

```
private async void OnConsumeServiceClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
```

```
Uri("http://wp8test.azurewebsites.net/api/values", UriKind.Absolute));
    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>;
    }
}
```

By using the **GetStringAsync()** method of the **HttpClient** class, we retrieve the JSON response from the service. Then, by using the **DataContractJsonSerializer** class and the **ReadObject()** method, we convert the JSON data into a collection of Person objects. Since the **ReadObject()** method requires a stream as input and not directly the JSON string, we first need to convert it into a **MemoryStream** object.

And vice versa: here is how we can perform a POST operation to send some data to the service:

```
private async void OnPostDataClicked(object sender, RoutedEventArgs e)
{
    Person person = new Person
    {
        Name = "Matteo",
        Surname = "Pagani"
    };
    DataContractJsonSerializer serializer = new
DataContractJsonSerializer(typeof(Person));
    MemoryStream stream = new MemoryStream();
    serializer.WriteObject(stream, person);
    string result = Encoding.UTF8.GetString(stream.ToArray(), 0, (int)stream.Length);
    IHttpContent content = new HttpStringContent(result);
    HttpClient client = new HttpClient();
    await client.PostAsync(new
Uri("http://wp8test.azurewebsites.net/api/value",UriKind.Absolute), content);
}
```

In this case, we are performing the opposite operation. We take our objects (in this case, it is a sample **Person** object) and we convert it into a JSON structure using the **WriteObject()** method of the **DataContractJsonSerializer** class. Again, we need to use a **MemoryStream** since the method is able to write the result only to a stream and not directly to a string. In the end, we execute the **PostAsync()** method of the **HttpClient** class. Since the service accepts the data as a string, we encapsulate the JSON into a **HttpStringContent** object that is passed as parameter of the method, together with service's URL.

Tip: When it comes to defining the mapping between a class and JSON, the operation sometimes is not easy to accomplish, especially if the JSON is complex. To make this process easier, Visual Studio offers a feature can do this for you; just copy in the clipboard the JSON data returned by your service and choose Edit -> Paste Special -> JSON as class.

Using Json.NET

In the previous samples, we saw how using the **`DataContractJsonSerializer`** class for interacting with REST services is not very straightforward. Despite the fact that we are working with strings, we always need to convert them into a **`MemoryStream`** object in order to perform all of the required operations.

We can simplify our code by introducing Json.NET, a popular third-party library that is able to handle serialization in a better way by offering simple methods and better performance. In addition, it offers a powerful language called LINQ to JSON in order to perform complex operations on the JSON data. Json.NET is available as a NuGet package here while the official website (found here) hosts the documentation.

Let's take a look at how (thanks to Json.NET) we are able to simplify the two previous code samples in order to interact with a REST service. Let's start with the download sample:

```
private async void OnConsumeServiceClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
Uri("http://wp8test.azurewebsites.net/api/values", UriKind.Absolute));
    List<Person> people = JsonConvert.DeserializeObject<List<Person>>(result);
}
```

The deserialization procedure is performed by using the **`JsonConvert`** class, which offers a **`DeserializeObject<T>()`** method where **`T`** is the type of data we expect in return. As input, it simply requires the JSON string we just downloaded from the service by using the **`HttpClient`** class.

Here is the reverse process to send some data to the service:

```
private async void OnConsumeServiceClicked(object sender, RoutedEventArgs e)
{
    Person person = new Person
    {
        Name = "Matteo",
        Surname = "Pagani"
    };
    string result = JsonConvert.SerializeObject(person);
```

```
    IHttpContent content = new HttpStringContent(result);
    HttpClient client = new HttpClient();
    await client.PostAsync(new Uri("http://wp8test.azurewebsites.net/api/value",
UriKind.Absolute), content);
}
```

Again, the operation is performed by using the **JsonConvert** class. However, in this case, we use the **SerializeObject()** method, which requires as parameter the object to convert and it simply returns the serialized string. Now we can continue the execution as we did in the previous sample. We encapsulate the JSON string into a **HttpStringContent** object and we send it by using the **PostAsync()** method of the **HttpClient** class.

Controlling the Serialization

Json.NET offers some useful attributes to control the serialization and deserialization process so that we can manually define the mapping between the object's properties and the JSON properties. By default, JSON properties are serialized by using the same name of the object's properties (so, for example, the **Name** property of the **Person** class is converted into a **Name** property in the JSON file). However, thanks to attributes, we are able to change this behavior. Let's take a look at a real sample:

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

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

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

Thanks to the **JsonProperty** attribute that has been applied to every property, we have been able to manually define which name to use when the property is translated into a JSON file. The previous example is common in real apps since C# uses a different notation than the JSON one. In C#, properties usually start with an uppercase letter while in JSON they start with a lowercase letter.

Another way to control the serialization is by using a class called **JsonSerializerSettings**, which offers many settings such as how dates, errors or numbers should be managed. The following sample shows another common scenario, which is **null** values management:

```
private async void OnConsumeServiceClicked(object sender, RoutedEventArgs e)
{
    Person person = new Person
    {
```

```
        Name = "Matteo",
        Surname = "Pagani"
    };
    JsonSerializerSettings settings = new JsonSerializerSettings();
    settings.NullValueHandling = NullValueHandling.Ignore;
    string result = JsonConvert.SerializeObject(person, settings);
    IHttpContent content = new HttpStringContent(result);
    HttpClient client = new HttpClient();
    await client.PostAsync(new Uri("http://wp8test.azurewebsites.net/api/value",
UriKind.Absolute), content);
}
```

By default, when Json.NET tries to serialize a property with a **`null`** value, it includes it anyway in the JSON file and sets it to **`null`**. However, some services do not manage this approach well. If they find some **`null`** properties in the JSON, they raise an error. Thanks to the **`JsonSerializerSettings`** class, we are able to tell Json.NET not to include in the JSON the empty properties. We do so by setting the **`NullValueHandling`** property to **`NullValueHandling.Ignore`**. As you can see, the **`SerializeObject()`** method of the **`JsonConvert`** class can accept a second parameter, which is the **`JsonSerializerSettings`** object we previously defined.

LINQ to JSON

When you download a JSON response from a service and you want to convert it into objects that can be manipulated in code, the deserialization process is helpful since it takes care of automatically performing the conversion. However, sometimes it is not our scenario. For example, we could download a complex JSON but we would need to extract just some of its properties. In this case, Json.NET offers a powerful language called LINQ to JSON, which we can use to perform LINQ queries on a JSON file so that we retrieve only the data we need.

To perform such operations, we need to use the **`JObject`** class, which offers the **`Parse()`** method that is able to convert a plain JSON string in a complex structure that we can explore, like in the following sample:

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

Now, let's see what the most common operations are that we can perform by using the **`JObject`** class.

Simple JSON

Let's assume that you have a simple JSON string, like the following one:

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

In this case, the **JObject** class behaves like a **Dictionary<string, object>** collection, so we can retrieve the properties simply by referring to them with their name, like in the following sample:

```
private async void OnParseJson(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
Uri("http://wp8test.azurewebsites.net/api/values/1", UriKind.Absolute));
    JObject json = JObject.Parse(result);
    string value = json["Name"].Value<string>();
}
```

To extract the value of the property, we use the **Value<T>()** method where **T** is the property's type. This way, the value is automatically converted to the proper type (to a string, in this sample).

Complex JSON

Similar to C#, a JSON string can also contain complex objects, where a property is represented by another object, like in the following sample:

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

Address is a complex property since it contains other subproperties such as **Street** and **City**. To access to these properties, we need to use the **SelectToken()** method, which requires as parameter the full JSON path. The following sample shows how to extract the value of the **City** property:

```
private async void OnParseJson(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
Uri("http://wp8test.azurewebsites.net/api/values/1", UriKind.Absolute));
    JObject json = JObject.Parse(result);
    string city = json.SelectToken("Address.City").Value<string>();
}
```

Collections

As we saw in previous samples in this chapter, JSON can also be used to store collections of items. In this case, we can use the **Children()** method of the **JObject** class to return all of the items that belong to the collection. The following sample shows how to create a subcollection that contains only the value of the **Name** property for each item:

```
private async void OnGetDataClicked(object sender, RoutedEventArgs e)
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(new
Uri("http://wp8test.azurewebsites.net/api/values", UriKind.Absolute));
    JObject json = JObject.Parse(result);
    List<string> list = json.Children().Select(x =>
x["Name"].Value<string>()).ToList();
}
```

Working with RSS Feeds

RSS is a widely used standard to aggregate a set of information (such as the news published by a website) into a single XML file that it can be easily processed by a feed reader or by any app that is able to perform HTTP requests and parse the XML data. The following sample shows part of the RSS feed of my blog:

```
<rss version="2.0">
  <channel>
    <title>qmatteoq.com</title>
    <link>http://wp.qmatteoq.com</link>
    <description>Windows Phone, Windows 8 and much more</description>
```

```
    <lastBuildDate>Tue, 29 Jul 2014 13:30:00 +0000</lastBuildDate>
    <language>en-US</language>
    <item>
      <title>Using Caliburn Micro with Universal Windows app – Design time
data</title>
      <link>http://feedproxy.google.com/~r/qmatteoq_eng/~3/hT1nxxxn9rY/</link>
      <comments>http://wp.qmatteoq.com/using-caliburn-micro-with-universal-windows-
app-design-time-data/#comments</comments>
      <pubDate>Tue, 29 Jul 2014 13:30:00 +0000</pubDate>
      <category><![CDATA[Universal Apps]]></category>
      <category><![CDATA[Windows 8]]></category>
      <category><![CDATA[Windows Phone]]></category>
      <category><![CDATA[Caliburn]]></category>
      <guid isPermaLink="false">http://wp.qmatteoq.com/?p=6296</guid>
      <description><![CDATA[One of the most powerful features in XAML is design time
data support. Let's say that you have an app that displays some news retrieved from a
RSS feed using a ListView or a GridView control. If you try to edit the visual layout
of your app using Blend or the Visual Studio designer, you…]]></description>
    </item>
 </channel>
</rss>
```

Theoretically, the knowledge acquired in this chapter should be enough to properly manage a RSS feed. Similar to the JSON data we previously saw, we could simply download the RSS content by using the **HttpClient** class and convert it into objects by using the **DataContractSerializer** class or LINQ to XML. However, the Windows Runtime includes a set of classes that are able to do this operation for us. We will just have to provide the URL of the RSS feed in order to get in return a collection of objects with all of the feed items.

The following sample shows how to perform some basic operations:

```
private async void OnDownloadFeedClicked(object sender, RoutedEventArgs e)
{
    SyndicationClient client = new SyndicationClient();
    SyndicationFeed feed = await client.RetrieveFeedAsync(new
Uri("http://feeds.feedburner.com/qmatteoq_eng", UriKind.Absolute));
    Title.Text = feed.Title.Text;
    Description.Text = feed.Subtitle.Text;
    NumberOfItems.Text = feed.Items.Count.ToString();
}
```

The download operation is performed by using the **SyndicationClient** class, which offers a method called **RetrieveFeedAsync()** that simply requires as parameter the RSS feed's URL. What we get in return is a **SyndicationFeed** object, which contains a set of properties that are mapped with the data stored in the XML file. In the previous sample, we extract the title, the subtitle, and number of items, and we display them to the user by using a set of **TextBlock** controls.

All of the items that are included in the feed are stored in a collection called Items. Each item is represented by the **SyndicationItem** class, which offers a set of properties that map the XML ones. The following sample shows how to retrieve the first news of the feed and display to the user its title and summary (stored in the **Title** and **Summary** properties):

```
private async void OnDownloadFeedClicked(object sender, RoutedEventArgs e)
{
    SyndicationClient client = new SyndicationClient();
    SyndicationFeed feed = await client.RetrieveFeedAsync(new
Uri("http://feeds.feedburner.com/qmatteoq_eng", UriKind.Absolute));
    if (feed.Items.Count > 0)
    {
        SyndicationItem item = feed.Items.FirstOrDefault();
        FirstNewsTitle.Text = item.Title;
        FirstNewsSummary.Text = item.Summary;
    }
}
```

Chapter 3 Interacting with the Real World

The Geolocation Services

The geolocation services offered by the Windows Runtime can be used to track the user's position in the world; these services are available on both Windows and Windows Phone. In fact, the OS can determine the current position by using different approaches such as a Global Positioning System (GPS) sensor or the cellular or Wi-Fi connection. The approach that is used is transparent to the user. The Windows Runtime will take care of choosing the best one for us, according to the requirements and hardware configuration of the device.

Note: To use the geolocation service, you need to enable the Location capability in the manifest file.

The main class we are going to use to interact with the geolocation services is called **`Geolocator`**, which is part of the **`Windows.Devices.Geolocation`** namespace.

Retrieving the Status of the Services

For many reasons, the geolocation services might not be available at a specific moment. For example, perhaps the user is in an indoor place and their device can't reach the GPS or perhaps the area is not covered by a cellular network. Consequently, it is important to always check that the geolocation services are properly working before performing any operation with the **`Geolocator`** class.

However, there are two different approaches when it comes to developing a Windows or Windows Phone app. In Windows, we need to use the **`Geolocator`** class in the regular way but we need to be ready to intercept the **`UnauthorizedAccessException`** that is raised in case something goes wrong. Let's take a look at the following sample:

```
private async void OnGetPositionClicked(object sender, RoutedEventArgs e)
{
    try
    {
        Geolocator geolocator = new Geolocator();
        Geoposition position = await geolocator.GetGeopositionAsync();
    }
    catch (UnauthorizedAccessException exc)
    {
        DeviceAccessInformation accessInformation =
DeviceAccessInformation.CreateFromDeviceClass(DeviceClass.Location);
        if (accessInformation.CurrentStatus == DeviceAccessStatus.DeniedBySystem)
        {
            Error.Text = "The geolocation services have been disabled for the
```

```
system";
        }
        else if (accessInformation.CurrentStatus == DeviceAccessStatus.DeniedByUser)
        {
            Error.Text = "The geolocation services have been disabled for the app ";
        }
    }
}
```

As you can see, the usage of the **Geolocator** class (we will explain later about how to use it) is encapsulated inside a **try / catch** statement. In case we get an exception, we can use the **DeviceAccessInformation** class to determine the cause. In fact, the services could have been blocked for the entire OS (we get the value **DeniedBySystem**) or just for the current app (we get the value **DeniedByUser**). The first scenario is managed by the Windows settings; the user can choose to disable the geolocation services for all of the apps. The second scenario is directly managed by the app. The first time we are going to use the **Geolocator** class, a warning message will be displayed to the user, asking them for confirmation to proceed.

In Windows Phone, we have a way to determine the status of the services before performing any operation; we just need to check the **LocationStatus** property of the **Geolocator** class. The **PositionStatus** enumerator will help us to understand the status. If its value is **Disabled**, it means that the user has disabled the service in the phone's settings. When the services are active and ready to be used, we will get the **Ready** value. The **Geolocator** class also offers an event called **StatusChanged**, which we can subscribe if we want to be immediately notified every time the status of the services changes.

```
private void OnGetPositionClicked(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    if (geolocator.LocationStatus == PositionStatus.Disabled)
    {
        Error.Text = "Geolocation services are disabled";
    }
}
```

Retrieving the User's Position

The simplest way to determine the user's position is to use the **GetGeopositionAsync()** method, which performs a single acquisition. The method returns a **Geoposition** object, which contains all of the information about the current location in the **Coordinate.Point.Position** property such as **Latitude**, **Longitude** and **Altitude**.

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

```
    Geolocator geolocator = new Geolocator();
    Geoposition position = await geolocator.GetGeopositionAsync();
    Longitude.Text = position.Coordinate.Point.Position.Longitude.ToString();
    Latitude.Text = position.Coordinate.Point.Position.Latitude.ToString();
}
```

Another approach is to subscribe to an event called **PositionChanged**. This way, we can track the user's position and be notified every time he or she moves away from his or her previous position. We can customize the frequency of the notification with two parameters: **MovementThresold** (which is the distance, in meters, that should be travelled from the previous point) and **ReportInterval** (which is the timeframe, in milliseconds, that should pass before one notification and the next). Here is a sample code to continuously track the user's position:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.MovementThreshold = 100;
    geolocator.ReportInterval = 1000;
    geolocator.PositionChanged += geolocator_PositionChanged;
}

async void geolocator_PositionChanged(Geolocator sender, PositionChangedEventArgs
args)
{
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        Geoposition currentPosition = args.Position;
        Latitude.Text =
currentPosition.Coordinate.Point.Position.Latitude.ToString();
        Longitude.Text =
currentPosition.Coordinate.Point.Position.Longitude.ToString();
    });
}
```

The event handler that we subscribed to the **PositionChanged** event contains a parameter, which has a property called **Position**. It is a **Geoposition** object, which includes the coordinates of the current user's position inside the **Coordinate.Point.Position** property. Please note that we are using the **Dispatcher** to display these information in the page; it is required since the **PositionChanged** event is managed in a background thread.

Testing the Geolocation Services

Both Windows and Windows Phone provides some tools to makes life easier for developers who need to test geolocation services. Windows offers a tool inside the Simulator; one of the available options has a World symbol. By clicking the World symbol, you will be able to manually insert the latitude and longitude coordinates that you want to simulate.

The Windows Phone emulator offers a more advanced tool, which provides a visual map where you can place one or more pushpins. Each pushpin's coordinates will be sent to the emulator and detected by the **Geolocator** class. In addition, the tool also offers a way to create a route between multiple pushpin so that you can simulate the user's travel by car, by bike or on foot. Routes can also be saved as XML files so that you can load them even after you have closed the emulator.

Geofencing

Geofencing is one of the new features related to the geolocation services that have been added in Windows 8.1 and Windows Phone 8.1. Thanks to this feature, you will be able to define one or more circular areas on the map (which are called geofences). When the user enters or exits the area, your app will be notified even if it is not running at that moment (thanks to the background tasks that we will introduce in Chapter 11). There are many scenarios in which geofencing can be useful. For example, an app connected to a store brand could notify the user to some special promotions every time he or she walks near one of their shops.

An app can create up to 20,000 geofences, even if it is suggested not to exceed the 1,000 quota (otherwise they could become hard to manage). Every geofence is identified by the **Geofence** class, which is part of the **Windows.Devices.Geolocation.Geofencing** namespace. When you create a new **Geofence** object, you need to set the following parameters:

- The geofence id, which is a string that univocally identifies the geofence; an app can't create two geofences with the same id
- The geofence area, by using a **Geocircle** object, which is represented by a pair of coordinates (latitude and longitude) and the size of the area. (It is suggested not to create geofences with an area smaller than 100 meters since they could be hard to track by the device.)
- Which events should trigger the notification, by using one of the values of the **MonitoredGeofenceStates** enumeration; the possible states are **Entered** (the user entered the area), **Exited** (the user left the area), and **Removed** (the geofence has been removed from the system)
- A **boolean** value, which is used to define whether or not it is a one-time geofence; in case the value is **true**, the geofence will automatically be removed from the system once the user has entered or exited the area
- A time interval (called a dwell time) which must be spent inside or outside the area so that the notification is triggered
- A start date and time; if it is not specified, the geofence will immediately be activated
- A time interval after the start date; if it is specified, the geofence will automatically be removed from the system when this interval has passed

When the **Geofence** object has been properly configured, you can add it to the queue by using the **GeofenceMonitor** class, which has a unique instance (stored in the **Current** property) that is shared among all of the apps. Here is a complete geofencing sample:

```
private void OnSetGeofenceClicked(object sender, RoutedEventArgs e)
{
    string id = "geofenceId";
```

```
    BasicGeoposition geoposition = new BasicGeoposition();
    geoposition.Latitude = 45.8040;
    geoposition.Longitude = 9.0838;
    double radius = 500;
    Geocircle geocircle = new Geocircle(geoposition, radius);
    MonitoredGeofenceStates mask = 0;
    mask |= MonitoredGeofenceStates.Entered;
    mask |= MonitoredGeofenceStates.Exited;
    mask |= MonitoredGeofenceStates.Removed;
    TimeSpan span = TimeSpan.FromSeconds(2);
    Geofence geofence = new Geofence(id, geocircle, mask, true, span);
    GeofenceMonitor.Current.Geofences.Add(geofence);
}
```

In the previous sample, we defined a geofence with the following properties:

- It is placed in a specific position of the map (defined with a **`Geoposition`** object) and it has a radius of 500 meters
- We register to monitor all of the available states, so we will be notified every time the user enters or exits the area or every time the geofence is removed from the system
- It is a one-time geofence
- The dwell time is set to two seconds; the notification will be triggered only if the user stays or leaves the area for more than two seconds

To add the **`Geofence`** to the system, we call the **`Add()`** method on the **`Geofences`** collection offered by the **`GeofenceMonitor`** instance.

Managing the Geofence Activation

When the app is running foreground and a geofence is triggered, the system invokes an event called **`GeofenceStateChanged`** exposed by the **`GeofenceMonitor`** class.

In the event handler of this event, you have access to a **`GeofenceMonitor`** object, which is able to return (by using the **`ReadReports()`** method) the list of geofences that have been triggered. This method will return you a collection of **`GeofenceStateChangeReport`** objects; each of them contains a property called **`NewState`** which will tell you the event that triggered the geofence. Take a look at the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    GeofenceMonitor.Current.GeofenceStateChanged += Current_GeofenceStateChanged;
}

private async void Current_GeofenceStateChanged(GeofenceMonitor sender, object args)
{
    var reports = sender.ReadReports();
    foreach (GeofenceStateChangeReport report in reports)
    {
        GeofenceState state = report.NewState;
```

```
        switch (state)
        {
            case GeofenceState.Entered:
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
                {
                    MessageDialog dialog = new MessageDialog("The user entered into
the area");
                    await dialog.ShowAsync();
                });
                break;
            case GeofenceState.Exited:
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
                {
                    MessageDialog dialog = new MessageDialog("The user has left the
area");
                    await dialog.ShowAsync();
                });
                break;
            case GeofenceState.Removed:
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
                {
                    MessageDialog dialog = new MessageDialog("The geofence has been
removed");
                    await dialog.ShowAsync();
                });
                break;
        }
    }
}
```

For each report stored in the collection returned by **`ReadReports()`** method, we check the value of the **`NewState`** property. In the previous sample, we manage all of the possible states, which are represented by the **`GeofenceState`** enumerator, by using a **`switch`** statement such as **`Entered`**, **`Exited`** or **`Removed`**. It is up to you to perform the operation that best fits your scenario. For example, you could send a notification or display a specific page to the user. In the sample, we just displayed a pop-up message to the user with a description of what has happened. Please note that the **`GeofenceStateChanged`** event is managed in a background thread; we need to use the **`Dispatcher`** if we want to interact with the controls placed in the page.

The code we have just seen can also be used in a background task, which is a special class that contains some code that can also be performed when the app is not running. This way, we will always be able to detect the geofences activation, even if the user is not using our app. You will see in detail how to create such a background task in Chapter 11.

Interacting with a Map

One of the most common scenarios in which you can add support to the geolocation services in your app is displaying a user's position on a map. Unfortunately, map management is one of the most complex features to manage in a Universal Windows app since the developer story is not shared between the two platforms. Windows offers a control based on the Bing Maps services while Windows Phone uses a control based on the services offered by the Here services. Consequently, the only way to include any code related to map management (whether it is the XAML definition or it is the logic to interact with the map) in a shared project is by using conditional compilation.

Using Maps on Windows Phone

Windows Phone 8.1 has introduced a new map control that is specific for Windows Store apps. If you are developing a Silverlight app, you will continue to use the map control that was available in 8.0. This control is called **`MapControl`** and it is included in the **`Windows.UI.Xaml.Maps`** namespace. You will need to declare it in your page before using it, like in the following sample:

```
<Page
x:Class="MapDemo.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MapDemo"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:maps="using:Windows.UI.Xaml.Controls.Maps"
mc:Ignorable="d"
Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>
        <maps:MapControl />
    </Grid>

</Page>
```

Publishing an App that Uses the MapControl

If you try to add the previous code in your app and you launch it on the emulator or on the phone, you will notice that a warning message will be displayed over the map. This warning message will claim that you have not declared a valid map service token. This token is required in order to publish an app that uses the map control on Windows Phone. This token is provided by the Dev Center during the submission process. To get it, you will need to start the publishing process (which will be detailed in Chapter 12) but, instead of completing it and submitting the app, you will just have to complete the first step. This first step asks you for some basic information such as the name, category, or price of your app.

After completing the first step, you will get access to a section called **Map services**, which will allow you to get two tokens. The first one is called **Map service ApplicationID** and it needs to be inserted in the manifest file. However, this information can't be changed by using the visual editor. Rather, you will need to right-click the Package.appxmanifest file in Visual Studio and choose View Code. You will get access to the XML definition of the manifest file. At the beginning of the file, you will find the following definition:

```
<mp:PhoneIdentity PhoneProductId="89d51a2c-cdb4-49ee-959d-83f8db75d8f3"
PhonePublisherId="00000000-0000-0000-0000-000000000000"/>
```

You will need to replace the existing value of the **PhoneProductId** attribute with the Map service ApplicationID provided by the portal.

The second token is called **Map service Authentication Token** and it is used directly in the app's code. Its first purpose is to register the **MapControl** so that you can remove the warning message. For this purpose, you will have to set this token to the **MapServiceToken** property of the control, like in the following sample:

```
<maps:MapControl MapServiceToken="K38JSFqmSBMVu0iTXR4aEg" />
```

The other purpose is to get access to the services offered by the Windows Runtime APIs that belong to the **Windows.Services.Maps** namespace. In this scenario, you will have to assign the token to the **ServiceToken** property of the **MapService** class when the page is loaded:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";
}
```

Interacting with the Map

The basic way to interact with the map control is by setting the **Center** and **ZoomLevel** properties, which define the position where you want to center the map as well as the zoom level. Both properties can be set in XAML; however, the most common scenario is to set them in code (for example, to display the user's position on the map). For this purpose, the **MapControl** class offers a method called **TrySetViewAsync()**, which automatically centers the map on a specific position with an assigned zoom level. The following sample code shows how to center the map on the current user's position detected by the device:

```


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

```
    Geolocator locator = new Geolocator();
    Geoposition geoposition = await locator.GetGeopositionAsync();
    await MyMap.TrySetViewAsync(geoposition.Coordinate.Point, 15);
}
```

The **TrySetViewAsync()** method accepts, as parameter, a **Geopoint** object with the coordinates to where you want to center the map (which is stored in the **Coordinate.Point** property of the **Geoposition** object) and the zoom level to apply.

Displaying a Pushpin

The **MapControl** class supports a way to display one or more pushpins on the map by using the **MapIcon** class, which represents a base pushpin rendered with a name and an image. A **MapIcon** object is identified by two properties:

- **Location**, which is a **Geopoint** object that identifies the pushpin's coordinates
- **Title**, which is the label displayed over the pushpin

By default, the **MapIcon** class uses a default image to render the pushpin. However, you can customize it by using the **Image** property, which requires a reference to the image's stream (identified by the **RandomAccessStreamReference** class). The following sample shows how to create a pushpin and place it on the map at the current position:

```
private async void OnAddPushpinClicked(object sender, RoutedEventArgs e)
{
    Geolocator locator = new Geolocator();
    Geoposition geoposition = await locator.GetGeopositionAsync();
    MapIcon icon = new MapIcon();
    icon.Location = geoposition.Coordinate.Point;
    icon.Title = "Your position";
    icon.Image = RandomAccessStreamReference.CreateFromUri(new Uri("ms-
appx:///Assets/logo.jpg"));
    MyMap.MapElements.Add(icon);
}
```

The pushpin created in the previous sample has a custom title and icon, which is an image included in the Assets folder of the Visual Studio project. The **MapControl** class includes a collection called **MapElements**, with all of the elements that should be displayed in overlay over the map. To display the pushpin, we just need to add it to the collection by using the **Add()** method.

However, there is another way to create a set of pushpins, which offers more flexibility to the developer. Instead of displaying just an image, we will be able to define a custom layout by using XAML and manage the user's interaction (such as tapping on the pushpin). The approach is similar to the one we learned in Chapter 3 about displaying a collection with a **ListView** or a **GridView** control. We are going to define the template of a single pushpin and then we will connect a collection of pushpins to the map. Each pushpin will be rendered by using the specified template.

We can do this by using the **MapItemsControl** class, like in the following sample:

```
<maps:MapControl x:Name="MyMap">
    <maps:MapItemsControl x:Name="Pushpins">
        <maps:MapItemsControl.ItemTemplate>
            <DataTemplate>
                <Border Background="CornflowerBlue" Tapped="OnPushpinClicked">
                    <TextBlock Text="{Binding Name}" Foreground="Black"
                               maps:MapControl.Location="{Binding Location}"
                               Style="{StaticResource TitleTextBlockStyle}" />
                </Border>
            </DataTemplate>
        </maps:MapItemsControl.ItemTemplate>
    </maps:MapItemsControl>
</maps:MapControl>
```

There is one important difference compared to a standard **DataTemplate**: the **MapControl** class offers an attached property called Location, which is used to define the pushpin's position on the map. It is a **Geopoint** object, which contains the coordinate of the position. Consequently, you will also need a custom class to represent the pushpin, like in the following sample:

```
public class PushPin
{
    public string Name { get; set; }
    public Geopoint Location { get; set; }
}
```

The last step is to define a collection of **PushPin** objects and assign it to the **ItemsSource** property of the **MapItemsControl** class, in the same way we would do with a **ListView** control:

```
private void OnAddPushpinsClicked(object sender, RoutedEventArgs e)
{
    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 9.0800;
    Geopoint geopoint1 = new Geopoint(position1);
```

```
    PushPin pushpin1 = new PushPin();
    pushpin1.Name = "Pub";
    pushpin1.Location = geopoint1;

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.8052;
    position2.Longitude = 9.0825;
    Geopoint geopoint2 = new Geopoint(position2);
    PushPin pushpin2 = new PushPin();
    pushpin2.Name = "Bank";
    pushpin2.Location = geopoint2;

    List<PushPin> pushpins = new List<PushPin>();
    pushpins.Add(pushpin1);
    pushpins.Add(pushpin2);
    Pushpins.ItemsSource = pushpins;
}
```

However, if you take a look at the **`ItemTemplate`** we defined in XAML, you will notice that we subscribed to the **`Tapped`** event of the **`Border`** control. The purpose is to manage the user's interaction so that, when the user taps on a pushpin, we can display some additional information. The following sample shows how to manage this event in order to display a pop-up message with the name of the selected location:

```
private async void OnPushpinClicked(object sender, TappedRoutedEventArgs e)
{
    Border border = sender as Border;
    PushPin selectedPushpin = border.DataContext as PushPin;
    MessageDialog dialog = new MessageDialog(selectedPushpin.Name);
    await dialog.ShowAsync();
}
```

The **`Tapped`** event (like any other event) provides a parameter called **`sender`**, which is the XAML control that triggered the event. Thanks to this parameter, we are able to get a reference to the **`Border`** control and to access its **`DataContext`**, which is the **`Pushpin`** object that has been selected. This way, it is easy to perform a cast and retrieve the value of the **`Name`** property so that we can display it to the user.

Showing a Route on the Map

The maps API offers a set of classes and methods that can calculate a route between two or more locations on the map, in the same way we can do with a satellite navigation system. We will be able to display the route directly on the map and get detailed navigation steps. The operation is performed by using the **`MapRouteFinder`** class, which offers a way to calculate a driving route (by using the **`GetDrivingRouteAsync()`** method) or a walking route (by using the **`GetWalkingRouteAsync()`** method). These two methods are able to calculate a route from point A to point B. However, you can also calculate a route with multiple waypoints by using the **`GetDrivingRouteFromWaypointsAsync()`** or **`GetWalkingRoutFromWaypointsAsync()`** methods. In both cases, we will use the **`Geopoint`** class to identify the places' locations.

The following sample shows how to calculate the driving route between two points:

```
private async void OnShowRouteClicked(object sender, RoutedEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";

    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 9.0800;
    Geopoint startPoint = new Geopoint(position1);

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.4608;
    position2.Longitude = 9.1763;
    Geopoint endPoint = new Geopoint(position2);

    MapRouteFinderResult result = await
MapRouteFinder.GetDrivingRouteAsync(startPoint, endPoint, MapRouteOptimization.Time,
MapRouteRestrictions.Highways);
    if (result.Status == MapRouteFinderStatus.Success)
    {
        Duration.Text = result.Route.EstimatedDuration.ToString();
        Length.Text = result.Route.LengthInMeters.ToString();
        List<string> directions = new List<string>();
        foreach (MapRouteManeuver direction in result.Route.Legs[0].Maneuvers)
        {
            directions.Add(direction.InstructionText);
        }
    }
}
```

Using the **`MapRouteFinder`** class is one of the scenarios that requires a valid map token. Consequently, the first thing to do is to properly set the **`ServiceToken`** property of the **`MapService`** class.

As you can see, the **GetDrivingRouteAsync()** method does not support just setting a starting and an ending point but also some parameters in order to customize the route calculation. In the previous sample, we chose the kind of optimization by using the **MapRouteOptimization** enumerator (in this case, it will calculate the shortest route based on time) and specified that we wanted to avoid highways (by using the **MapRouteRestrictions** enumerator). The method will return a **MapRouteFinderResult** object, which offers many properties with the details of the calculated route. In the previous sample, we displayed to the user the duration (stored in the **EstimatedDuration** property) and the length of the route (store in the **LenghInMeters** property). One important property is called **Legs**, which is a collection of all of the routes that have been calculated. Every route offers a collection called **Menuevers**, which is the list of all of the instruction to go from point A to point B. In the sample, we store in a collection all of the values of the **InstructionText** property, which contains the textual indications (like, for example, “At the end of the street go left.”).

In addition, we can use the **MapRouteFinder** class not just to retrieve the information about the route but also to display it on the map by using the **MapRouteView** class and the **Routes** collection offered by the **MapControl** class. Let’s take a look at the following sample:

```
private async void OnShowRouteClicked(object sender, RoutedEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";

    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 9.0800;
    Geopoint startPoint = new Geopoint(position1);

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.4608;
    position2.Longitude = 9.1763;
    Geopoint endPoint = new Geopoint(position2);

    MapRouteFinderResult result = await
MapRouteFinder.GetDrivingRouteAsync(startPoint, endPoint, MapRouteOptimization.Time,
MapRouteRestrictions.Highways);
    if (result.Status == MapRouteFinderStatus.Success)
    {
        MapRouteView routeView = new MapRouteView(result.Route);
        routeView.RouteColor = Colors.Red;
        routeView.OutlineColor = Colors.Black;
        MyMap.Routes.Add(routeView);
    }
}
```

The **MapRouteView** class takes care of encapsulating the **Route** property of the **MapRouteFinderResult** class. In addition, we can use it to customize the visual layout of the route by setting the route color (**RouteColor**) and the color of the border applied to the route (**OutlineColor**). In the end, you just need to add the newly created **MapRouteView** object to the **Routes** collection of the **MapControl** object you have placed in the page.

Working with Coordinates and Addresses

Up until now, we have always used the **MapControl** object, with positions expressed with geographic coordinates (such as latitude and longitude). However, in many scenarios, you may need to work with information that is easier to understand for the user, such as a civic address. The Windows Runtime offers as set of APIs to perform geocoding (which means converting an address in coordinates) and reverse geocoding (which means converting a set of coordinates into an address) operations.

Geocoding is performed by using the **MapLocationFinder** class, which offers a method called **FindLocationsAsync()**, which accepts as parameter the name of the location we want to find, like in the following sample:

```
private async void OnGeocodeClicked(object sender, RoutedEventArgs e)
{
    string address = "Piazza Duomo, Milan";
    MapLocationFinderResult result = await
MapLocationFinder.FindLocationsAsync(address,
    null);
    if (result.Status == MapLocationFinderStatus.Success)
    {
        Geopoint position = result.Locations.FirstOrDefault().Point;
        await MyMap.TrySetViewAsync(position, 12);
    }
}
```

Optionally, you can also pass a second parameter to the **FindLocationsAsync()** method, which is a **Geopoint** object with the coordinates of the approximate area where the location is placed. If you do not have this information, you can simply pass null as value, as we did in the previous sample. The method returns a **MapLocationFinderResult** object, which contains two important pieces of information. The first one is **Status**, which informs the developer whether or not the operation has successfully completed. In the previous sample, we move on with the rest of the code only if we get a **Success** value. The other piece of information is **Locations**, which is a list of all of the locations that have been found for the specified search keyword. For each location, we have access to a property called **Point**, with the location's coordinate. In the previous sample, we take the coordinates of the first location and we display it on the map by calling the **TrySetViewAsync()** method.

Also, the opposite operation (reverse geocoding) is performed by using the **MapLocationFinder** class. The difference is, in this case, we are going to use the **FindLocationsAtAsync()** method which requires, as parameter, a **Geopoint** object with the coordinates of the place. Also, in this situation, we will get in return a **MapLocationFinderResult** object, which offers a **Locations** property with all of the places that match the given coordinates. The difference is, this time, instead of using the **Point** property (since we already know the coordinates), we get access to the **Address** one, like in the following sample:

```
private async void OnReverseGeocodeClicked(object sender, RoutedEventArgs e)
{
    Geolocator locator = new Geolocator();
    Geoposition geoposition = await locator.GetGeopositionAsync();


    MapLocationFinderResult result = await
MapLocationFinder.FindLocationsAtAsync(geoposition.Coordinate.Point);
    if (result.Status == MapLocationFinderStatus.Success)
    {
        MapAddress address = result.Locations.FirstOrDefault().Address;
        string fullAddress = string.Format("{0}, {1}", address.Street, address.Town);
        MessageDialog dialog = new MessageDialog(fullAddress);
        await dialog.ShowAsync();
    }
}
```

In this sample, we retrieve the user's position and we display, with a pop-up message, the corresponding address, retrieved by combining the values of the **Street** and **Town** properties of the **Address** class.

The Bing Maps Control in Windows 8.1

Unlike Windows Phone, Windows does not offer a built-in control to manage maps. Microsoft released the Bing Maps control as a separate Visual Studio extension, which you can download from the MSDN documentation here. The control is developed using C++. Consequently, as we have seen for SQLite in Chapter 6, you can't compile the project by using the standard "Any CPU" configuration. You will have to prepare a different package for each CPU architecture (i.e., x86, x64, and ARM). After you have installed the extension, you will find a new option in the Reference Manager window, which is displayed when you right-click your project in Visual Studio and you choose Add reference. In the section called **Extensions**, you will find a library called **Bing Maps for C#, C++ or Visual Basic**; enable it to be able to use the Bing Maps control in your app.

As previously mentioned, the second step is to change the build configuration, otherwise the project will not compile. Open the **Configuration Manager** window (which is available in the Visual Studio's Build menu) and, in the drop-down **Active solution platform** menu, replace the default value ("Any CPU") with x86 (if you are testing your app on a computer) or ARM (if you are testing your app on an ARM device such as the Surface or the Lumia 2520).

Now you are ready to add the control to your page. It is simply called **Map** and it is included in the **Bing.Maps** namespace:

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

    <Grid>
        <maps:Map x:Name="MyMap" />
    </Grid>

</Page>
```

How to Publish an App that Uses the Bing Maps Control

In the same way we needed to retrieve an access token to properly use the map control in Windows Phone, we now need to perform the same operation in order to use the Bing Maps control in a Windows Store app. The difference, in this case, the token is provided by the Bing Developer Portal. The URL where to start the procedure can be found here.

Inside this page, you will find the option **Get the basic key**. By following the wizard and providing some info about your app (such as its name), you will get a token. This token needs to be specified in the **Credentials** property of the control, like in the following sample:

```
<maps:Map x:Name="MyMap"
Credentials="AivHKU3GuTYYQlkA7JIZg6L48I0C6zEKMv1ntcRk5pR6EozLupMRRaLGxyqVC" />
```

Interacting with the Map

The basic properties to interact with the map are **Center** and **ZoomLevel**, which define the position where you want to center the map and the zoom level you want to apply. Similar to what we saw with the Windows Phone control, we can use a method called **SetView()** to automatically center the map on a specific position with a defined zoom:

```
private async void OnFindPositionClicked(object sender, RoutedEventArgs e)
{
    Geolocator locator = new Geolocator();
    Geoposition position = await locator.GetGeopositionAsync();
    Location location = new Location(position.Coordinate.Point.Position.Latitude,
position.Coordinate.Point.Position.Longitude);
    MyMap.SetView(location, 15);
}
```

Unfortunately, as you can see, in this case we need to work with a different class that identifies a position than the one used by the **Geolocator** class. In fact, the **GetGeopositionAsync()** method returns a **Geoposition** object while the **SetView()** method requires a **Location** object. Consequently, we need to manually convert from one type to the other before we can interact with the map.

Displaying a Pushpin

The **Map** control provides, in the **Bing.Maps** namespace, a specific class in which to manage pushpins called **Pushpin**. However, you can't customize the icon by providing another image, Instead, you will need to customize the XAML template. You have the chance anyway to customize the text, which will be displayed in overlay over the pushpin. The following sample shows how to display a pushpin over the map:

```
private void OnFindPositionClicked(object sender, RoutedEventArgs e)
{
    Location position1 = new Location(45.8080, 009.0800);
    Pushpin pin = new Pushpin();
    pin.Text = "Pub";
    pin.Tapped += pin_Tapped;

    MapLayer.SetPosition(pin, position1);
    MyMap.Children.Add(pin);
}


async void pin_Tapped(object sender, TappedRoutedEventArgs e)
{
    Pushpin pin = sender as Pushpin;
    MessageDialog dialog = new MessageDialog(pin.Text);
    await dialog.ShowAsync();
}
```

We defined a **Pushpin** object by setting the **Text** property with the name to display. In addition, we subscribed to the **Tapped** event, which is triggered when the user taps on it. In the previous sample, we displayed to the user (using a pop-up message) the name of the selected pushpin. We are able to do it thanks to the parameter called **sender**, which is the **Pushpin** object that triggered the event.

The approach to add the pushpin to the map is different than the one we saw with Windows Phone. We need to call the **SetPosition()** method of the **MapLayer** class, passing as parameters the **Pushpin** object and the **Location** object that identifies the position. However, this line of code just takes care of setting the pushpin's position. To effectively display it on the map, we will need to add it to the **Children** collection of the **Map** control, which contains all of the objects that are displayed in overlay.

However, we can also use another approach that is similar to the one we saw with the Windows Phone control. We can create a collection of pushpins and then define a template, which will be used to render each pushpin, in the same way we do with a **ListView** or **GridView** control to display a list of items. To achieve this goal, we need to use the **MapItemsControl** object, which offers a property called **ItemTemplate** to customize the look and feel of the pushpin:

```
<maps:Map x:Name="MyMap">
    <maps:MapItemsControl x:Name="Pushpins">
        <maps:MapItemsControl.ItemTemplate>
            <DataTemplate>
                <maps:Pushpin Tapped="pin_Tapped" Text="{Binding Name}">
                    <maps:MapLayer.Position>
                        <maps:Location Latitude="{Binding Latitude}"
                                       Longitude="{Binding Longitude}" />
                    </maps:MapLayer.Position>
                </maps:Pushpin>
            </DataTemplate>
        </maps:MapItemsControl.ItemTemplate>
    </maps:MapItemsControl>
</maps:Map>
```

To define the pushpin's position, we use a special attached property called **Location**, offered by the **MapLayer** class. This property has two attributes, **Latitude** and **Longitude**, which define the pushpin's coordinates. In the same way we did with Windows Phone, we need a custom class that defines a pushpin and which we will use to populate the **MapItemsControl** object, like in the following sample:

```
public class MyPushpin
{
    public string Name { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

Now you can simply create your collection of **MyPushpin** objects and assign it to the **ItemsSource** property of the **MapItemsControl** object. The following sample shows how to display two pushpins on the map by using the template we previously defined:

```
private void OnAddPushpinsClicked(object sender, RoutedEventArgs e)
{
    List<MyPushpin> pushpins = new List<MyPushpin>();

    MyPushpin pin1 = new MyPushpin
    {
        Name = "Bar",
        Latitude = 45.8080,
        Longitude = 9.0800
```

```
    };

    MyPushpin pin2 = new MyPushpin
    {
        Name = "Banca",
        Latitude = 45.8052,
        Longitude = 9.0825
    };

    pushpins.Add(pin1);
    pushpins.Add(pin2);

    Pushpins.ItemsSource = pushpins;
}
```

Working with Coordinates and Addresses

Also, the Bing Maps software developer kit (SDK) offers a set of APIs that can be used to perform geocoding and reverse geocoding operations. In both cases, the class we are going to use is called **SearchManager**, whose instance is directly exposed by the **Map** control. The following code shows how to perform geocoding, which is retrieving the coordinates of a given address:

```
private async void OnGeocodeClicked(object sender, RoutedEventArgs e)
{
    SearchManager manager = MyMap.SearchManager;
    GeocodeRequestOptions geocodeRequest = new GeocodeRequestOptions("Piazza Duomo,
Milan");
    LocationDataResponse response = await manager.GeocodeAsync(geocodeRequest);
    if (!response.HasError)
    {
        Location location = response.LocationData.FirstOrDefault().Location;
        MyMap.SetView(location, 15);
    }
}
```

The geocoding operation is represented by the **GeocodeRequestOptions** class, which requires an address to resolve as the parameter when you create a new instance. Then, after getting a reference to the **SearchManager** class from the **Map** control, we invoke the **GeocodeAsync()** method, passing as parameter the request we have just defined. The response is a **LocationDataResponse**, which contains a collection called **LocationData** with all of the places that have been found. Specifically, we can use the **Location** property to get the coordinates. The previous sample takes the coordinates of the first location and displays it on the map. The **LocationDataResponse** object also offers a property called **HasError**. It is set to **true** if something went wrong so it is important to check its value before performing additional operations.

The reverse geocoding operation (which means converting a set of coordinates into a civic address) is performed in a similar way. The main difference is, in this case, we are going to use a **ReverseGeocodeRequestOptions** object, which requires as parameter the **Location** object with the coordinates of the place we want to resolve:

```
private async void OnReverseGeocodeClicked(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    Geoposition position = await geolocator.GetGeopositionAsync();
    Location location = new Location(position.Coordinate.Point.Position.Latitude,
position.
    Coordinate.Point.Position.Longitude);

    SearchManager manager = MyMap.SearchManager;
    ReverseGeocodeRequestOptions reverseGeocodeRequest = new
ReverseGeocodeRequestOptions
    (location);
    LocationDataResponse response = await
manager.ReverseGeocodeAsync(reverseGeocodeRequest);

    if (!response.HasError)
    {
        GeocodeAddress address = response.LocationData.FirstOrDefault().Address;
        MessageDialog dialog = new MessageDialog(address.FormattedAddress);
        await dialog.ShowAsync();
    }
}
```

The previous sample shows how to retrieve the address of the current's user position, obtained by using the Geolocator class. In this case, since we already know the place's coordinates, we can use the **FormattedAddress** property (which is included in every object that belongs to the **LocationData** collection) in order to get the civic address.

The Sensors

Most of the tablets and smartphones on the market include a set of sensors which can be used to detect if and how the device is moving in space. Many games use this approach in order to provide an alternative way to control the game (instead of relying on virtual joy pads that are displayed on the screen).

Windows and Windows Phone devices offer many sensors; each of them is represented by one of the classes that is included in the **Windows.Devices.Sensors** namespace. Here are the main ones:

- **Accelerometer**: Identified by the **Accelerometer** class; it measures the phone's position on the X, Y, and Z axes

- **Inclinometer**: Identified by the **`Inclinometer`** class; it measures roll, pitch, and yaw
- **Compass**: Identified by the **`Compass`** class; it measures the phone's position compared to the magnetic north
- **Magnetometer**: Identified by the **`Magnetometer`** class; it measures the magnetic field's intensity
- **Gyrometer**: Identified by the **`Gyrometer`** class; it measures the angular speed of the device

However, in most cases, what is important for the developer is to discover the position of the device with the best accuracy. For this purpose, the Windows Runtime offers a special class called **`OrientationSensor`**, which is able to combine all of the information retrieved by the available sensors to provide a set of optimized values. It does so by automatically excluding all of the "dirty" values that could ruin the user experience (UX). This class, in order to work properly, requires that the device includes, at least, an accelerometer and a compass. However, the best results are achieved if the device also includes a gyroscope. Consequently, if your app heavily relies on these sensors in order to work properly, it is better to properly configure them in the **Requirements** section of the Windows Phone manifest file. This way, if the user has a device that does not satisfy these requirements, he or she will not be able to install the app at all from the Store.

No matter which sensor we use, they all work in the same way. Every class offers a method called **`GetDefault()`**, which retrieves a reference to the sensors. It is important to always check that the value is not **`null`** before performing additional operations. In fact, the device where the app is running could not offer that sensor; consequently, any other interaction would return an exception. After you have a valid reference to the sensor, you can use two different approaches, which are similar to what we saw with the GPS tracking.

The first approach is to ask for a single reading by using the **`GetCurrentReading()`** method. The data type you will get in return can change from sensor to sensor. Typically, the name of the class matches the name of the sensors plus the **`Reading`** suffix (for example, the **`Accelerometer`** class returns an **`AccelerometerReading`** object). The following sample shows how to use the **`Accelerometer`** class to get a single reading of the current position:

```
private void OnStartAccelerometerClicked(object sender, RoutedEventArgs e)
{
    Accelerometer accelerometer = Accelerometer.GetDefault();
    if (accelerometer != null)
    {
        AccelerometerReading reading = accelerometer.GetCurrentReading();
        X.Text = reading.AccelerationX.ToString();
        Y.Text = reading.AccelerationY.ToString();
        Z.Text = reading.AccelerationZ.ToString();
    }
}
```

The position of the device on the three axes (stored in the **`AccelerationX`**, **`AccelerationY`**, and **`AccelerationZ`** properties of the reading object) are displayed on the page by using three **`TextBlock`** controls.

The other approach is to continuously monitor the sensors so that you will be notified every time the phone is moved into a new position. In this scenario, we can subscribe to an event, exposed by all of the sensor's classes, called **ReadingChanged**. The following sample shows how to subscribe to this event for the **Accelerometer** class so that we can continuously update the page with the current coordinates:

```
private void OnStartAccelerometerClicked(object sender, RoutedEventArgs e)
{
    Accelerometer accelerometer = Accelerometer.GetDefault();
    if (accelerometer != null)
    {
        accelerometer.ReadingChanged += accelerometer_ReadingChanged;
    }
}

void accelerometer_ReadingChanged(Accelerometer sender,
AccelerometerReadingChangedEventArgs args)
{
    Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        X.Text = args.Reading.AccelerationX.ToString();
        Y.Text = args.Reading.AccelerationY.ToString();
        Z.Text = args.Reading.AccelerationZ.ToString();
    });
}
```

Do notice that the **ReadingChanged** event is executed in a background thread so we need to use the **Dispatcher** to interact with the XAML controls placed in the page.

How to Test the Movement Sensors

Unfortunately, the Windows simulator does not offer any method to test the movement sensors. You will have to test your apps on a real tablet. Windows Phone offers a section in the emulator's additional tools called **Accelerometer**. Unfortunately, it works only with the **Accelometer** class so it can't be used to test other sensors. But using this tool is easy: by dragging and dropping the three-dimensional (3-D) render of the device, you will send the X, Y, and Z coordinates to the emulator, which will be detected by the **Accelerometer** class.

In addition, the emulator's tools also offer a section called **Sensors** containing a list of all of the sensors that a Windows Phone device can have. You will be able to enable or disable each of them. This way, you can test that your code properly manages the scenarios in which one or more of the sensors you use in your app are not available.

Chapter 4 Contracts and Extensions

Contracts

Contracts are the mechanism offered by the Windows Runtime to connect two third-party apps so that they can exchange data. There are two kind of contracts:

- Source contracts, which are implemented by apps that want to share some data
- Target contracts, which are implemented by apps that want to receive shared data

The most important feature about contracts is that they are completely transparent to the developer. The source app does not have to know anything about the target that will receive the data. In the same way, the target app is able to receive the shared data without knowing anything about the source app.

The Sharing Contract

The sharing contract is used to share any kind of data (e.g., text, images, files, etc.) between two apps. The following image shows how the sharing contract works:

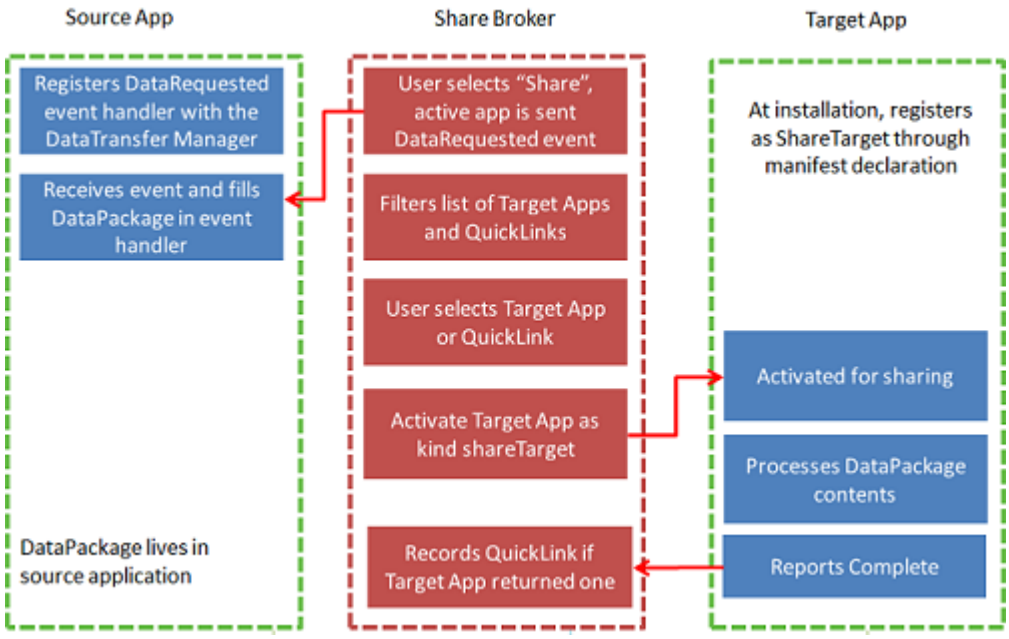


Figure 9: The sharing contract's architecture

The source app registers itself to be notified every time the user chooses to share some data. Inside this event handler, we are going to define the content to share. Then, it will be time for the OS to take care of the next step of the operation. It will display a list containing all of the apps that have registered the share target contract and which are able to support the data type we are sharing. The last step is to perform the real sharing operation, which is performed by the target app. It will receive the shared data and it will take care of processing it. In the rest of the chapter, we will look, in detail, at how to implement both the source and target sharing contracts.

Activating the Share

When it comes to sharing some content and displaying the list of target apps, there is an important difference between Windows and Windows Phone. Windows provides a built-in way to trigger the sharing operation, which is the Share button that is placed inside the Charms bar. On the other hand, Windows Phone does not offer a Charms bar so it is up to us to define the best way to trigger the sharing operation (for example, by placing a specific button in the app bar). Starting the sharing operation is easy. You just have to call the **`ShowShareUI()`** method of the **`DataTransferManager`** class (that is part of the **`Windows.ApplicationModel.DataTransfer`** namespace), which is the main class we are going to use to interact with the sharing contract.

```
private void OnShareItemClicked(object sender, RoutedEventArgs e)
{
    DataTransferManager.ShowShareUI();
}
```

However, this code also works on Windows. Invoking the **`ShowShareUI()`** method will achieve the same result as pressing the Share button in the Charms bar.

Sharing Content

The sharing operation is performed at page level since every page can share different kinds of content. For this reason, each page has its own instance of the **`DataTransferManager`** class, which is retrieved by using the **`GetForCurrentView()`** method. Since, in Windows, the sharing operation can be triggered outside the app (by using the Charm bar), the **`DataTransferManager`** class offers a specific event called **`DataRequest`**, which is triggered every time a sharing action is started. It is inside this event handler that we are going to define the content to share, like in the following sample:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        DataTransferManager transferManager =
DataTransferManager.GetForCurrentView();
        transferManager.DataRequested += transferManager_DataRequested;
```

```
    }

    private void transferManager_DataRequested(DataTransferManager sender,
DataRequestedEventArgs args)
    {
        //manage the sharing operation
    }
}
```

The event handler contains a parameter, whose type is **DataRequestedEventArgs**. It is the heart of our sharing operation since it allows us to define all of the properties of the request, thanks to the **Request.Data** object. Regardless of the type of content we want to share, there are two properties inside the **Properties** object that are always required: **Title** and **Description**.

They are especially useful on Windows since this information is displayed as the header of the sharing panel. Since the user can trigger the sharing action outside of the app, it is important to highlight to the user which kind of content he or she is going to share:

```
private void transferManager_DataRequested(DataTransferManager sender,
DataRequestedEventArgs
    args)
{
    args.Request.Data.Properties.Title = "Sharing demo";
    args.Request.Data.Properties.Description = "This is a sharing demo";
}
```

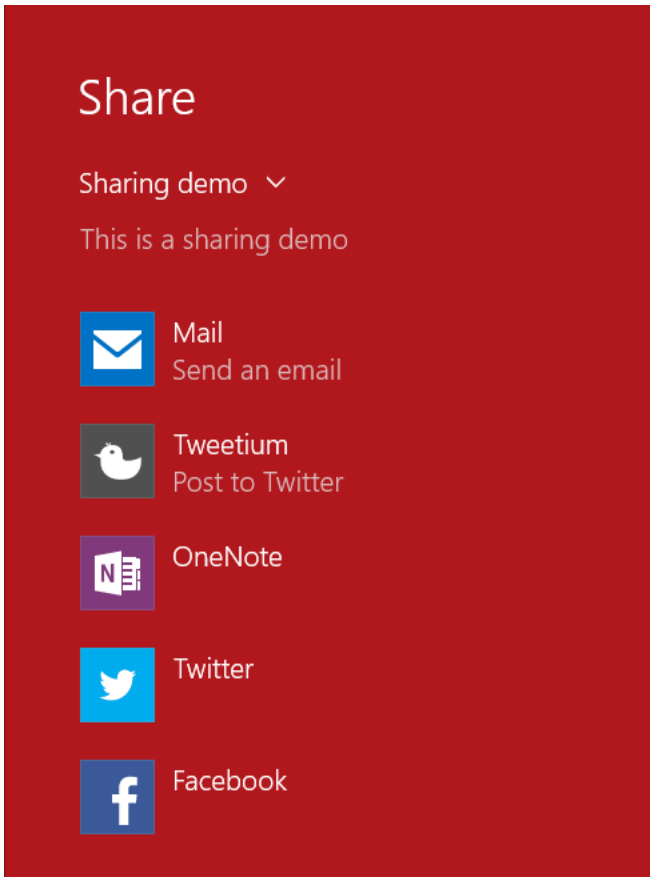


Figure 10: The sharing panel in Windows

After you have defined title and description, it is time to define the real content. The **`Request.Data`** object offers many methods that begin with the Set prefix. Each of them represents one of the data types you can share. Let's take a look at some samples of the most common ones.

Sharing a Text

Text can be shared with the **`SetText()`** method, which simply accepts the string to share, like in the following sample:

```
private void transferManager_DataRequested(DataTransferManager sender,
DataRequestedEventArgs
args)
{
    args.Request.Data.Properties.Title = "Sharing demo";
    args.Request.Data.Properties.Description = "This is a sharing demo";
    args.Request.Data.SetText("This is a demo text");
}
```

Sharing a Link

A link is simply a string. However, some apps are able to manage it in a special way. For example, the People app in Windows 8.1 is able to automatically generate a preview of the website before sharing the link on a social network. The method to use to share a link is called **`SetWebLink()`** and it simply requires, as parameter, a Uniform Resource Identifier (**`Uri`**) object with the address, like in the following sample:

```
void transferManager_DataRequested(DataTransferManager sender,
DataRequestedEventArgs args)
{
    args.Request.Data.Properties.Title = "Sharing demo";
    args.Request.Data.Properties.Description = "This is a link to share";
    Uri url = new Uri("http://wp.qmatteoq.com", UriKind.Absolute);
    args.Request.Data.SetWebLink(url);
}
```

Sharing a File

Sharing a file is one of the most frequently used features since all of the complex data (such as images) are treated as files by the OS.

Note: The Request.Data object offers a specific method for sharing images, which is called SetBitmap(). However, it is useless in Windows Phone since all of the native apps do not manage it; instead, they treat them as regular files. Consequently, we will not discuss the use of this method in this book.

However, before showing you how to share a file in an app, there is an important concept to introduce: asynchronous operations. If you have read Chapter 5 (which is about managing the local storage). you will know that all of the storage APIs are asynchronous and based on the "async and await" pattern. Consequently, in a sharing operation, the OS will not be able to determine when the files are ready and when the sharing contract can be activated. If you remember the app's life cycle as explained in Chapter 4, a similar situation occurs when you need to save the app's state during the suspension event.

To avoid this issue, we are going to use the same mechanism, which is using a deferral object. In this case, its type is **`DataRequestDeferral`** and it is provided by the **`GetDeferral()`** method offered by the **`Request`** object. After getting a reference to this object, you will be able to perform any asynchronous operation. You will just have to remember to call the **`Complete()`** method once the operation is done. Let's take a look at how to use this approach in combination with the method provided to share one or more files, called **`SetStorageItems()`**:

```
async void transferManager_DataRequested(DataTransferManager sender,
DataRequestedEventArgs args)
{
    args.Request.Data.Properties.Title = "Sharing demo";
    args.Request.Data.Properties.Description = "This is a file sharing demo";
    DataRequestDeferral deferral = args.Request.GetDeferral();
```

```
    List<StorageFile> files = new List<StorageFile>();
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("text.txt");
    files.Add(file);
    args.Request.Data.SetStorageItems(files);
    deferral.Complete();
}
```

The first thing we do is get a reference to the **`DataRequestDeferral`** object by using the **`GetDeferral()`** method offered by the **`Request`** object (which is one of the properties of the method's parameters). Then, we retrieve a reference to the file (or the files) we want to share. In the previous file, we want to share a text file called **text.txt** stored in the local storage. To perform the sharing, we call the **`SetStorageItems()`** method of the **`Request.Data`** object, passing as parameter the list of files to share. This method always requires a collection so we always need to create a list (even if, as in the previous sample, we are going to share just one file). In the end, we complete the asynchronous operation by calling the **`Complete()`** method exposed by the **`DataRequestDeferral`** object.

Receiving Content from Another App

Receiving some content from another app requires you to subscribe to one of the contracts offered by the OS. Consequently, before writing some code, we will need to register the contract in the manifest file inside the section called **Declarations**. Inside the **Available Declarations**' drop-down menu, you will find a list of all of the contracts and extensions available. The one that is required for our scenario is called **Share Target** and it requires us to set two pieces of information:

- The data type (or types) we want to support, which can be:
  - Text
  - URI
  - HTML
  - StorageItems (for files)
- In case we want to be able to receive files, we need to also specify which types we are going to support by setting up the section titled **Supported file types**. (We will need to press the **Add new** button for each type we want to support and to specify the file's extension. Otherwise, we can simply enable the **Supports any file type** option to be able to receive any file's type)

The next step is to manage the sharing operation. Typically, when our app is chosen as a share target, a specific page of the app is opened. It will provide a preview of the content that will be shared and will allow the user to confirm or cancel the operation. Consequently, you will not find any sample about how to define the sharing page's layout. It is up to you and to the content you want to be able to share to define the layout that fits best your needs. For example, if your app allows the user to share an image, the sharing page could display a thumbnail. Or, if your app can receive a link, it could display a preview of the website.

When a target app is activated by a share contract, it does not follow the traditional app's life cycle. Consequently, the **OnLaunched()** method of the **App** class is not triggered as it is when the app is opened by using the main tile. In a sharing scenario, the app is activated by invoking the **OnShareTargetActivated()** method which can be declared in the **App** class, like in the following sample:

```
protected override void OnShareTargetActivated(ShareTargetActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
        rootFrame = new Frame();
    }
    rootFrame.Navigate(typeof(SharePage), args.ShareOperation);
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
}
```

This method simply takes care of redirecting the user to the page we prepared to manage the sharing operation (in the previous sample, it is called **SharePage**). By using the **Navigate()** method of the **Frame** class, we also add, as navigation parameter, the **ShareOperation** property that is included in the **OnShareTargetActivated()** method's parameters. This property contains all of the information about the sharing operation (such as the data that has been shared). Consequently, we will need to use it in the sharing page in order to perform all of the operations. The only thing to highlight in the previous code is that, before performing the navigation, we check whether or not the **Frame** already exists, otherwise we create it. This way, we can properly manage the app's life cycle. In fact, the app could also be activated when it was suspended in memory.

Let's take a look, in detail, at how to receive the most common data types from another app.

Receiving a Text

No matter what data type we want to receive, we will need to manage the **OnNavigatedTo()** method (as we learned about in Chapter 4 about navigation). Thanks to this method, we are able to retrieve the parameter that has been passed by the previous page. This parameter contains a **ShareOperation** object, which offers a property called **Data**. It contains all of the information about the content that has been shared. Let's take a look at the following sample:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    ShareOperation shareOperation = e.Parameter as ShareOperation;
    if (shareOperation.Data.Contains(StandardDataFormats.Text))
    {
        string text = await shareOperation.Data.GetTextAsync();
        MessageDialog dialog = new MessageDialog(text);
        await dialog.ShowAsync();
    }
```

```
}
```

The first operation is to understand what kind of content has been shared. In fact, a target app could implement many sharing contracts. Consequently, we need to detect which data type we have received so that we can properly manage it. This operation is performed by checking the content of the **Data** property. The Windows Runtime offers an enumerator called **StandardDataFormats**, which offers a value for each supported data type. By using the **Contains()** method, we are able to check if the **Data** property contains a specific data type. In the previous sample, we check for the **Text** data type. After we have identified the proper data type, the **Data** object offers many methods (which start with the **Get** prefix) that are able to parse the content and convert it into the proper type. In the previous sample, since we are working with text, we call the **GetTextAsync()** method, which simply returns the string that has been shared by the source app. In the sample, we display it to the user using a pop-up message.

However, the sharing operation is not completed. This way, we just showed a preview of the content received by the other app to the user. The last and most important step is to interact with the shared content. The type of interaction is not fixed but, rather, it depends upon the app's purpose. For example, a Twitter app will "allow to post" the received content on the user's timeline whereas a Facebook app will "allow to post" the received text as a status on the social network. Consequently, you will not find any specific sample on how to perform the sharing operation in this chapter. However, there is one step that it is always required regardless of the way you perform the sharing operation. That step is notifying the OS that the sharing operation is completed and that the target app can be closed so that control can be returned to the source app. This notification is performed by calling the **ReportCompleted()** method on the **ShareOperation** object we received as parameter in the **OnNavigatedTo()** event. Here is a complete sample:

```
public sealed partial class SharePage : Page
{
    private ShareOperation shareOperation;

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

    protected override async void OnNavigatedTo(NavigationEventArgs e)
    {
        shareOperation = e.Parameter as ShareOperation;
        if (shareOperation.Data.Contains(StandardDataFormats.Text))
        {
            string text = await shareOperation.Data.GetTextAsync();
            MessageDialog dialog = new MessageDialog(text);
            await dialog.ShowAsync();
        }
    }
```

```
    private void OnShareClicked(object sender, RoutedEventArgs e)
    {
        //perform the sharing operation
        shareOperation.ReportCompleted();
    }
}
```

You will notice that we have defined the **ShareOperation** object at class level. This way, we can interact with it both in the **OnNavigatedTo()** method (to retrieve the shared content) and in the method that performs the sharing (in this case, it is an event handler that is triggered when the user presses a button in the page).

Receiving a Link

Now that we have seen how to receive a text, it will be easy to understand how to receive all of the other data types. In fact, the base approach is always the same. We subscribe to the **OnNavigatedTo()** method to receive the **ShareOperation** object, which contains the shared content. Then, after the sharing operation is completed, we call the **ReportedCompleted()** method. The only difference is that, for each data type, we will need to retrieve the content in a different way.

When it comes to working with links, we will have to check if the **Data** property of the **ShareOperation** object contains a **WebLink** content. In this case, we can retrieve it by using the **GetWebLinkAsync()** method, like in the following sample:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    ShareOperation shareOperation = e.Parameter as ShareOperation;
    if (shareOperation.Data.Contains(StandardDataFormats.WebLink))
    {
        Uri uri = await shareOperation.Data.GetWebLinkAsync();
        MessageDialog dialog = new MessageDialog(uri.AbsoluteUri);
        await dialog.ShowAsync();
    }
}
```

Receiving a File

When you receive a file from a source app, the **Data** object contains a value of type **StorageItems**. In this scenario, we can retrieve the list of shared files by using the **GetStorageItemsAsync()**, which returns a read-only collection. If you want to perform additional operations, it is necessary to copy the files in the app's local storage, like in the following sample:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    ShareOperation shareOperation = e.Parameter as ShareOperation;
    if (shareOperation.Data.Contains(StandardDataFormats.StorageItems))
    {
        var list = await shareOperation.Data.GetStorageItemsAsync();
        foreach (IStorageItem item in list)
        {
            StorageFile file = item as StorageFile;
            await file.CopyAsync(ApplicationData.Current.LocalFolder);
        }
    }
}
```

Managing the Sharing Page in a Windows App

I have previously mentioned that, in this chapter, you will not find any samples on how to define the sharing page since there is not a specific guideline. Everything depends on the content you want to be able to receive and how you want to present it to the user. However, there is an important difference between Windows and Windows Phone. On a smartphone, the sharing page is a traditional app's page, which is directly opened when the app is activated by a sharing contract. But on a computer or tablet, the OS is able to run multiple apps at the same time. Consequently, when a target app is activated to receive shared content, it is not opened in full screen mode like it is when it is launched from the main tile. Rather, it is opened in a special panel that it is pulled from the right margin of the screen.

This panel is, indeed, a traditional page which you can manage just like any other page. It is composed of an XAML file (with the layout) and a code behind file (with the sharing logic). However, you can specify in Visual Studio that the purpose of the page is to manage the sharing. This way, the visual designer will automatically display the area that will be displayed when the page is opened due to a sharing operation, and you will be able to make sure that the layout you have defined fits the available space.

To achieve this goal, it is enough to add the **`d:ExtensionType`** attribute to the Page definition in XAML and set its value to **`ShareTarget`**, like in the following sample:

```
<Page
    x:Class="ContractsTarget.SharePage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:ExtensionType="ShareTarget">
```

```
</Page>
```

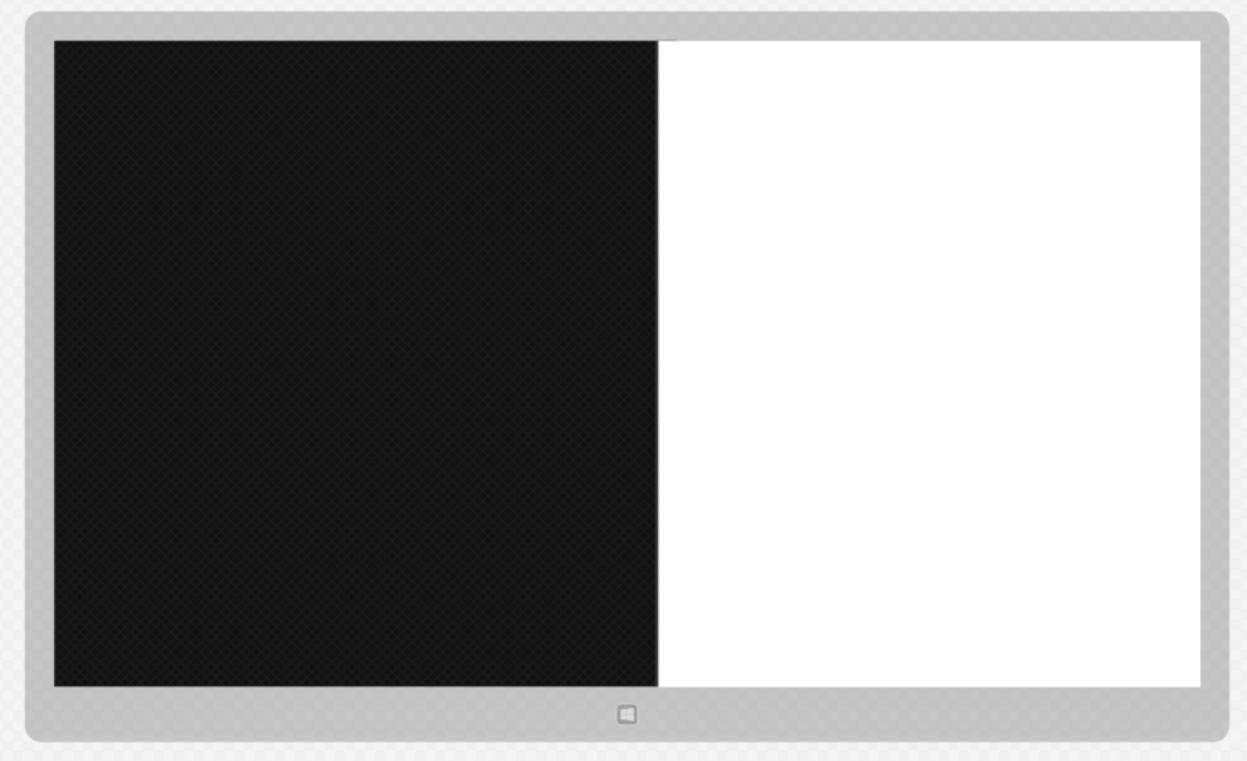


Figure 11: The Visual Studio designer highlights the area displayed when the page is opened due to a sharing contract

The FileOpenPicker Contract

In Chapter 5, we learned how to use the **`FileOpenPicker`** class, which we can use to import one or more files from the device into our app. However, the Windows Runtime also offers a contract connected to this class. This way, our app can behave as a picker source so the user will be able to choose a file not only from one of the standard device libraries (such as music or pictures) but also from our app.

To enable this feature, we need to add the specific contract in the manifest file. It is available in the **Available declarations** drop-down menu in the **Declarations** section and it is called **File Open Picker**. It is important to set which file types we want to support by declaring their extensions. However, we can also enable the **Supports any file type** option to support any file regardless of the extension. The next step is to add a page to our project, which will be invoked when the contract is activated. This page will display to the user the files created by the app so that he or she will be able to pick one of them. Consequently, I will not share any specific examples about how to define the layout of this page since it depends upon the kind of data you want to expose. For example, if your app creates text files, you could display them by using a **`ListView`** control. If your app creates images, you could display them with a thumbnail by using a **`GridView`** control.

As with any other contract, the app's activation is managed by the **App** class. Specifically, in this case, it is done so by a method called **OnFileOpenPickerActivated()**.

```
protected override void OnFileOpenPickerActivated(FileOpenPickerActivatedEventArgs
args)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
        rootFrame = new Frame();
    }
    rootFrame.Navigate(typeof(OpenPickerPage), args.FileOpenPickerUI);
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
}
```

The approach is similar to the one we have seen for the sharing contract. By using the **Navigate()** method of the **Frame** class, we redirect the user to the page we have prepared to manage the file picker. As navigation parameters, we pass to the page a property called **FileOpenPickerUI** which we are going to use to interact with the picker. Consequently, the first thing we will have do in the picker page is to retrieve, in the **OnNavigatedTo()** method, a reference to the **FileOpenPickerUI** object and store it in a variable defined at class level, like in the following sample:

```
public sealed partial class OpenPickerPage : Page
{
    private FileOpenPickerUI fileOpenPickerUI;
    public OpenPickerPage()
    {
        this.InitializeComponent();
    }
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        if (e.Parameter != null)
        {
            fileOpenPickerUI = e.Parameter as FileOpenPickerUI;
        }
    }
}
```

As previously mentioned, the page will display the list of files that the user can share so that he or she can pick one of them. Every time the user chooses one of the files, we need to interact with the **FileOpenPickerUI** object to communicate to the app that invoked the **FileOpenPicker** object which files have been selected. We achieve this goal by calling the **AddFile()** method and passing as parameter the selected file, represented by a **StorageFile** object. The following sample shows a scenario with a **ListView** control that displays the list of available files. When the user taps on one of them, the **SelectionChanged** event is triggered. We use it to add the selected file to the **FileOpenPickerUI** object.

```
private async void OnFilesSelected(object sender, SelectionChangedEventArgs e)
{
    string fileName = Files.SelectedItem.ToString();
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync(fileName);
    fileOpenPickerUI.AddFile(fileName, file);
}
```

As you can see, the **AddFile()** method also requires a unique identifier for the file other than the reference to a **StorageFile** object.

What happens next is based upon how the **FileOpenPicker** class was used by the source app. If the picker has been invoked to select just one file, by calling the **AddFile()** method the control will be immediately returned to the original app, which will receive the **StorageFile** object we have passed. But, if the picker has been invoked to allow multiple selections, your app will keep the control so that the user can tap on more items in the list and add more files to the collection. The OS will automatically add a button to your page to complete the operation. Only after the user has pressed it will the control be returned to the original app (which will receive a collection with all of the selected files).

In case you are dealing with multiple selections, the **FileOpenPickerUI** class also offers a way to perform additional operations. For example, the user could decide to remove from the list a file that he or she has previously selected from the list. In this case, it is enough to call the **Remove()** method by passing as parameter the file identifier. The following sample shows a more accurate way to manage the **SelectionChanged** event of the **ListView** control. Before adding the file to the **FileOpenPickerUI** object, we check to see if it is already on the list by using the **ContainsFile()** method. If the answer is yes, instead of adding it, we remove it. Otherwise, we simply add it as we did in the previous sample:

```
private async void OnFilesSelected(object sender, SelectionChangedEventArgs e)
{
    string fileName = Files.SelectedItem.ToString();
    if (fileOpenPickerUI.ContainsFile(fileName))
    {
        fileOpenPickerUI.RemoveFile(fileName);
    }
    else
    {
        StorageFile file = await
```

```
ApplicationData.Current.LocalFolder.GetFileAsync(fileName);
        fileOpenPickerUI.AddFile(fileName, file);
    }
}
```

Supporting Search

Search is one of the topics that has deeply changed in Windows 8.1. In fact, Windows 8 offered a specific search contract which Windows Store apps were able to implement. This way, we were able to include results related to our app's data in the native search feature, which is activated by pressing the Search button in the Charms bar. Consequently, the guidelines stated that the app should always rely on this contract to provide the search feature. You could not add a search area in your app; instead, the user had to use the Charms bar to start a search within the app.

However, many users were not comfortable using the Charms bar, especially on traditional computers. Since, by default, the Charms bar is hidden, many users thought that many apps did not implement a search feature. Consequently, Microsoft decided to change the search guidelines in order to improve discoverability. Now, Windows Store apps are able to include their own search area. For this reason, the search contract is now deprecated. You will still find it in the list of available contracts but it has been maintained only for backward compatibility with existing 8.0 apps that are implementing it.

It is important to highlight that this approach has changed only when it comes to Windows Store apps for Windows. Windows Phone has never supported the concept of "universal search" and does not provide a Charms bar. Consequently, developers have always been able to support search in the way they prefer. In fact, you will not find the search contract in the manifest file of a Windows Phone project.

The Windows Runtime includes a set of controls that can be useful to implement search in a Windows Store app. However, you will not find a universal control but, rather, two different controls for Windows and Windows Phone.

Adding Search in Windows

Windows has added a new control that can be used to implement search called **`SearchBox`**. It behaves like a standard **`TextBox`** control but it adds also a set of useful features, such as:

- It includes a magnifier icon, which can be tapped to trigger the search
- It supports auto suggestion; you can provide a list of keywords, which are prompted to the user as suggestions while it is typing
- It keeps track of the search history

The following sample shows how to include the **SearchBox** control in your page:

```
<SearchBox PlaceholderText="Search a person"
           SuggestionsRequested="OnSuggestionRequested"
           SearchHistoryEnabled="True"
           FocusOnKeyboardInput="True"
           QuerySubmitted="OnQuerySubmittedClicked" />
```

You will notice three important properties offered by the control:

- **PlaceholderText**: It is a text that is displayed in the box until the user starts typing some text; it is a way to remind the user which kind of data he or she is able to search
- **SearchHistoryEnabled**: It can be used to enable the search history; if it is set to **True**, every time the user performs a search the keyword will be saved and displayed in a drop-down list to the user (when he or she sets the focus on the box)
- **FocusOnKeyboardInput**: It is useful when the app is used on a computer with a keyboard; when it is set to **True**, the focus is automatically assigned to the control as soon as the user starts typing something on the keyboard

The main event offered by the control is called **QuerySubmitted**. It is invoked when the user starts a search by pressing the Enter button on the keyboard or by tapping on the magnifier icon. The event handler offers a parameter, which contains, inside a property called **QueryText**, the search keyword. The following sample redirects the user to a results page, passing the keyword as navigation parameter:

```
private void OnQuerySubmittedClicked(SearchBox sender,
SearchBoxQuerySubmittedEventArgs args)
{
    Frame.Navigate(typeof(SearchResultsPage), args.QueryText);
}
```

With the usual navigation approach, we will be able to retrieve the keyword as **navigation** parameter in the destination page (in the previous sample, it is called **SearchResultsPage**). The purpose of this page is to query your data with the specified keyword and to display the results to the user. You will not find any specific sample in this chapter since the type of query to perform and the layout of the result page depends upon the data that your app is able to manipulate and how it is structured. For example, this page could trigger a search using a REST service or perform a search in a local database.

As we saw when we added the **SearchBox** control in the page, we can subscribe to another event called **SuggestionRequested**. It is triggered when the user is typing some text in the box and we want to provide suggestions based on the app's data. For example, if the **SearchBox** control is used to perform a search on a set of customers stored in a local database, we could suggest to the user the customers whose names matches the one he or she is typing. The following sample shows how to manage this event handler to provide suggestions:

```
private void OnSuggestionRequested(SearchBox sender,
```

```
SearchBoxSuggestionsRequestedEventArgs args)
{
    List<Person> list = new List<Person>
    {
        new Person
        {
            Name = "Matteo",
            Surname = "Pagani"
        },
        new Person
        {
            Name = "Ugo",
            Surname = "Lattanzi"
        },
        new Person
        {
            Name = "Marco",
            Surname = "Dal Pino"
        }
    };
    IEnumerable<string> names = list.Select(x => x.Name);
    args.Request.SearchSuggestionCollection.AppendQuerySuggestions(names);
}
```

The event handler's parameter contains a property called **`Request`**, which offers many ways to interact with the **`SearchBox`**. Specifically, we are interested in a collection called **`SearchSuggestionCollection`**, which is the list of suggestions that are proposed to the user. We can add a single word by using the **`AppendQuerySuggestion()`** method or, as in the previous sample, we can add a list of words by using the **`AppendQuerySuggestions()`** method. You will notice that the suggestions are simple strings. Consequently, in the previous sample, we perform a LINQ query by using the **`Select()`** method to retrieve only the values of the **`Name`** property from the collection of **`Person`** object we have previously created:

Figure 12: The SearchBox control in Windows 8.1

Adding Search in Windows Phone

To add search in a Windows Store app for Windows Phone, we are going to use a similar approach to the one we have just seen for Windows. The difference is, instead of using the **SearchBox** control (which is not available in Windows Phone), we are going to use a control called **AutoSuggestBox**. Unlike the one in Windows 8.1, this control is not specific for search scenarios. It can be used in all situations in which you want to provide suggestions to the user while he or she is typing some text.

Consequently, it behaves like a standard **TextBox** control. It does not offer an integrated button to trigger the search (such as the magnifier icon in the **SearchBox** control) so we need to trigger it in another way (for example, by relying on a **Button** control placed near the box). Like with a **TextBox** control, we will be able to retrieve the search keyword using the **Text** property. The following sample shows how to add this control to a page:

```
<StackPanel>
    <AutoSuggestBox PlaceholderText="Search a person" x:Name="SearchBox" />
    <Button Content="Search" Click="OnSearchStarted" />
</StackPanel>
```

Also, this control offers a property called **PlaceholderText** to set a text that is displayed in the box until the user starts typing some text. The following sample shows how to use the **Button** control we have included in the page to trigger a search:

```
private void OnSearchClicked(object sender, RoutedEventArgs e)
{
    Frame.Navigate(typeof(SearchPage), SearchBox.Text);
}
```

As you can see, the code is simple. We redirect the user to a specific page (in this sample, called **SearchPage**), which will display all of the results according to how our data is structured inside the app. As navigation parameter, we pass to the destination page the search keyword, which is stored in the **Text** property of the control. We will be able to retrieve it, as usual, by using the **OnNavigatedTo()** method.

Suggestions in the **AutoSuggestBox** control are implemented in a different way than the **SearchBox** one. Instead of offering a specific event, the approach is similar to the one used by controls such as **ListView** or **GridView**. We are going to set the list of suggestions as the value of the **ItemsSource** property of the control, like in the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    List<Person> list = new List<Person>
    {
```

```
        new Person
        {
            Name = "Matteo",
            Surname = "Pagani",
        },
        new Person
        {
            Name = "Ugo",
            Surname = "Lattanzi",
        },
        new Person
        {
            Name = "Marco",
            Surname = "Dal Pino",
        }
    };
    IEnumerable<string> names = list.Select(x => x.Name);
    SearchBox.ItemsSource = names;
}
```

However, the **AutoCompleteBox** control offers a deeper way to customize the drop-down menu that is displayed with the suggestions when the user starts typing some text. Let's take a look at the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    List<Person> list = new List<Person>
    {
        new Person
        {
            Name = "Matteo",
            Surname = "Pagani",
            Role = "Support Engineer"
        },
        new Person
        {
            Name = "Ugo",
            Surname = "Lattanzi",
            Role = "Web Developer"
        },
        new Person
        {
            Name = "Marco",
            Surname = "Dal Pino",
            Role = "Mobile Developer"
        }
    };
    SearchBox.ItemsSource = list;
```

```
}
```

As you can see, we have added a new property to the person class called **Role**, which contains the job position of the user. In addition, we did not filter the collection with LINQ to retrieve just the list of names. We have set, as **ItemsSource** of the **AutoSuggestBox** control, the full list of **Person** objects.

The **AutoSuggestBox** control behaves like a **ListView** or **GridView** control so we are able to define the **ItemTemplate** property with a **DataTemplate**, which will be used to define the layout of every single suggestion in the drop-down list. The following sample shows how to define the **ItemTemplate** so that the suggestions list displays the users' first and last names above their roles. In this sample, their names will be listed in a bigger font than their roles:

```
<AutoSuggestBox PlaceholderText="Search a person"
                AutoMaximizeSuggestionArea="True"
                TextMemberPath="Name"
                x:Name="SearchBox">
    <AutoSuggestBox.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding Path=Name}" Margin="0, 0, 5, 0" />
                    <TextBlock Text="{Binding Path=Surname}" />
                </StackPanel>
                <TextBlock Text="{Binding Path=Role}" FontSize="16" />
            </StackPanel>
        </DataTemplate>
    </AutoSuggestBox.ItemTemplate>
</AutoSuggestBox>
```

It is important to mention the usage of a property called **TextMemberPath**. Even if, in this case, we are displaying a list of suggestions by using a complex object (the **Person** class), in the end what the **AutoSuggestBox** needs is a string, which is inserted in the box when the user taps on the suggestion. Consequently, when we use complex objects, we need to specify which property of the object will be used to fill the box. In the previous sample, we have set the property to **Name**. This way, when the user will tap on a suggestion, the name of the person will automatically be inserted in the control.

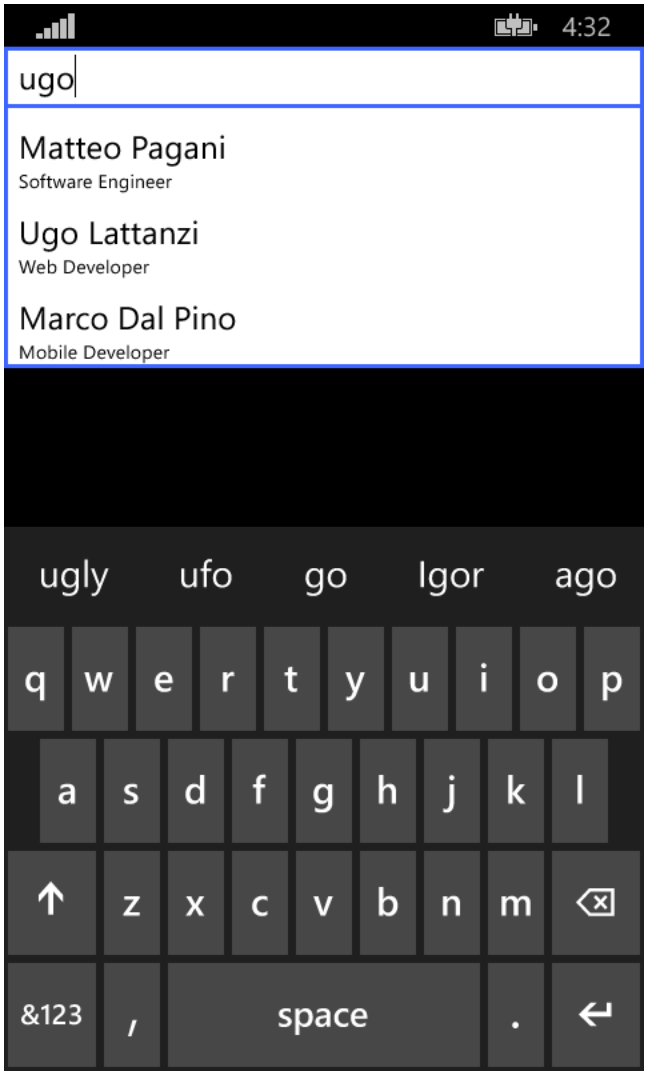


Figure 13: The AutoSuggestBox control in Windows Phone

Extensions

Extensions are similar to contracts except that, in this case, the agreement is not established between two apps but between the app and the OS. Let's take a look at the most important ones.

File Activation

When an app registers for the file extension activation, it is able to manage a set of specific file types. Whenever the user tries to open a file whose extension has been registered by our app, we will be able to intercept the request and process the file. Let's take a look, in detail, at how to

support file activation from both ways: the source app (the one that opens the file) and the target app (the one that receives the file).

Receiving the File

To be able to support a set of file types, we need to add the **File Type Associations** declaration in the **Declarations** section of the manifest file. We will be asked to specify the following information:

- **Name**, which is a unique identifier of the file type
- One or more file types we want to support; the only required information is the extension (such as .log) but we can also specify the content type (for example, if it is a text file, we can set it to text/plain)
- A logo, which will be used to identify the app; it's used in case there are more apps that support the same file type (a screen will display the list of apps, each one represented with the logo and the name)

Note: There is a set of extensions (such as .bat or .exe) which can't be registered by an app, otherwise you will get an exception. You can find the full list of prohibited extensions in the MSDN documentation here.

File activation, as we have seen with contracts, is managed with a specific activation method in the **App** class, which is triggered when the app is opened consequently to a file activation request. The method is called **OnFileActivated()** and the following sample shows how to manage it:

```
protected override void OnFileActivated(FileActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
        rootFrame = new Frame();
    }
    rootFrame.Navigate(typeof(ViewFilePage), args.Files[0]);
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
}
```

The approach is similar to the one we saw for the sharing contract. We have added to the project a specific page that will deal with the received file, which is called **ViewFilePage**. When the app has been opened due to a file activation, we redirect the user to this page passing, as navigation parameter, the received file. The file is stored inside a collection called **Files**, which is a property of the event handler parameters. However, since a file activation can't occur with more than one file, it is enough to retrieve the first item of the collection and pass it to the destination page.

Thanks to the navigation system, we are able to retrieve, in the **OnNavigatedTo()** method of the destination page, the file, which is represented by the **StorageFile** class, which is the common class we saw in Chapter 5 that identifies files in the Windows Runtime. The following sample shows an app that registered to receive text files. Consequently, we will display its content to the user with a pop-up message by using the **FileIO** class we learned in Chapter 5:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    if (e.Parameter != null)
    {
        StorageFile file = e.Parameter as StorageFile;
        string content = await FileIO.ReadTextAsync(file);
        MessageDialog dialog = new MessageDialog(content);
        await dialog.ShowAsync();
    }
}
```

Opening the File

So far, we have seen how to register an app so that it is able to open one or more file types. Now, it is time to see the other side of the channel: how can an app open a file? We can do it by using the **LaunchFileAsync()** method offered by the **Launcher** class, which belongs to the **Windows.System** namespace, like in the following sample:

```
private async void OnOpenFileClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("text.log");
    await Launcher.LaunchFileAsync(file);
}
```

The previous sample tries to open a file called **text.log**, which is stored in the local storage of the app. Now the OS will look for any app registered to support the **.log** extension and it will pass the file to the default one. However, there is an important difference to highlight between Windows and Windows Phone.

In fact, Windows allows developers to register any extension (except for the one previously mentioned such as .bat or .exe), including the ones that are managed by apps included in the OS itself such as images or audio files. By default, the **LaunchFileAsync()** method opens the default app registered for the file type. For example, if you try to open a JPG image, Windows will open the Photos app. However, you can also allow users to choose which app they want to use by passing another parameter to the **LaunchFileAsync()** method, like in the following sample:

```
private async void OnOpenFileClicked(object sender, RoutedEventArgs e)
```

```
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("text.log");
    await Launcher.LaunchFileAsync(file, new LauncherOptions
    {
        DisplayApplicationPicker = true
    });
}
```

We have created a new **LauncherOptions** object and set its **DisplayApplicationPicker** property to **true**. The result will be that, instead of directly opening the default app, Windows will display a picker like the following one:

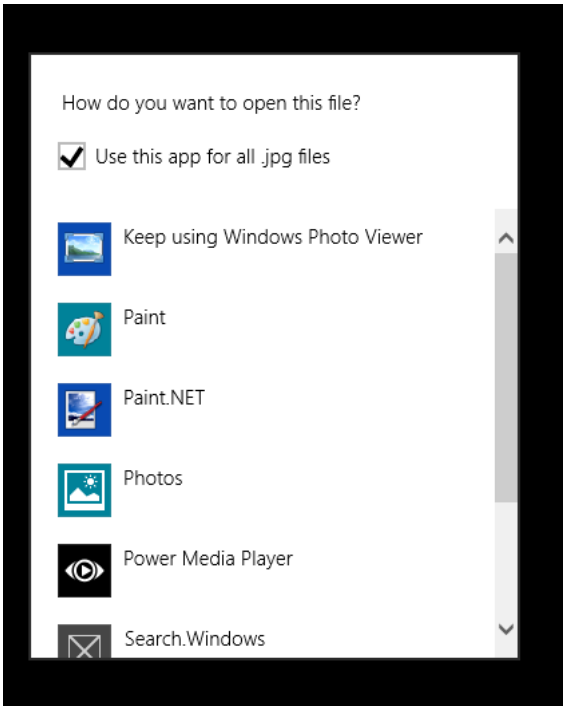


Figure 14: The app picker

Windows Phone, however, is not able to manage this scenario. We will not be able to manage our app's file types which are natively supported by the OS such as images, audio or Office files. We will not get any exception (like when we try to register a prohibited extension such as .exe); the association will simply not work. Windows Phone will always use the default app to open such files.

However, Windows Phone also supports the concept of app picker but this applies only to extensions that are not natively supported by the OS. For example, if we install from the Store more than one app that is able to manage the .log extension (which is a custom file type not supported by Windows Phone), the OS will prompt a screen to the user containing a list of all of the apps that support it.

What happens if we do not have any app that is able to manage the required file extension? On both Windows and Windows Phone, a pop-up message will inform the user and will allow him or her to search for an app on the Store. The results will automatically be filtered only for the apps that are able to manage such an extension. However, in Windows, you are also able to force the file opening. You will be able to choose one of the installed apps even if they did not explicitly register to support that file extension.

Protocol Activation

The Protocol activation extension is similar to the File activation one except that, instead of being able to support a file type, we are able to support a URI scheme. This way, we can intercept when an app tries to invoke a command by using a specific protocol (for example, log://). The difference is, in this case, the other apps will be able to pass only plain data since the URI is a string. This extension is often used when we want to provide other apps a way to run a specific command or to display a specific view of our app. As for the protocol activation, let's take a look at how to implement this feature both in a source app (the one that will invoke a URI) and in a target app (the one that will receive the URI).

Receiving a URI

As with the other contracts and extensions, the first step is to add a declaration in the manifest file. In this case, we need to select, from the **Available declarations** drop-down menu, the **Protocol** item. The only information required is the **Name** field, which is the name of the URI scheme we want to register (for example, if we want to subscribe to the log:// schema, we need to specify log as name). Optionally, you can also specify an image in the **Logo** field, which will be used in case there are multiple apps registered to handle the same protocol.

Note: In this scenario, there are also some protocols that are reserved by the system and can't be registered by a third-party app. You can find the complete list in the MSDN documentation here.

The next step, as we did for the file activation extension, is to provide a page to redirect the user to when the app is launched due to a protocol activation. As with every other event connected to the app's life cycle, we do it in the **App** class. However, there is not a specific method to manage the protocol activation so we need to use the generic **OnActivated()** method and detect if we are in this scenario by checking the value of the **Kind** property, like in the following sample:

```
protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.Protocol)
    {
        ProtocolActivatedEventArgs eventArgs = args as ProtocolActivatedEventArgs;
        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
```

```
            rootFrame = new Frame();
        }
        rootFrame.Navigate(typeof(UriViewPage), eventArgs.Uri);
        Window.Current.Content = rootFrame;
        Window.Current.Activate();
    }
}
```

The approach should be familiar. We simply redirect the user by using the **`Navigate()`** method of the **`Frame`** class to the page we added to our project to manage the protocol activation scenario (in this case, it is called **`UriViewPage`**). As navigation parameter, we add the full URI that has been invoked, which is stored in the **`Uri`** property of the activation parameters.

Consequently, we can retrieve this information in the landing page by using the **`OnNavigatedTo()`** method, as usual, like in the following sample that simply shows to the user a pop-up message with the received URI:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    if (e.Parameter != null)
    {
        Uri uri = e.Parameter as Uri;
        MessageDialog dialog = new MessageDialog(uri.AbsoluteUri);
        await dialog.ShowAsync();
    }
}
```

Opening a URI

A third-party app can invoke a URI simply by calling the **`LaunchUriAsync()`** of the same **`Launcher`** class we have seen for the file activation, like in the following sample:

```
private async void OnLaunchUriClicked(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri("log:/text", UriKind.Absolute);
    await Launcher.LaunchUriAsync(uri);
}
```

Also, in this case, when it comes to dealing with the two platforms, we can apply all of the warnings we have discussed about file activation. Windows allows developers to register any protocol even if they are supported by native apps. The **`LaunchUriAsync()`** directly launches the default app but you can add a second parameter (which type is **`LauncherOptions`**) to display a picker with the list of all of the apps that have registered to support that protocol, like in the following sample:

```
private async void OnLaunchUriClicked(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri("http://www.qmatteoq.com", UriKind.Absolute);
    await Launcher.LaunchUriAsync(uri, new LauncherOptions
    {
        DisplayApplicationPicker = true
    });
}
```

For example, the previous code will show the list of all of the browsers installed on your device that are able to handle the HTTP protocol. However, the previous code will not trigger any picker in Windows Phone. As developers, you can't override protocols that are registered by native apps (such as HTTP, which can be opened only by Internet Explorer). You will not get any exception but your protocol registration will simply be ignored.

Sending Text Messages (Windows Phone Only)

Windows Phone includes a set of APIs that can be used to send text messages to another person. However, these APIs always require the user's permission before sending a message. If you need to send text messages without interacting with the user, you will have to rely on third-party services.

Text messages are identified by the **ChatMessage** class, which is included in the **Windows.ApplicationModel.Chat** namespace that offers many properties to customize the text message's look and feel. The most important are **Body** to define the text of the message and **Recipients** which is a collection of all of the phone numbers that will receive the message. After you have defined the message, you need to pass it as parameter to the **ShowComposeSmsMessageAsync()** method exposed by the **ChatMessageManager** class.

```
private async void OnSendMessageClicked(object sender, RoutedEventArgs e)
{
    ChatMessage message = new ChatMessage();
    message.Body = "Message";
    message.Recipients.Add("012345678");
    await ChatMessageManager.ShowComposeSmsMessageAsync(message);
}
```

The previous code will open the **Messaging** app in Windows Phone with a new message already populated with the information that we have defined.

Sending Email Messages (Windows Phone Only)

Despite the fact that Windows also includes a native Mail app, only Windows Phone offers a set of APIs that can be used to create a new email message. These APIs work in the same way as the text message APIs we saw. The email message will not be silently sent but, rather, we will just create a new email message populated with all of the required information. The user will have to confirm the operation and actually send the email message.

An email message is identified by the **`EmailMessage`** class, which is part of the **`Windows.ApplicationModel.Email`** namespace. After you have defined all of the required properties, you are able to send it by calling the **`ShowComposeNewMailAsync()`** method offered by the **`EmailManager`** class, like in the following sample:

```
private async void OnSendMailClicked(object sender, RoutedEventArgs e)
{
    EmailMessage mail = new EmailMessage();
    mail.Subject = "Subject";
    mail.Body = "Body";
    mail.To.Add(new EmailRecipient("info@qmatteoq.com", "Matteo Pagani"));
    await EmailManager.ShowComposeNewEmailAsync(mail);
}
```

As you can see, the **`EmailMessage`** class offers some basic properties to customize the email message's content such as **`Subject`** or **`Body`**. In addition, it offers three collections to add to the list of recipients: **`To`** (for regular recipients), **`Cc`** (for carbon copy recipients), and **`Bcc`** (for blind carbon copy recipients). Every recipient is identified by the **`EmailRecipient`** class, which requires the email address and the recipient's name when it is instantiated.

One important feature that has been added with these new APIs (which was not available in Windows Phone 8.0) is support for attachments. Now you will be able to add a file to the email message as an attachment by using the **`Attachment`** collection:

```
private async void OnSendMailClicked(object sender, RoutedEventArgs e)
{
    EmailMessage mail = new EmailMessage();
    mail.Subject = "Subject";
    mail.Body = Body";
    mail.To.Add(new EmailRecipient("info@qmatteoq.com", "Matteo Pagani"));
    StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync("Document.docx");
    var stream = RandomAccessStreamReference.CreateFromFile(file);
    EmailAttachment attachment = new EmailAttachment(file.Name, stream);
    mail.Attachments.Add(attachment);
    await EmailManager.ShowComposeNewEmailAsync(mail);
}
```

The first step is to retrieve a reference to the **StorageFile** object that identifies the file we want to include in the email message. In the previous sample, we attach a file called Document.docx stored in the local storage. Then, we create a new **EmailAttachment** object, which requires as parameters the file name and the stream with the file's content. We retrieve the stream by using the **CreateFromFile()** method of the **RandomAccessStreamReference** class. As parameter, we pass the **StorageFile** we have just retrieved. In the end, we just add the **EmailAttachment** object we created to the **Attachments** collection of the **EmailMessage** class.

Using the Speech Services

The Windows Runtime offers developers a set of services that can be used to allow people to interact with their apps by using voice. The user, instead of interacting with the app by tapping on the touch screen, can use voice commands, which are useful in scenarios in which it is not comfortable or safe to constantly watch the screen (such as while you are driving). In addition, Windows Phone provides a set of advanced vocal features thanks to the integration with Cortana, the digital assistant (the name is inspired by the Halo's video game series) that has been introduced in 8.1. There are three scenarios that you can implement in an app:

- **Text-to-Speech (TTS)**: By using a synthesized voice, the app is able to read text to the user; actually, it is the only scenario that is supported both both Windows and Windows Phone
- **Voice commands**: Thanks to this feature, the user will be able to open the app and perform an operation simply by pronouncing a command; this feature is strictly connected to Cortana. (In countries where the digital assistant is available, Cortana will take care of managing the command and performing the required operation.)
- **Speech recognition**: Voice commands are useful but they can be triggered only by the OS when the app is not running; however, when the user enters in our app, we can continue to use the speech features thanks to a set of APIs that can enable speech recognition so the user will be able to keep pronouncing commands or dictating text)

Let's take a look at how to use these features in detail.

Adding Text-to-Speech (TTS)

TTS, as previously mentioned, is the only feature supported by both platforms. However, only in Windows Phone will you have to enable the Microphone capability in the manifest file in order to use it. The speech synthesizer is identified by the **SpeechSynthesizer** class, which belongs to the **Windows.Media.SpeechSynthesis** namespace. The basic usage is simple. You just have to call the **SynthesizeTextToStreamAsync()** method to convert a text into an audio stream, which can be saved in the local storage or played to the user. Let's take a look at how to reproduce a synthesized text by using the **MediaElement** control, which will be explained in detail in Chapter 9. For the moment, it is important just to know that its purpose is to play multimedia elements such as music or video. However, there is an important behavior to highlight, and that is, playing the audio in the following way will interrupt the current audio on the device if, for example, the user is using the phone to listen to music. If you want to avoid this issue, you can find a detailed tutorial in this blog post here, which was published by Rajen Kishna (who works as a Microsoft Evangelist in The Netherlands).

```
private async void OnSpeakClicked(object sender, RoutedEventArgs e)
{
    SpeechSynthesizer synthesizer = new SpeechSynthesizer();
    SpeechSynthesisStream stream = await
synthesizer.SynthesizeTextToStreamAsync("This is a text to read");
    MediaElement element = new MediaElement();
    element.SetSource(stream, stream.ContentType);
    element.Play();
}
```

The text passed as parameter to the **SynthesizeTextToStreamAsync()** method is converted into a **SpeechSynthesisStream**. By using the **SetSource()** method of the **MediaElement** class, we are able to assign it to the multimedia player and reproduce it with the **Play()** method.

The Windows Runtime also offers a more advanced way to read a text by using a standard language called Speech Synthesis Markup Language (SSML). It is a standard based on XML and it has been defined by the W3C consortium. You can find the full specifications on the W3C's website here. Thanks to a set of special tags and attributes, this language is able to define how the text should be pronounced. For example, you can define whether or not a sentence should be pronounced slowly or fast or if it should be pronounced by a male or a female voice. Here is a SSML file sample:

```
<?xml version="1.0" encoding="utf-8" ?>
<speak xmlns="http://www.w3.org/2001/10/synthesis"
xmlns:dc="http://purl.org/dc/elements/1.1"
xml:lang="en"
version="1.0" >
  This text is pronounced normally
  <break time="500ms" />
  <prosody rate="x-slow">This text is pronounced at a slower speed</prosody>
  <voice gender="female">This text is read by a woman</voice>
</speak>
```

As you can see, some sentences are embedded into some special tags, such as:

- **break**: It can be used to add a pause before the next sentence is pronounced; by using the **time** attribute, you are able to specify the pause length
- **prosody**: It can be used to change how the text should be pronounced; in the previous sample, we use the **rate** attribute to set the reading speed
- **voice**: It can be used to specify the kind of voice to use to read the text; in the sample, we use the **gender** attribute to simulate a woman's voice

The Windows Runtime does not directly support loading a SSML file into the code. You will have to embed the XML into a string in your app and pass it as parameter of the **SynthesizeSsmlToStreamAsync()** method of the **SpeechSynthesizer** class, like in the following sample:

```
private async void OnSpeakClicked(object sender, RoutedEventArgs e)
{
    SpeechSynthesizer synthesizer = new SpeechSynthesizer();
    string ssml = @"<speak xmlns='http://www.w3.org/2001/10/synthesis'
                    xml:lang='en' version='1.0'>
                    This text is pronounced normally
                    <break time='500ms' />
                    <prosody rate='x-slow'>This text is pronounced at a slower
speed</prosody>
                    <voice gender='female'>This text is read by a woman
                    </voice>
                    </speak>";
    SpeechSynthesisStream stream = await
synthesizer.SynthesizeSsmlToStreamAsync(ssml);
    MediaElement element = new MediaElement();
    element.SetSource(stream, stream.ContentType);
    element.Play();
}
```

Voice Commands in Windows Phone

As previously mentioned, only the Windows Phone platform is able to support voice commands/ If you have previously worked with Windows Phone 8.0, you will find this next section familiar since the APIs are similar.

Voice commands are activated by holding the Search button that is available in all Windows Phone devices. Whether or not you are using Cortana, the phone will start to listen to your voice, waiting for a command to be pronounced. Commands are defined by using a Voice Command Definition (VCD) file, which is a special XML file. You can create a new one by adding a new XML file to your Visual Studio project.

As a sample scenario to introduce voice commands, we are going to create a notes management app. The user will be able to add and open notes simply by using his or her voice. Here is what the VCD file looks like:

```
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.1">
  <CommandSet xml:lang="en" Name="NotesCommandSetEnglish">
    <CommandPrefix>Notes</CommandPrefix>
    <Example> Use 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 />
    </Command>
```

```
  </CommandSet>
</VoiceCommands>
```

The VCD file includes a base node called **`CommandSet`**, which contains all of the commands that are supported by the app. A **`CommandSet`** is identified by two important attributes:

- **`xml:lang`**: It is the language to which the command set refers; a VCD file can have multiple **`CommandSet`** tags, one for each language supported by the app
- **`Name`**: It is a unique identifier of the command set

There are two other important information about the **`CommandSet`**, which are defined with two tags:

- **`CommandPrefix`**: It is an optional parameter which is the prefix that should be pronounced by the user before using the real command; if we do not set this prefix, the OS will automatically use the name of the app. (Consequently, this tag is useful when our app has a complex name, which could be hard to pronounce.)
- **`Example`**: Windows Phone is able to show, to the user, the list of all of the installed apps that are able to support voice commands; for each app, it will also display a brief sample of one of the available commands)

This sample below is taken from the **`Example`** tag:

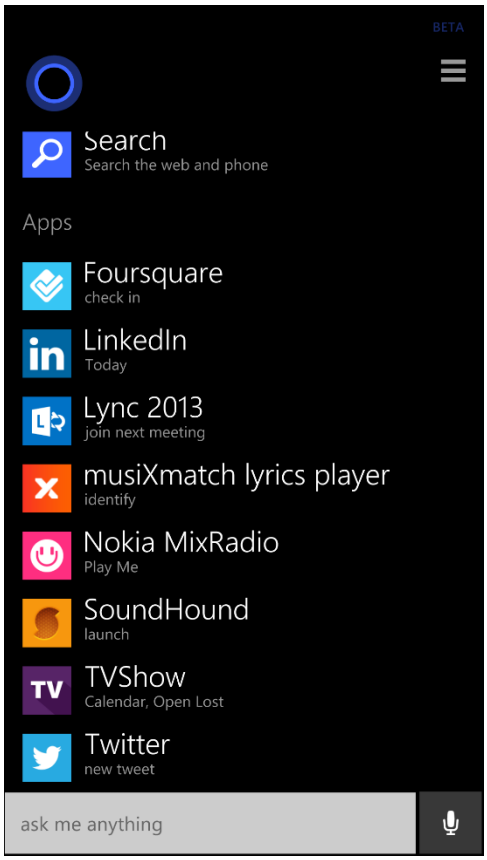


Figure 15: The list of installed apps that are able to support voice commands

Once we have defined the set, we can add all of the commands we want to support by using the **Command** tag. Every command is identified by a unique name, which is set with the **Name** attribute. In the previous VCD sample, the file contains just one command, identified by the **AddNote** name. For each command, we need to specify the following information:

- **Example**: Also in this scenario, we can show the user a sample of the command
- **ListenFor**: It is the most important information since it contains the command that the user can pronounce. (As you can see from the sample, we can add multiple **ListenFor** tags, one for each sentence that we want to support for the same command. A special feature of the **ListenFor** tag is that you can wrap some words inside square brackets. These words are optional; the command will be recognized whether or not the user will pronounce them. In the previous sample, the command will be activated whether or not the user has pronounced “add a new note” or “add new note” since the word “a” has been included inside square brackets)
- **Feedback**: It is a text that will be displayed to the user when the command has been recognized successfully
- **Navigate**: It is a parameter that is used to define which page of the app will handle the voice command. (However, in Windows Phone 8.1, it is no longer used since the app will always be opened by using a specific activation event, which will take care of redirecting the user to the proper page)

Once we have defined the VCD file, we need to install it into the system by using the **VoiceCommandManager** class included in the **Windows.Media.SpeechRecognition** namespace, like in the following sample:

```
private async void OnRegisterCommandClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
Package.Current.InstalledLocation.GetFileAsync("VoiceCommands.xml");
    await VoiceCommandManager.InstallCommandSetsFromStorageFileAsync(file);
}
```

The first step is to retrieve a reference to the VCD file included in the project. We use the **Package.Current.InstalledLocation** class we learned to use in Chapter 5. Then we pass the file as parameter of the **InstallCommandSetsFromStorageFileAsync()** method exposed by the **VoiceCommandManager** class.

Now the command set is installed and ready to be used. Voice command activation is managed by the **OnActivated()** method defined in the **App** class. Since it is the generic activation method, we need to check the **Kind** property to verify that we are in the voice command scenario. The value we expect is **VoiceCommand**:

```
protected override void OnActivated(IActivatedEventArgs args)
{
```

```
    if (args.Kind == ActivationKind.VoiceCommand)
    {
        VoiceCommandActivatedEventArgs commandArgs = args as
        VoiceCommandActivatedEventArgs;
        SpeechRecognitionResult result = commandArgs.Result;
        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
            rootFrame = new Frame();
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
        string commandName = result.RulePath.FirstOrDefault();
        switch (commandName)
        {
            case "AddNote":
                rootFrame.Navigate(typeof(AddNote));
                break;
        }
    }
}
```

Thanks to the method's argument, we are able to retrieve the result of the speech recognition operation, which is stored in the **`Result`** property, which is a **`SpeechRecognitionResult`** object. The property we are interested in is called **`RulePath`**, which contains all of the voice commands that have been used to activate the app. Since an app can only be opened by one command at a time, we retrieve just the first one. The value we will get in return is a string with the command name we have defined in the VCD file. In the previous sample, since our VCD file contains only one command definition, we check if the **`AddNote`** command has been triggered. If we fall under this case, we redirect the user to the page that is able to manage the given command (in this sample, it is a page called **`AddNote`** used to add a new note to the list).

Managing Command Parameters

Voice commands also offer a way to add some parameters. This way, other than just opening a specific page of the app, we can also trigger a specific operation. Let's consider the following scenario. In our notes management app, we want to provide a voice command to open an existing note. To make it properly work, we need to add a way to support a parameter with the note identifier that we want to open. The user should be able to pronounce a command such as "Open note 1." The following VCD sample shows how to support this scenario:

```
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.1">
  <CommandSet xml:lang="en" Name="NotesCommandSet">
    <CommandPrefix>Notes</CommandPrefix>
    <Example> Use 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>
```

The **OpenNote** command contains a keyword embedded into curly braces called number. It is a parameter which can assume a dynamic value from a set of keywords stored in a **PhraseList** element.

As you can see, right after the **Command** tag, we added a **PhraseList** section with a set of **Item** elements. Each of them is one of the parameters that is accepted by the command and that can be pronounced by the user. This means that the user will be able to pronounce a command such as “Open the note 2.” However, the previous sample has a downside: the list of supported parameters is fixed while, in most scenarios, the list can be dynamic (since new content can be included in the app). In our notes management app sample, the user is able to add new notes and he or she should be able to open them using a voice command.

For this reason, the Windows Runtime includes a method that is able to dynamically update a **PhraseList** section, like in the following sample:

```
private async void OnUpdateListClicked(object sender, RoutedEventArgs e)
{
    VoiceCommandSet set =
VoiceCommandManager.InstalledCommandSets["NotesCommandSet"];
    await set.SetPhraseListAsync("number", new string[] { "1", "2", "3", "4", "5"
});
}
```

The first step is to retrieve a reference to the **CommandSet** defined in the VCD file, thanks to the **InstalledCommandSets** collection offered by the **VoiceCommandManager** class. We retrieve it by using its unique identifier, which has been set in the **Name** attribute of the **CommandSet** tag in the VCD file. Then we can call the **SetPhraseListAsync()** method, passing two parameters. The first one is the name of the **PhraseList** section we want to update (in our case, it is number, which is the value of the **Label** property) and the second one is a collection of all of the parameters we want to support. It is important to mention that this method does not add the new parameters to the already existing ones but it always overrides the existing list. Consequently, you will always have to pass all of the supported parameters each time. For example, in a real notes management app, you will have to add the identifier of each note stored in the app every time the user adds a new note.

Now, let's take a look at how to change the **OnActivated()** method of the **App** class to manage the voice command's parameter:

```
protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.VoiceCommand)
    {
        VoiceCommandActivatedEventArgs commandArgs = args as
            VoiceCommandActivatedEventArgs;
        SpeechRecognitionResult result = commandArgs.Result;
        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
            rootFrame = new Frame();
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
        string commandName = result.RulePath.FirstOrDefault();
        switch (commandName)
        {
            case "OpenNote":
                if (result.SemanticInterpretation.Properties.ContainsKey("number"))
                {
                    string selectedNote = result.SemanticInterpretation.
                        Properties["number"].FirstOrDefault();
                    rootFrame.Navigate(typeof (DetailPage), selectedNote);
                }
                break;
        }
    }
}
```

When the app is activated by a voice command that contains a parameter, the **SpeechRecognitionResult** object, other than just the pronounced command, contained a collection called **Properties**, which is part of the **SemanticInterpretation** object. This collection contains a list of all of the supported parameters. We can retrieve the value simply by using, as key, the name of the parameter, which is the value we assigned to the **Label** property of the **PhraseList** section in the VCD file (in the previous sample, it is number). This way, we are able to redirect the user to the proper page, adding as the navigation parameter the number of the note to open, and retrieve it as usual by using the **OnNavigatedTo()** method of the destination page:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    string noteId = e.Parameter.ToString();
    //load the note with the selected id from the database
}
```

Managing Commands with a Free Parameter

The features that we have seen so far were also available in Windows Phone 8.0. One of the new features that has been introduced in Windows Phone 8.1 is support for commands with a free parameter. Instead of pronouncing a specific command, the user can dictate a full text, which is received by the app. Let's again consider our notes management app. The user could invoke the command to create a new note and immediately dictate the note's text, without waiting for the app to be opened.

The approach to support this scenario in a VCD file is similar to the one we have seen for traditional commands. The main difference is that we are going to use the **PhraseTopic** tag instead of the **PhraseList** one, like in the following sample:

```
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.1">
  <CommandSet xml:lang="en" Name="NotesCommandSet">
    <CommandPrefix>Notes</CommandPrefix>
    <Example> Use notes and add a new note </Example>
    <Command Name="AddNote">
      <Example> add a new note with text "buy the milk"</Example>
      <ListenFor>add a [new] note with text {noteText} </ListenFor>
      <Feedback>Adding a new note...</Feedback>
      <Navigate />
    </Command>
    <PhraseTopic Label="noteText" />
  </CommandSet>
</VoiceCommands>
```

The syntax is the same as we previously saw; the parameter is included into curly braces (in this case, it is named **noteText**). The name of the parameter is set with the **Label** property of the **PhraseTopic** tag. That is all. Since the text can be freely dictated, we do not have a fixed list of supported parameters.

Now, let's take a look at how we can retrieve this new type of parameter in the **OnActivated()** method of the **App** class:

```
protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.VoiceCommand)
    {
        VoiceCommandActivatedEventArgs commandArgs = args as
            VoiceCommandActivatedEventArgs;
        SpeechRecognitionResult result = commandArgs.Result;
        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
            rootFrame = new Frame();
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
        string commandName = result.RulePath.FirstOrDefault();
        switch (commandName)
        {
            case "AddNote":
                if
(result.SemanticInterpretation.Properties.ContainsKey("noteText"))
                {
                    string noteText =
result.SemanticInterpretation.Properties["noteText"].
                        FirstOrDefault();
                    rootFrame.Navigate(typeof (DetailPage), noteText);
                }
                break;
        }
    }
}
```

As you can see, the code is the same as we have seen for the standard parameters. If the full text has been successfully recognized, we will find an item in the **Properties** collection with, as key, the identifier we assigned to the **PhraseTopic** tag in the VCD file. This way, we can retrieve the text and pass it, as navigation parameter, to a specific page of the app so that we can retrieve it by using the **OnNavigatedTo()** method, like in the following sample:

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    string noteText = e.Parameter.ToString();
```

```
    MessageDialog dialog = new MessageDialog(noteText);
    await dialog.ShowAsync();
}
```

Speech Recognition

The last supported scenario for speech services, which again is only available on Windows Phone, is speech recognition. Once the app is opened, we are no longer able to use the voice commands feature we have seen before to intercept the user's voice. Thanks to the **SpeechRecognizer** class, which is part of the **Windows.Media.SpeechRecognition** namespace, we are able to implement this feature in our apps.

Speech Recognition Using the Native Windows Phone Interface

The **SpeechRecognizer** class offers a way to recognize a text by using the same user interface (UI) that Windows Phone offers to intercept a voice command. This feature is activated by calling the **RecognizeWithUIAsync()** method:

```
private async void OnRecognizeTextClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    await recognizer.CompileConstraintsAsync();
    SpeechRecognitionResult result = await recognizer.RecognizeWithUIAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

Before using the speech recognition methods, it is always important to call the **CompileConstraintsAsync()** method, which prepares the grammar required to recognize the pronounced text. Then we can call the **RecognizeWithUIAsync()** method, which will display the speech recognition interface. After the recognition is completed, you will get a **SpeechRecognitionResult** object in return containing all of the information about the recognized text.

Thanks to the **Status** property, we are able to determine whether or not the operation was successful. If it was, the property is set with the **Success** value of the **SpeechRecognitionResultStatus** enumerator. Consequently, we can retrieve the recognized text by using the **Text** property. We also have a way to customize the recognition dialog's appearance by setting the following properties exposed by the **UIOptions** object of the **SpeechRecognizer** class:

- **AudiblePrompt**: It is the title of the dialog

- **ExampleText**: It is a text, displayed under the title, which shows an example of the expected result
- **IsReadBackEnabled**: By default, once the recognition is completed, Windows Phone will read the recognized text back to the user. (We can disable this behavior by setting this property to false)
- **ShowConfirmation**: By default, once the recognition is completed, Windows Phone will show the recognized text in the dialog. (We can disable this behavior by setting this property to false)

The following sample shows how to customize the recognition dialog by using these properties:

```
private async void OnRecognizeTextClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    await recognizer.CompileConstraintsAsync();
    recognizer.UIOptions.AudiblePrompt = "I'm listening for the note's text";
    recognizer.UIOptions.ExampleText = "Remember to buy the milk";
    recognizer.UIOptions.IsReadBackEnabled = false;
    recognizer.UIOptions.ShowConfirmation = false;
    SpeechRecognitionResult result = await recognizer.RecognizeWithUIAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

Standard Speech Recognition

We can use the **SpeechRecognizer** class to also recognize a text without using the Windows Phone dialog. The app will simply wait for a text or a command without showing anything to the user. The approach is similar to the one we saw in the previous section, except that, in this case, we are going to use the **RecognizeAsync()** method. The result is the same; we will get, in return, a **SpeechRecognitionResult** object, which stores the recognized text in the **Text** property. The following sample shows a basic usage of this method:

```
private async void OnRecognizeTextClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    await recognizer.CompileConstraintsAsync();
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

Customizing the Grammar

By default, the **SpeechRecognizer** class uses a grammar that is suitable to recognize generic texts. Consequently, it includes most of the words that compose a language's grammar. In some scenarios, having such a big grammar is useless. For example, if we are using speech recognition just to manage a command (such as Confirm or Cancel), it would be better to use a limited grammar to avoid wrong recognitions. In these scenarios, the **SpeechRecognizer** class offers a collection called **Constraints**, which is a list of supported grammars. The Windows Runtime offers many classes that represent different grammars; each of them implements the basic **ISpeechRecognitionConstraint** interface.

One of these classes is called **SpeechRecognitionTopicConstraint**, which can be used to define a grammar suitable for a specific subject. Let's take a look at the following sample:

```
private async void OnRecognizeTextClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    ISpeechRecognitionConstraint constraint = new SpeechRecognitionTopicConstraint
    (SpeechRecognitionScenario.WebSearch, "Windows Phone");
    recognizer.Constraints.Add(constraint);
    await recognizer.CompileConstraintsAsync();
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

When we create a new **SpeechRecognitionTopicConstraint** object, we need to set two parameters. The first parameter is the type of search and we set this by using a **SpeechRecognitionScenario** enumerator's value (it can be **WebSearch** for a web search scenario or **Dictation** for a generic text). The second parameter is the keyword for which we want to optimize the search. After this step, we can proceed with the recognize operation as usual. We just remember to call the **CompileConstraintsAsync()** method of the **SpeechRecognizer** class right after defining the grammar and before starting the recognition.

Another supported customization is using a grammar with a limited set of words, which is useful whenever you want to manage voice commands but not recognize a full text. Let's use, again, the notes management app we previously described. Let's say that we have included a page on which the user can add a new note by dictating the note. After the text has been inserted, we want to ask the user to confirm the operation. In this case, it would be useless to provide the full grammar because we will want to recognize only the **Ok** and **Cancel** commands. Let's take a look at how we can support this scenario with a code sample:

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

```
    SpeechRecognizer recognizer = new SpeechRecognizer();
    ISpeechRecognitionConstraint constraint = new SpeechRecognitionListConstraint
    (new[] { "OK", "Cancel" });
    recognizer.Constraints.Add(constraint);
    await recognizer.CompileConstraintsAsync();
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success &&
    result.Confidence != SpeechRecognitionConfidence.Rejected)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

In the previous sample, we created a new **SpeechRecognitionListConstraint** object, which accepts, as parameter, a collection of supported words. The rest of the code is the same as we previously saw. We call the **CompileConstraintsAsync()** method and we perform the recognition by using the **RecognizeAsync()** method. However, there is an important difference to highlight. The **Status** property of the **SpeechRecognitionResult** class notifies the developer only if the text has been successfully recognized but it does not tell you if the recognized text is included in the list of supported words. Consequently, other than just checking whether or not the **Status** property is equal to **Success**, we need to also check the value of the **Confidence** property. Only if the value is different than **Rejected** can we be sure that the recognized text is valid.

The last chance to customize the grammar is offered by a standard defined by the W3 Consortium called the Speech Recognition Grammar Specification (SRGS), which is based upon XML. It allows us to define custom grammars in a powerful way. You can find the complete specifications on the standard on the W3C's website here.

This standard can be used to define, in a precise way, the set of words you want to support and the exact order in which they should be pronounced so that the command will be successfully recognized. Here is a SGRS file sample:

```
<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>
```

In this file, we can add a set of rules that are identified by the rule tag, which represents the supported commands (which are included in the one-of section). The order used to define the commands is important since the user will have to strictly follow it, otherwise the command will not be recognized. For example, the previous SGRS file can be used to accept commands such as “Open a note” or “Load a note” but “The note open” will not be accepted since the order does not match what we have defined.

SGRS files are standard XML files. To use them, you will simply have to add an XML file to your Visual Studio project. Then, you will be able to load it into your app by using the **`SpeechRecognitionGrammarFile`** class, which requires as parameter when a new instance is created a **`StorageFile`** object that references the SGRS file. The following sample shows how to load a SGRS file called **grammar.xml**, which is included in the Visual Studio project:

```
private async void OnRecognizeTextClicked(object sender, RoutedEventArgs e)
{
    SpeechRecognizer recognizer = new SpeechRecognizer();
    StorageFile file = await
Package.Current.InstalledLocation.GetFileAsync("grammar.xml");
    ISpeechRecognitionConstraint constraint = new
SpeechRecognitionGrammarFileConstraint(file);
    recognizer.Constraints.Add(constraint);
    await recognizer.CompileConstraintsAsync();
    SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    if (result.Status == SpeechRecognitionResultStatus.Success &&
        result.Confidence != SpeechRecognitionConfidence.Rejected)
    {
        MessageDialog dialog = new MessageDialog(result.Text);
        await dialog.ShowAsync();
    }
}
```

Chapter 5 Creating Multimedia Applications

Playing audio and video

The Windows Runtime offers a control, called **MediaElement**, that can be used to produce audio and video tracks within your application. However, as already mentioned in the previous chapter about using Speech APIs, it's important to highlight that this control, by default, stops any background audio activity running on the phone. The basic usage is very simple: just add the control to your page and set the **Source** property with the path of the media file. It can be either a remote URL or a local path (like a file in the local storage). The following sample shows how to use the **MediaElement** control to reproduce a remote audio file:

```
<MediaElement Source="http://wpdevfusion.com/podcasts/05-2014.mp3"
AutoPlay="True" x:Name="Media" />
```

With the **AutoPlay** property set to **true**, we are able to play the audio track as soon as the page is loaded. We can also load a media file by using the code behind; this is useful when we don't have direct access to the file we want to play, and we first need to retrieve it (for example, it needs to be downloaded from a website).

The following sample shows how to pick an audio file from the device (using the **FileOpenPicker** class we talked about in Chapter 5) and to play it:

```
private async void OnOpenFileClicked(object sender, RoutedEventArgs e)
{
    FileOpenPicker picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".mp4");
    StorageFile file = await picker.PickSingleFileAsync();
    IRandomAccessStreamWithContentType stream = await file.OpenReadAsync();
    Media.SetSource(stream, stream.ContentType);
}
```

After retrieving the **StorageFile** object that represents the audio file selected by the user, we open the stream by using the **OpenReadAsync()** method. Then, we pass the stream to the **SetSource()** method of the **MediaElement** control we added in the XAML page, passing as parameter the content type (which can be retrieved from the **ContentType** property of the **IRandomAccessStreamWithContentType** interface).

Note: If you read Chapter 5, you'll know that the previous sample code works only in a Windows app. Windows Phone requires a different management of the FileOpenPicker class, since it's not possible to keep two applications open at the same time.

Controlling the stream

The **MediaElement** control provides an embedded interface to control the stream, by providing the most common controls like Play, Pause, Next, etc. To enable it, we need to set the **AreTransportControlsEnabled** property to **True**, like in the following sample:

```
<MediaElement Source="http://wpdevfusion.com/podcasts/05-2014.mp3"
              AreTransportControlsEnabled="True"
              x:Name="Media"/>
```

However, if we want to manually control the stream, we can use the methods offered by the **MediaElement** control. The following sample shows a very basic interface to control the audio or video stream:

```
<StackPanel>
    <MediaElement Source="http://wpdevfusion.com/podcasts/05-2014.mp3"
x:Name="Media" />
    <Button Content="Play" Click="OnPlayClicked" />
    <Button Content="Pause" Click="OnPauseClicked" />
    <Button Content="Stop" Click="OnStopClicked" />
</StackPanel>
```

Here is an example of how we manage the **Click** events exposed by the **Button** controls we've added in the page:

```
private void OnPlayClicked(object sender, RoutedEventArgs e)
{
    Media.Play();
}
private void OnPauseClicked(object sender, RoutedEventArgs e)
{
    if (Media.CanPause)
    {
        Media.Pause();
    }
}
private void OnStopClicked(object sender, RoutedEventArgs e)
{
    Media.Stop();
}
```

To adapt the user interface to the current status of the player, we can use the **CurrentState** property of the **MediaElement** control, which returns one of the values of the **MediaElementState** enumerator. This class contains different values; each of them identifies one of the possible states, like **Playing**, **Paused**, **Stopped**, etc. We can also get real time notifications when the status changes, by subscribing to the **CurrentStateChanged** event. The following sample code shows how to use the **CurrentState** property to properly change the behavior of the user interface. When the Play button is pressed and the player is not playing, we're going to play the media file; otherwise we're going to pause it.

```
private void OnPlayClicked(object sender, RoutedEventArgs e)
{
    if (Media.CurrentState == MediaElementState.Paused || Media.CurrentState
== MediaElementState.Stopped)
    {
        Media.Play();
    }
    else if (Media.CurrentState == MediaElementState.Playing)
    {
        Media.Pause();
    }
}
```

The **MediaElement** control offers many other properties to customize the streaming, like:

- **IsMuted**: By setting it to **true**, you'll immediately set the volume to zero.
- **IsFullWindow**: By setting it to **true**, you'll force the player to run in full screen.
- **Volume** can be used to define the audio volume, from **0** (no audio) to **1** (maximum volume).
- **IsLooping**: When set to **true**, the media file will be played in loop until the user stops the player.
- **PosterSource**: The path of an image that will be displayed until the media file is fully loaded. It's especially useful when you're loading a file directly from a remote resource, like a website.
- **DefaultPlaybackRate**: A numeric value that can be used to increase the player speed. By default, it's set to **1**, but it can be changed with a smaller value (if you want to reproduce the media file slower) or with a bigger value (if you want to reproduce the media faster).
- **Position**: The current position (in time) of the player.

Managing the automatic screen lock

All smartphones and tablets offer an automatic screen lock feature. To preserve battery, after some time that the device is not used (which means that the user is not interacting with the device), it's automatically locked, and the current running application is suspended. However, this feature doesn't play well with multimedia applications; when the user is watching a movie, for example, she doesn't interact with the device, but that doesn't mean she's not using it.

Consequently, we can use a specific API to enable or disable this behavior. Typically, we're going to disable it when the media file is playing, and enable it again when the media file is paused or stopped. We can achieve this goal by using the **DisplayRequest** class, which is included in the **Windows.System.Display** namespace. The following sample shows how to improve the Play method we've already seen, by enabling or disabling the automatic screen lock feature according to the status of the player:

```
private void OnPlayClicked(object sender, RoutedEventArgs e)
{
    DisplayRequest request = new DisplayRequest();
    if (Media.CurrentState == MediaElementState.Paused || Media.CurrentState
==
    MediaElementState.Stopped)
    {
        request.RequestActive();
        Media.Play();
    }
    else if (Media.CurrentState == MediaElementState.Playing)
    {
        request.RequestRelease();
        Media.Pause();
    }
}
```

The automatic screen lock feature is disabled by calling the **RequestActive()** method of the **DisplayRequest** class; when the video is paused, we enable it again by calling the **RequestRelease()** method.

Playing audio in background

The code we've seen so far to interact with the **MediaElement** control has a limitation: once the application is suspended, the player is automatically stopped. For many scenarios, this limitation can ruin the user experience, since the user expects to keep listening to the audio while he's doing other activities (like sending email or surfing the web). For this reason, Windows and Windows Phone include a way to manage the playback when the application is not running. When you press the button to increase or decrease the system volume, or when you bring up the Charms bar, a native player will be displayed, with a preview of the current media element and a set of quick buttons to interact with the streaming. This native player is able to interact with the streaming using a background task, which will be detailed in Chapter 11. For the moment, it's important just to know that they are separate projects that are part of the application's solution, which contains some code that can be executed even when the application is not running. However, the implementation is different for the two platforms, as we'll see in the next sections.

Playing background audio in Windows 8.1

In Windows, the background task concept to manage audio has been implemented in a particular way; the application will act as a background task, so we won't have to manage the streaming in a separate project.

To support background streaming, we have to set a specific property of the **`MediaElement`** control, called **`AudioCategory`**; we need to set it to **`BackgroundCapableMedia`**, like in the following sample.

```
<MediaElement Source="http://wpdevfusion.com/podcasts/05-2014.mp3"
x:Name="Media" AudioCategory="BackgroundCapableMedia" />
```

Now we can enable background audio support in the manifest file. In the **Declaration** section you'll find, in the **Available declarations** dropdown menu, an item called **Background Tasks**. This declaration can be configured with a set of properties, according to the background task's type we want to support. We'll talk in detail about them in Chapter 11; in this case, just select the **Audio** type. The next step is to define the **Entry point**, which is the fully qualified name (including the namespace) of the class that manages the task. In this case, as we mentioned before, it's the app itself, so we will have to add the full signature of the **`App`** class of our application. For example, if we've created a project called **`MultimediaPlayer`**, we'll need to set as entry point the value **`MultimediaPlayer.App`**.

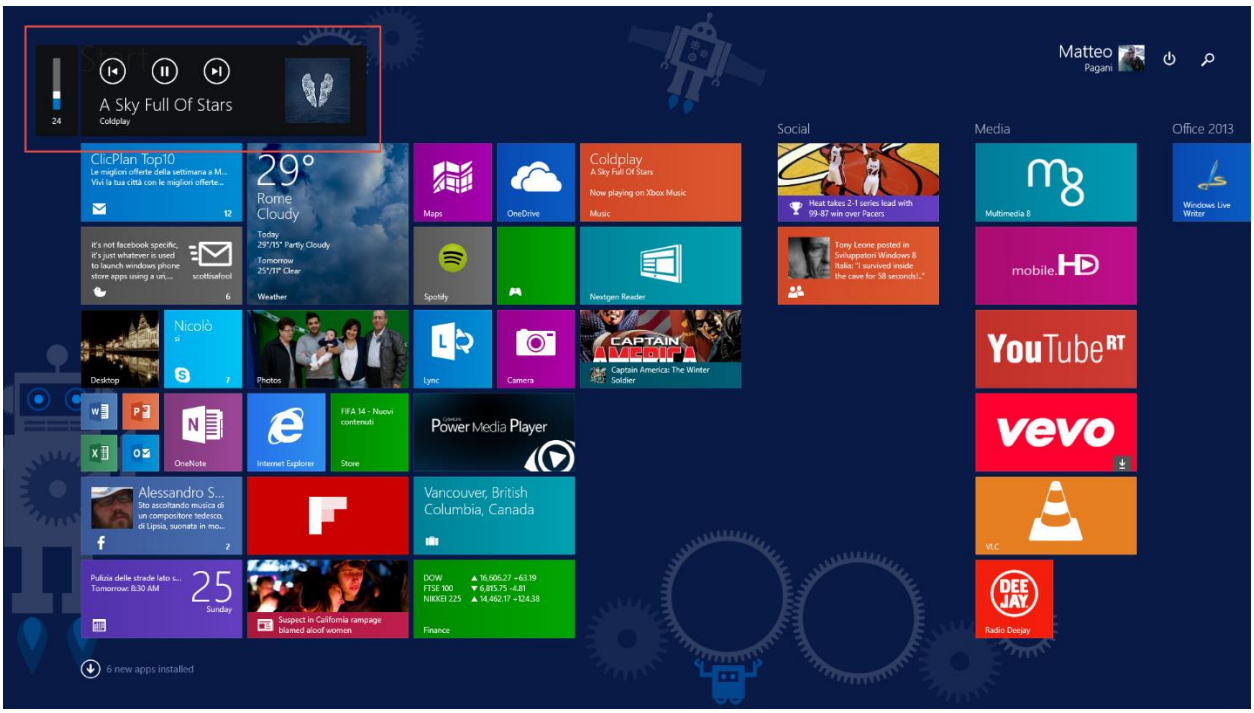


Figure 16: The background player in Windows 8.1

Now we need to apply some changes in the way we manage the **MediaElement** control. The first step is to get a reference to the **SystemMediaTransportControls** class, which is part of the **Windows.Media** namespace. Its purpose is to manage all the interactions with the native player, which can also occur when the application is not running; this way, we'll be able to change the playback state accordingly. Consequently, when the page is loaded, we need to get a reference to this class using the **GetForCurrentView()** method, and then define which states and events we want to manage, like in the following sample.

```
public sealed partial class MainPage : Page
{
    private SystemMediaTransportControls systemControls;

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

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        systemControls = SystemMediaTransportControls.GetForCurrentView();
        systemControls.ButtonPressed += SystemControls_ButtonPressed;
        systemControls.IsPlayEnabled = true;
        systemControls.IsPauseEnabled = true;
        Media.CurrentStateChanged += Media_CurrentStateChanged;
    }
}
```

In the **OnNavigatedTo()** method of the page, we perform a series of important operations:

- We define which player commands we want to enable and manage. The **SystemMediaTransportControls** class offers many **Boolean** properties, one for each command. In the previous sample, we've set to **true** the **IsPlayEnabled** and **IsPauseEnabled** properties. This way, we'll be able to manage the play and pause commands. Some other available properties are **IsFastForwardEnabled**, **IsRewindEnabled**, **IsPreviousEnabled**, and **IsNextEnabled**.
- We subscribe to the **ButtonPressed** event of the **SystemMediaTransportControls** object. It's invoked every time the player status changes as a consequence of an external action (for example, the user has stopped the playback using the native Windows interface, or by pressing a special key on their keyboard).
- We subscribe to the **CurrentStateChanged** event of the **MediaElement** control. We've already seen this event before, which is raised every time the playback status changes. It's important to synchronize the status of the **MediaElement** control with the status of the background player. The goal is to avoid a situation where, for example, the user presses the Pause button in the application, and the background player continues to display the Pause button instead of the Play button, as if the streaming hasn't been paused.

The following sample shows how to manage the **CurrentStateChanged** event:

```
void Media_CurrentStateChanged(object sender, RoutedEventArgs e)
{
    switch (Media.CurrentState)
    {
        case MediaElementState.Playing:
            systemControls.PlaybackStatus = MediaPlaybackStatus.Playing;
            break;
        case MediaElementState.Paused:
            systemControls.PlaybackStatus = MediaPlaybackStatus.Paused;
            break;
        case MediaElementState.Stopped:
            systemControls.PlaybackStatus = MediaPlaybackStatus.Stopped;
            break;
        case MediaElementState.Closed:
            systemControls.PlaybackStatus = MediaPlaybackStatus.Closed;
            break;
        default:
            break;
    }
}
```

As you can see, we just set the **PlaybackStatus** property of the **SystemMediaTransportControls** object with one of the values of the **MediaPlaybackStatus** enumerator, according to the state that is detected in the **CurrentState** property of the **MediaElement** control.

Now we will see how to manage the **ButtonPressed** event exposed by the **SystemMediaTransportControls** object:

```
private async void SystemControls_ButtonPressed(SystemMediaTransportControls
sender,
SystemMediaTransportControlsButtonPressedEventArgs args)
{
    if (args.Button == SystemMediaTransportControlsButton.Play)
    {
        await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            Media.Play();
        });
    }
    else if (args.Button == SystemMediaTransportControlsButton.Pause)
    {
        await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            Media.Pause();
        });
    }
```

```
}
```

Every time the user interacts with the background player, the **ButtonPressed** event is triggered, and by using the **Button** property of the method's parameter, we are able to detect which button has been pressed. Consequently, we need to change the playback status according to the invoked action. In the previous sample, you can see how to manage the **Play** and **Pause** commands. To perform the commands, we need to use the **Dispatcher**; this is required because the background player invokes the **ButtonPressed** event in a background thread, while the **MediaElement** control is declared in the user interface.

Showing the track information in the background player

If you've already used the native Music app that comes preinstalled on Windows, you might have noticed that the background player is able to display some information about the current you're playing, like title, author, or the album's cover. We can achieve the same result with our application, again using the **SystemMediaTransportControls** class. One of the properties offered by this object is called **DisplayUpdater**, and it can be used to update the information displayed in the background player.

The following sample shows how to perform this operation:

```
private void OnUpdateInfoClicked(object sender, RoutedEventArgs e)
{
    SystemMediaTransportControlsDisplayUpdater updater =
systemControls.DisplayUpdater;
    updater.Type = MediaPlaybackType.Music;
    updater.MusicProperties.Artist = "Coldplay";
    updater.MusicProperties.AlbumArtist = "Ghost Stories";
    updater.MusicProperties.Title = "A Sky Full Of Stars";
    updater.Thumbnail =
    RandomAccessStreamReference.CreateFromUri(new Uri("ms-
appx:///Music/music1_AlbumArt.jpg"));
    updater.Update();
}
```

The first step is to define, with the **Type** property, the type of content we're going to reproduce, which can be one of the values of the **MediaPlaybackType** enumerator: **Music** or **Video**. The rest of the code depends by the kind of media file we're playing. If it's an audio track, we can set the information using the **MusicProperties** object; if it's a video, we can use the **VideoProperties** object. The previous sample shows how to set information about a music track; we set properties like **Artist**, **AlbumArtist**, **Title**, and **Thumbnail**. After we've set all the properties, we need to call the **Update()** method to update the background player.

The **SystemMediaTransportControlsDisplayUpdater** class, however, offers an easier way to update the information: often, the audio or video track already contains the metadata that describes the file's content, so we can extract it to automatically update the background player. We can perform this operation by calling the **CopyFromFileAsync()** method of the **SystemMediaTransportControlsDisplayUpdater** class, which requires the media type and a **StorageFile** object that represents the media file.

```
private async void OnUpdateInfoClicked(object sender, RoutedEventArgs e)
{
    SystemMediaTransportControlsDisplayUpdater updater =
systemControls.DisplayUpdater;
    updater.Type = MediaPlaybackType.Music;
    StorageFile musicFile = await
ApplicationData.Current.LocalFolder.GetFileAsync("audio.mp3");
    await updater.CopyFromFileAsync(MediaPlaybackType.Music, musicFile);
    updater.Update();
}
```

The previous sample extracts the audio metadata from a file called audio.mp3, which is saved in the local storage, and then uses it to update the information displayed in the background player.

Playing background audio in Windows Phone

Even though many of the concepts we're going to see in this section are similar to the ones we've seen for Windows, Windows Phone manages the background audio in a different way. In this case, we can't use the application itself as a background task; we need to create a real one. Consequently, the first step is to add a new project to our solution. The template type we need to choose is called **Windows Runtime Component** (which is the standard template to create libraries for Windows Runtime-based applications).

By default, the project contains an empty class called **Class1**; you can delete it or rename it, as you prefer. What's important is that the class needs to implement the **IBackgroundTask** interface, which will require you to implement the **Run()** method. In standard background tasks, it's the method that is executed when the task is triggered.

```
namespace AudioPlayerApp.MusicTask
{
    public sealed class AudioTask : IBackgroundTask
    {
        public void Run(IBackgroundTaskInstance taskInstance)
        {
        }
    }
}
```

We're going to see very soon which code we will need to include in the agent to manage the audio playback. But first, let's see how to register the task in the main application. The approach is similar to the one we've seen for Windows, except that in this case the **Entry Point** won't be the app itself, but the background task's class we've just created. Consequently, we'll need to add a **Background Tasks** item in the **Declarations** section of the manifest file. After checking the **Audio** type, we need to include, in the **Entry Point** field, the fully qualified name of the background task's class, which is the full namespace plus the name of the class. For example, if we refer to the previous sample, the entry will be **`AudioPlayerApp.MusicTask.AudioTask`**.

The next step is to add a reference to the task in the main application, by choosing the **Add reference** option when you right-click on the Windows Phone project in the solution. Now the main application and the task are connected, and we can start writing the required code to make the background audio work.

In Windows Phone, the main application acts only as an interface to control the playback; all the "dirty work" will be done by the background task, which will take care of managing the stream. The background audio player is identified by the **`BackgroundMediaPlayer`** class, which has a unique instance across the whole system: we can't have two apps at the same time that control the background audio playback. When you're going to get a reference to the current instance of the class, the background task will be automatically executed, and the **`Run()`** method invoked. Here is a sample initialization of the main application:

```
public sealed partial class MainPage : Page
{
    private MediaPlayer mediaPlayer;

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

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        mediaPlayer = BackgroundMediaPlayer.Current;
    }
}
```

Managing the communication between the application and the background task

Before we define the background task, it's important to introduce messages, which are special objects that are used to communicate with the background task from the main application, and vice versa.

Messages are important because the playback management is controlled by the background task; every time the user interacts with the interface (for example, by playing the Play button), we won't control the player directly, but instead send a message to the background task, which will execute the required action. Messages are identified by the **`ValueSet`** class, which is simply a collection of objects identified by a unique key. Inside this collection, we can add multiple values, which can be retrieved by the receiver of the message.

Messages are exchanged using two methods exposed by the **`BackgroundMediaPlayer`** class:

- **`SendMessageToBackground()`** is used by the main application to send messages to the background task.
- **`SendMessageToForeground()`** is used by the background task to send messages to the main application.

On the other side, the **`BackgroundMediaPlayer`** class offers two specific events to intercept messages, which are:

- **`MessageReceivedFromBackground`** is subscribed by the main application, and is triggered when a new message is sent by the background task. Typically, these messages are used to update the user interface when the playback changes (for example, the background task has loaded a new track, and we need to display the new information in the application).
- **`MessageReceivedFromForeground`** is subscribed by the background task, and is triggered when a new message is sent by the main application. Typically, it's used when the user interacts with the main application, and we need to ask to the background task to perform an action on the player (like stopping the playback or playing the next track).

To better understand how to use messages, let's see a real example. Let's assume that, in the main application, we have a button that can be used to start the playback. Here is the code that is executed:

```
private void OnPlayClicked(object sender, RoutedEventArgs e)
{
    ValueSet message = new ValueSet();
    message.Add("title", "A Sky Full Of Stars");
    BackgroundMediaPlayer.SendMessageToBackground(message);
}
```

We've created a new message, which is the **`ValueSet`** object. Then, in the same way we would have done with a regular **`Dictionary`** collection, we add a new item to the message. In this case, it's the title of the audio track to reproduce, identified by a unique key, which is **`title`**. The message is sent to the background task by calling the **`SendMessageToBackground()`** method of the **`BackgroundMediaPlayer`** class.

What happens when the background task has started to play a new track and we want to display the information in the application? Here is a sample code:

```
public sealed partial class MainPage : Page
{
```

```
    private MediaPlayer mediaPlayer;

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

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        mediaPlayer = BackgroundMediaPlayer.Current;
        BackgroundMediaPlayer.MessageReceivedFromBackground +=
BackgroundMediaPlayer_MessageReceivedFromBackground;
    }

    private void BackgroundMediaPlayer_MessageReceivedFromBackground(object
sender,
    MediaPlayerDataReceivedEventArgs args)
    {
        ValueSet set = args.Data;
        if (set.ContainsKey("title"))
        {
            Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
            {
                Title.Text = set["title"].ToString();
            });
        }
    }
}
```

When the page is loaded, in the **`OnNavigatedTo()`** method, we've subscribed to the **`MessageReceivedFromBackground`** event exposed by the **`BackgroundMediaPlayer`** class. When the background task sends a message with the information about the current track, the event handler is triggered, and thanks to the **`Data`** property of the handler's parameter, we are able to retrieve the content of the message. In the previous sample, we assume that the message contains an item with the title of the song, identified by the **`title`** key. We simply display it using a **`TextBlock`** control placed in the page. The **`MessageReceivedFromBackground`** event is managed by a background thread, so we need to use the **`Dispatcher`** class to interact with the user interface.

Managing the playback with the background task

So far, we've only seen code samples performed by the main application. Now, let's see how to control the playback from the background task. Let's see a more detailed definition than the previous one:

```
public sealed class AudioTask : IBackgroundTask
{
```

```
    private BackgroundTaskDeferral _deferral;
    private SystemMediaTransportControls transportControls;

    public void Run(IBackgroundTaskInstance taskInstance)
    {
        transportControls = SystemMediaTransportControls.GetForCurrentView();
        transportControls.IsEnabled = true;
        transportControls.IsPauseEnabled = true;
        transportControls.IsPlayEnabled = true;
        transportControls.ButtonPressed += transportControls_ButtonPressed;
        BackgroundMediaPlayer.MessageReceivedFromForeground +=
BackgroundMediaPlayer_MessageReceivedFromForeground;
        BackgroundMediaPlayer.Current.CurrentStateChanged +=
Current_CurrentStateChanged;
        taskInstance.Canceled += TaskInstance_Canceled;
        taskInstance.Task.Completed += Task_Completed;
        _deferral = taskInstance.GetDeferral();
    }

    private void Task_Completed(BackgroundTaskRegistration sender,
BackgroundTaskCompletedEventArgs args)
    {
        BackgroundMediaPlayer.Shutdown();
        _deferral.Complete();
    }
}
```

The code executed by the **Run()** method is very similar to the code we've seen for Windows. We use the **SystemMediaTransportControls** class (which reference is retrieved by calling the **GetForCurrentView()** method) to define which playback controls we want to support. Again, we also subscribe to two events. The first is **ButtonPressed**, exposed by the **SystemMediaTransportClass**, which will be invoked every time the user interacts with the background audio player. The second event is **CurrentStateChanged**, offered by the **BackgroundMediaPlayer** instance, which is triggered every time playback state changes and we need to sync the background audio player with the real playback status.

We also subscribe to two events connected to the task's lifecycle, **Completed** and **Canceled**, which are invoked when the task has completed its execution (with success or with an error). We need to manage them because of the asynchronous nature of the task; as we've seen for similar scenarios, we need to use a deferral object (which type, in this case, is **BackgroundTaskDeferral**) to notify the operating system when the operation is completed.

Audio background tasks work in a different way than standard background tasks. As we will see in Chapter 11, a regular background task typically executes the **Run()** method, and is then disposed by the operating system. Consequently, the **BackgroundTaskDeferral** object is created when the **Run()** method starts, and is marked as completed when the method is completed. However, background audio tasks are always kept alive until there's a media file to play. Consequently, we call the **GetDeferral()** method in the **Run()** block, but we invoke the **Complete()** method only when the background task is actually disposed, which happens when the **Completed** or **Canceled** events are triggered.

It's time to go back to the real background task implementation. Let's see, in detail, how to manage the **ButtonPressed** event of the **SystemMediaTransportControls**.

```
void transportControls_ButtonPressed(SystemMediaTransportControls sender,
SystemMediaTransportControlsButtonPressedEventArgs args)
{
    switch (args.Button)
    {
        case SystemMediaTransportControlsButton.Play:
            BackgroundMediaPlayer.Current.Play();
            break;
        case SystemMediaTransportControlsButton.Pause:
            BackgroundMediaPlayer.Current.Pause();
            break;
    }
}
```

The code is similar to the one we've seen for Windows; by using the **Button** property of the method's parameter, we are able to understand which button has been pressed. According to this information, we trigger the appropriate method of the **BackgroundMediaPlayer** class. The previous sample shows how to manage the **Play** and **Pause** buttons.

Another important event we've subscribed in the **Run()** method is **MessageReceivedFromForeground**, which you should already know. It's triggered when the main application sends a message to the background task because the user wants to change the playback status.

```
void BackgroundMediaPlayer_MessageReceivedFromForeground(object sender,
MediaPlayerDataReceivedEventArgs args)
{
    ValueSet valueSet = args.Data;
    foreach (string key in valueSet.Keys)
    {
        switch (key)
        {
            case "Play":
                Play(valueSet[key].ToString());
                break;
```

```
            case "Pause":
                BackgroundMediaPlayer.Current.Pause();
                break;
            case "Resume":
                BackgroundMediaPlayer.Current.Play();
                break;
        }
    }
}
```

Thanks to the **Data** property, we are able to retrieve the message, which type is **ValueSet**. Then we can iterate the **Keys** collection to understand which command has been received. According to the content of the message, we perform a different operation with the **BackgroundMediaPlayer** class. What happens when we want to play a track? In this case, in addition to the key, we also need to retrieve the content of the message, which is the URL of the file to reproduce. The following sample shows how to define the **Play()** method:

```
private void Play(string url)
{
    BackgroundMediaPlayer.Current.SetUriSource(new Uri(url));
    BackgroundMediaPlayer.Current.Play();
    transportControls.DisplayUpdater.Type = MediaPlaybackType.Music;
    transportControls.DisplayUpdater.MusicProperties.Title = "WPDev Fusion
1";
    transportControls.DisplayUpdater.Update();
    ValueSet info = new ValueSet();
    info.Add("title", "WPDev Fusion 1");
    BackgroundMediaPlayer.SendMessageToForeground(info);
}
```

The track is set by using the **SetUriSource()** method of the **BackgroundMediaPlayer** class, passing, as parameter, the URL (which is the content of the message we've received). Then, we can simply call the **Play()** method. In the previous sample, you can also see that we are able to set what information to display in the background audio player, like we did in Windows, by using the **DisplayUpdater** property of the **SystemMediaTransportControls** class. This time, in addition to updating the background player, we also send the information about the title of the current track to the main application, by sending a new message with the **SendMessageToForeground()** method. This way, if the application is running in the foreground, it will be able to intercept it and display the title of the new track to the user.

The last piece of code to analyze is the **CurrentStateChanged** event handler, which is invoked every time the playback state changes. We're going to use it in the same way we did for Windows: to synchronize the playback state with the background audio player state, so that they always match.

```
private void Current_CurrentStateChanged(MediaPlayer sender, object args)
{
```

```
    if (sender.CurrentState == MediaPlayerState.Playing)
    {
        transportControls.PlaybackStatus = MediaPlaybackStatus.Playing;
    }
    else if (sender.CurrentState == MediaPlayerState.Paused)
    {
        transportControls.PlaybackStatus = MediaPlaybackStatus.Paused;
    }
}
```

We simply the detect the current state, using the **`CurrentState`** property of the **`BackgroundMediaPlayer`** class, and we set the **`PlaybackStatus`** property of the **`SystemMediaTransportControls`** accordingly.

Acquiring photos and videos

Mobile devices, especially smartphones, are the devices most often used to capture photos and video. Many factors helped the adoption rate of smartphones for these scenarios:

- Internet connection: The user is immediately able to share their photo with friends on social networks, etc.
- Applications: The user can edit the photo immediately, by adding effects, fixing imperfections, etc.
- Camera evolution: In the beginning, the integrated camera was very poor compared to standard photo cameras, but things have dramatically changed in the latest years. Phones like the Lumia 1020 can completely replace a compact camera.

The Windows Runtime offers a set of APIs that, as developers, we can use to integrate the camera experience into our application. Let's see them in detail.

Capturing photos and videos using the native Windows application

The easiest way to capture a photo or a video is by relying on the native Camera application included in Windows. It's the best solution when acquiring a photo or a video isn't the core feature of the application, but a complementary one, like a Twitter application that allows users to take a photo and share it with their followers. In fact, by using this feature, you won't have to deal with all the complexity required to embed the video streaming and to manage the process. You will just defer the operation to the Camera application, and you'll simply get the photo or video in return. On the other side, as we're going to see later in this chapter, this approach is less powerful than managing the capture phase directly in your application. It's important to highlight that the following APIs are available only on Windows, and are not supported by Windows Phone.

To use this feature, we need to use a class called **CameraCaptureUI**, which belongs to the **Windows.Media.Capture** namespace. To properly use it, we need to enable the **Webcam** capability in the manifest file. Additionally, if we want to record a video with audio, we need to enable the **Microphone** capability.

Capturing a photo

Before capturing a photo, we need to configure some settings to customize the operation, which can be accessed using the **PhotoSettings** property of the **CameraCaptureUI** class. The most important ones are:

- **Format**, which is the file type we want to use to save the photo. It can be **Jpg** or **Png**, and it's set using one of the values of the **CameraCaptureUIPhotoFormat** enumerator.
- **MaxResolution**, which is the resolution to use to acquire the photo. It's set using the **CameraCaptureUIMaxPhotoResolution** enumerator, which accepts a range of values from **VerySmallQvga** (very low quality) to **HighestAvailable** (the best supported quality).

Once you've defined the capture settings, you can start the capturing process by calling the **CaptureFileAsync()** method. It requires as parameter the kind of media we want to capture (since we're talking about photos, we need to pass **CameraCaptureUIMode.Photo** as value). This method will open the Camera application so that the user can take the picture. The captured photo will be passed back to your application as a **StorageFile** object. The following sample shows how to use these APIs to capture a photo and display it to the user with an **image** control:

```
private async void OnTakePictureClicked(object sender, RoutedEventArgs e)
{
    CameraCaptureUI capture = new CameraCaptureUI();
    capture.PhotoSettings.Format = CameraCaptureUIPhotoFormat.Jpeg;
    capture.PhotoSettings.MaxResolution = CameraCaptureUIMaxPhotoResolution.
    HighestAvailable;
    StorageFile file = await
capture.CaptureFileAsync(CameraCaptureUIMode.Photo);
    if (file != null)
    {
        var stream = await file.OpenReadAsync();
        BitmapImage image = new BitmapImage();
        await image.SetSourceAsync(stream);
        CapturedImage.Source = image;
    }
}
```

Capturing a video

The code needed to capture a video is very similar to the one we've just seen. The main difference is that, if we want to customize the capture settings, we need to interact with the **VideoSettings** property offered by the **CameraCaptureUI** class. Also in this scenario, we can customize the recording format (which can be MP4 or WMV, and is set by using one of the values of the **CameraCaptureUIVideoFormat** enumerator) and the resolution.

Again, to launch the Camera application, we need to call the **CaptureFileAsync()** method, passing as parameter, this time, the value **CameraCaptureUIMode.Video**. What we get in return is the captured video as a **StorageFile** object. The following sample shows how to record a video and to display it to the user using a **MediaElement** control:

```
private async void OnRecordVideoClicked(object sender, RoutedEventArgs e)
{
    CameraCaptureUI capture = new CameraCaptureUI();
    capture.VideoSettings.Format = CameraCaptureUIVideoFormat.Mp4;
    capture.VideoSettings.MaxResolution = CameraCaptureUIMaxVideoResolution.
    HighestAvailable;
    StorageFile file = await
capture.CaptureFileAsync(CameraCaptureUIMode.Video);
    if (file != null)
    {
        var stream = await file.OpenReadAsync();
        RecordedVideo.SetSource(stream, stream.ContentType);
    }
}
```

Acquiring photos and videos within the application

For some applications, camera integration is a core feature. Think, for example, of Instagram, which allows users to apply filters to the photos before sharing them. For these scenarios, the Windows Runtime offers a set of powerful APIs (which, unlike the previous ones, are available on Windows Phone) that give the developer maximum flexibility. The photo or video isn't captured by another application, but by the app itself, which will have to take care of displaying the camera stream and capturing the image.

The base class to implement this scenario is called **MediaCapture**, and it's connected to a special XAML control that displays the camera stream directly inside the application. The control is called **CaptureElement**, and it can be easily included in the XAML, like in the following sample:

```
<CaptureElement x:Name="PreviewStream" />
```

By combining the **CaptureElement** control and the **MediaCapture** class, we are able to display the camera preview in our application with the following code:

```
public sealed partial class MainPage : Page
{
    private MediaCapture capture;

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

    private async void OnInitializeCameraClicked(object sender,
RoutedEventArgs e)
    {
        capture = new MediaCapture();
        await capture.InitializeAsync();
        PreviewStream.Source = capture;
        await capture.StartPreviewAsync();
    }
}
```

After we have created a new **MediaCapture** object, we initialize it by calling the **InitializeAsync()** method. Then, we can assign it as **Source** of the **MediaCapture** element we've included in our page. Now we are able to display the preview by calling the **StartPreviewAsync()** method.

However, the previous code is incomplete; it doesn't take into account that the application can run on a phone with two cameras, one on the back and one on the front. To support this scenario, the **InitializeAsync()** method also supports a parameter, which type is **MediaCaptureInitializationSettings**. Among the settings we can customize, we can define which camera to use. Here is a more complete initialization of the preview stream:

```
private async void OnInitializeCameraClicked(object sender, RoutedEventArgs
e)
{
    MediaCapture capture = new MediaCapture();
    DeviceInformationCollection collection = await DeviceInformation.
        FindAllAsync(DeviceClass.VideoCapture);
    MediaCaptureInitializationSettings settings = new
MediaCaptureInitializationSettings();
    if (collection.Count > 1)
    {
        settings.VideoDeviceId = collection[1].Id;
    }
    else
    {
        settings.VideoDeviceId = collection[0].Id;
```

```
    }
    await capture.InitializeAsync(settings);
    PreviewStream.Source = capture;
    await capture.StartPreviewAsync();
}
```

The first step is to query the operating system to detect how many cameras are available. We do it by calling the **FindAllAsync()** method of the **DeviceInformation** class, specifying that we are interested in the **VideoCapture** devices (which is one of the values of the **DeviceClass** enumerator). This method returns a collection with all the available cameras, each with its unique id, which is stored in the **Id** property. This is the identifier that we need to assign to the **VideoDeviceId** property of the **MediaCaptureInitializationSettings** class. The previous code tries to use the front-facing camera, by checking how many cameras the device has. If there's more than one, there's a front camera we can use; otherwise, we fall back on the default camera, which is the rear one. After we've properly defined the **VideoDeviceId** property, we can pass the **MediaCaptureInitializationSettings** object we've created to the **InitializeAsync()** method.

The code we've seen so far is the same, whether we want to take a photo or to record a video. From now on, however, we're going to use different methods, according to our scenario.

Capturing a photo

Photos are taken using the **CapturePhotoToStorageFileAsync()** method offered by the **MediaCapture** class. As parameters, it requires the file format we want to use (thanks to the **ImageEncodingProperties** class) and the file where we want to save the captured image (represented with a **StorageFile** object). The following sample shows how to take a picture and save it in a file called **image.jpg** in the local storage:

```
private async void OnTakePhotoClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await ApplicationData.Current.LocalFolder.
    CreateFileAsync("image.jpg", CreationCollisionOption.ReplaceExisting);
    ImageEncodingProperties properties =
ImageEncodingProperties.CreateJpeg();
    await capture.CapturePhotoToStorageFileAsync(properties, file);
}
```

The **ImageEncodingProperties** class offers different methods to create the photo with many formats. The previous sample uses the **CreateJpeg()** method to create a JPEG file; however, we could have used **CreateBmp()** to create a bitmap file, **CreatePng()** to create a PNG file, or **CreateUncompressed()** to acquire a raw image.

Capturing a video

The procedure to capture a video is very similar to the one we've seen for taking pictures. The difference is that, this time, we're going to use the **StartRecordToStorageFileAsync()** method of the **MediaCapture** class. As parameters, other than the usual **StorageFile** object, which identifies the file in which to save the content, it requires a **MediaEncodingProfile** object that defines the format and the quality to use.

```
private async void OnRecordVideoClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("video.mp4",
CreationCollisionOption.ReplaceExisting);
    MediaEncodingProfile profile =
MediaEncodingProfile.CreateMp4(VideoEncodingQuality.
    HD720p);
    await capture.StartRecordToStorageFileAsync(profile, file);
}
```

Also in this case, the **MediaEncodingProfile** class offers many methods to work with the most common video types. The previous example uses the **CreateMp4()** method to create a MP4 video, but we could have used **CreateWmv()** to create a WMV file, or **CreateAVI()** to create an AVI file. All these methods also require a parameter, which type is **VideoEncodingQuality**, to define the recording quality; in the previous sample, the resolution of the captured video will be 720p.

How to customize the capture process

One of the most important features of the **MediaCapture** class is the ability to support a deep customization of the capture parameters, like the contrast, the brightness, and the exposure time. We achieve this result by using the **VideoDeviceController** object, which is offered by the **MediaCapture** class. Some of the most important properties that we can customize are:

- **Brightness**
- **Contrast**
- **Exposure**
- **Focus**
- **FlashControl**, to enable or disable the flash

The following sample shows how to disable the flash, by modifying the **FlashControl** property of the **VideoDeviceController** object:

```
private async void OnTakePhotoClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("image.jpg",
CreationCollisionOption.ReplaceExisting);
    ImageEncodingProperties properties =
```

```
ImageEncodingProperties.CreateJpeg();
    capture.VideoDeviceController.FlashControl.Enabled = false;
    await capture.CapturePhotoToStorageFileAsync(properties, file);
}
```

In case of a video recording, you can also customize the audio recording parameters, by using the **AudioDeviceController**. The following sample shows how to change the recording volume level, by interacting with the **VolumePercent** property:

```
private async void OnRecordVideoClicked(object sender, RoutedEventArgs e)
{
    StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync("video.mp4",
CreationCollisionOption.ReplaceExisting);
    MediaEncodingProfile profile =
MediaEncodingProfile.CreateMp4(VideoEncodingQuality.
    HD720p);
    capture.AudioDeviceController.VolumePercent = 30;
    await capture.StartRecordToStorageFileAsync(profile, file);
}
```

Accessing the multimedia library

Every Windows and Windows Phone device has a set of multimedia libraries, where users can store their media content (music, videos, photos, etc.). Thanks to the APIs offered by the Windows Runtime, we are able to access these libraries and the stored files. The main difference between the APIs we're going to see in this section and the **FileOpenPicker** class we talked about in Chapter 5 is the access mode. Unlike with the **FileOpenPicker**, we'll be able to access any media file stored in these libraries, without having to ask to the user to pick the ones she wants to import into the application.

The media libraries are accessed by the same class we saw in Chapter 5 for removable devices, which is **KnownFolders**. This class offers the following objects:

- **PicturesLibrary**, to access the pictures library.
- **CameraRoll**, to access the camera roll where photos taken with the device are automatically saved.
- **VideosLibrary**, to access the videos library.
- **MusicLibrary**, to access the music library.

To access to these libraries, you'll have to enable a set of capabilities in the manifest file; there's a capability for each supported library.

Working with libraries is very easy if you've read Chapter 5; every library is mapped with a **StorageFolder** object, so you'll be able to read, write, and copy files in the same way you do with the local storage. The following sample shows how to list all the images that are saved in the user's pictures library, and to display their names to the user:

```
private async void OnGetImagesClicked(object sender, RoutedEventArgs e)
{
    var files = await KnownFolders.PicturesLibrary.GetFilesAsync();
    foreach (StorageFile file in files)
    {
        MessageDialog dialog = new MessageDialog(file.DisplayName);
        await dialog.ShowAsync();
    }
}
```

The following sample, instead, shows how to take a picture stored in the local storage (called **image.png**) and save it in the pictures library, using the **CopyAsync()** method:

```
private async void OnSaveImageClicked(object sender, RoutedEventArgs e)
{
    var file = await
ApplicationData.Current.LocalFolder.GetFileAsync("image.png");
    await file.CopyAsync(KnownFolders.PicturesLibrary);
}
```

There's an important concept to understand when working with libraries: a library doesn't refer to a specific file type, but to a specific device location. The difference is clear when it comes to the camera roll. As default behavior, every photo or video acquired with the device is stored into the Camera Roll folder. Consequently, if you try to retrieve all the video files using the **KnownFolders.VideosLibrary** class, you won't get all the videos acquired with the device, because they aren't stored in the video folder, but in the camera roll folder. This means that, for example, if you want to retrieve only the acquired video, you'll have to interact with the class **KnownFolders.CameraRoll** and use the **FileType** property of the **StorageFile** class to identify the file type, like in the following sample:

```
private async void OnGetVideosClicked(object sender, RoutedEventArgs e)
{
    StorageFolder library = KnownFolders.CameraRoll;
    IReadOnlyList<StorageFile> items = await library.GetFilesAsync();
    List<StorageFile> videos = items.Where(x => x.FileType ==
".mp4").ToList();
    string message = string.Format("There are {0} videos in your camera
roll", videos.Count);
    MessageDialog dialog = new MessageDialog(message);
    await dialog.ShowAsync();
```

```
}
```

Querying the multimedia libraries

Because libraries refer to a device location and not to a file's type, the various APIs that we've used to interact with the files follow the structure of the folder, and not the logical structure. Let's see an example to better understand this concept. Let's say that, in your Music library, you have many albums, each of them stored in a subfolder. If you call the **`GetFilesAsync()`** method of the **`KnownLibrary.MusicLibrary`** object, you'll get in return no results; since the method considers the structure of the folder, it isn't able to retrieve all the music files that are stored inside the various subfolders.

This approach can be complex to manage in some scenarios (for example, you just need to retrieve all the music files of the library, no matter how deep the folder structure is). Consequently, the **`GetFilesAsync()`** method accepts a set of parameters that are able to change the method's behavior and perform more complex operations. These parameters can be one of the values of the **`CommonFileQuery`** enumerator, and they are used to return a set of files in different ways. The following sample uses the **`OrderByName`** value to return all the music tracks stored in the music library (no matter which subfolder they are stored in), ordered by the track name:

```
private async void OnGetSongsClicked(object sender, RoutedEventArgs e)
{
    StorageFolder library = KnownFolders.MusicLibrary;
    IReadOnlyList<StorageFile> items = await
library.GetFilesAsync(CommonFileQuery.OrderByName);

    string message = string.Format("There are {0} music tracks in your
library", items.Count);
    MessageDialog dialog = new MessageDialog(message);
    await dialog.ShowAsync();
}
```

The only logical abstraction that is automatically provided by the operating system is about the SD card, especially on Windows Phone. If you have some music tracks stored on your device, and some stored in the SD card, the **`GetFilesAsync()`** method of the **`KnownFolders.MusicLibrary`** object will return all of them, since the operating system is able to consider the removable memory as part of the native library.

How to retrieve metadata

Multimedia files have an important difference from the standard files: they can have a set of metadata that describes the media's content. For example, a music track can have metadata that describes the title, author, and album; a picture can have EXIF data that describes the camera model, the resolution, etc.

The Windows Runtime offers a way to retrieve this special properties, by offering a set of methods (which are exposed by the **Properties** class of the **StorageFile** object) for each media type:

- **GetImagePropertiesAsync()** for pictures
- **GetMusicPropertiesAsync()** for music
- **GetVideoPropertiesAsync()** for videos

Every method returns a different kind of object, with different properties. With music, for example, you'll get in return a **MusicProperties** object, which offers properties like **Artist** or **Title**. For pictures, you'll get in return an **ImageProperties** object, with properties like **Width** or **Height**. The following sample code shows how to retrieve the EXIF data of the first picture in the camera roll, so that we can display the title and resolution to the user with a pop-up message:

```
private async void OnGetFilePropertiesClicked(object sender, RoutedEventArgs
e)
{
    StorageFolder library = KnownFolders.CameraRoll;
    IReadOnlyList<StorageFile> pictures = await library.GetFilesAsync();
    StorageFile picture = pictures.FirstOrDefault();
    if (picture != null)
    {
        ImageProperties imageProperties = await picture.Properties.
        GetImagePropertiesAsync();
        if (imageProperties != null)
        {
            string info = string.Format("{0} - {1} x {2}",
imageProperties.Title, imageProperties.Width, imageProperties.Height);
            MessageDialog dialog = new MessageDialog(info);
            await dialog.ShowAsync();
        }
    }
}
```

How to retrieve a thumbnail

Often, when you work with images or videos, you'll use thumbnails to improve the application's performance. For example, if you need to display a list of pictures, it wouldn't make sense to use the full-resolution image just to display a small preview of the image. The **StorageFile** class offers a method to automatically retrieve the thumbnail from a picture, video, or audio file (for audio, it's available only in Windows). The method is called **GetThumbnailAsync()**, and it requires, as parameter, one of the values of the **ThumbnailMode** enumerator, which is used to specify the kind of thumbnail we want to generate. For example, we can use the **ListView** parameter if the thumbnail is required for a list, or **SingleItem** if we need a to display a bigger preview. You can also pass a fixed number as parameter, which will be used as the width of the thumbnail to generate. The following sample generates a thumbnail from the first picture available in the device's camera roll:

```
private async void OnGetFilePropertiesClicked(object sender, RoutedEventArgs
e)
{
    StorageFolder library = KnownFolders.CameraRoll;
    IReadOnlyList<StorageFile> files = await library.GetFilesAsync();
    StorageFile picture = files.FirstOrDefault();
    if (picture != null)
    {
        StorageItemThumbnail thumbnail = await
picture.GetThumbnailAsync(ThumbnailMode.ListView);
        BitmapImage image = new BitmapImage();
        await image.SetSourceAsync(thumbnail);
        Thumbnail.Source = image;
    }
}
```

The resulting thumbnail, which is stored in a **StorageItemThumbnail** object, is displayed to the user with an **Image** control.

Chapter 6 Tiles and Notifications

Tiles are certainly the most distinctive visual feature of Windows and Windows Phone. Tiles are much more than quick shortcuts that are placed on the Start Screen to open the application; they can also deliver information and content without requiring the user to launch the app. Tiles are tightly connected to notifications, which are the most-used mechanism for interacting with the user when the application is not running. In this chapter, we'll see all the notification types supported by Windows and Windows Phone, and how to send them.

The notification types

The Windows Runtime offers four kind of notifications:

- **Toast notifications** are the most visible, since they display a banner at the top of the screen. They can also play a sound and trigger a device vibration.
- **Tile notifications** are used to update the content of a tile. They are non-intrusive, since they don't trigger any alert; the tile is silently updated with the new information.
- **Badge notifications** are used in combination with tiles, and instead of changing the whole tile content, they simply display a symbol or a number to capture the user's attention.
- **Raw notifications** don't have a fixed structure, and can contain any type of content.

Regardless of the type we're going to use, all the notifications (except the raw ones) are represented using the XML format, as we'll see later in this chapter. We'll also talk about push notifications, which make it much easier for a server application (like a website or a web service) to send a notification. In addition, the Windows Runtime is able to automatically adapt the notification's content to the platform guidelines. For example, to send a toast notification, we'll use the same exact content, even if it will be displayed in a different way in Windows and in Windows Phone.

If we exclude raw notifications (which can be sent only by a remote server), we can send notifications in four different ways:

- **From the application itself:** If the application content changes while the user is using it, we can immediately update the tile from code.
- **With a background task:** We'll see how they work in detail in Chapter 11.
- **Periodic:** The application periodically pulls an XML file published on Internet with the notification's content, and updates it.
- **Scheduled:** The application is able to schedule a set of notifications, which are sent at the exact date and time we've specified.
- **Push notifications:** The notification is generated by an external service (like a web application) and sent to the device using a service provided by Microsoft called WNS (Windows Notification Service).

Toast notifications

As we've already mentioned, toast notifications are the easiest for the user to see, so you should avoid abusing them. The following image shows how toast notifications are rendered in a different way on both platforms:

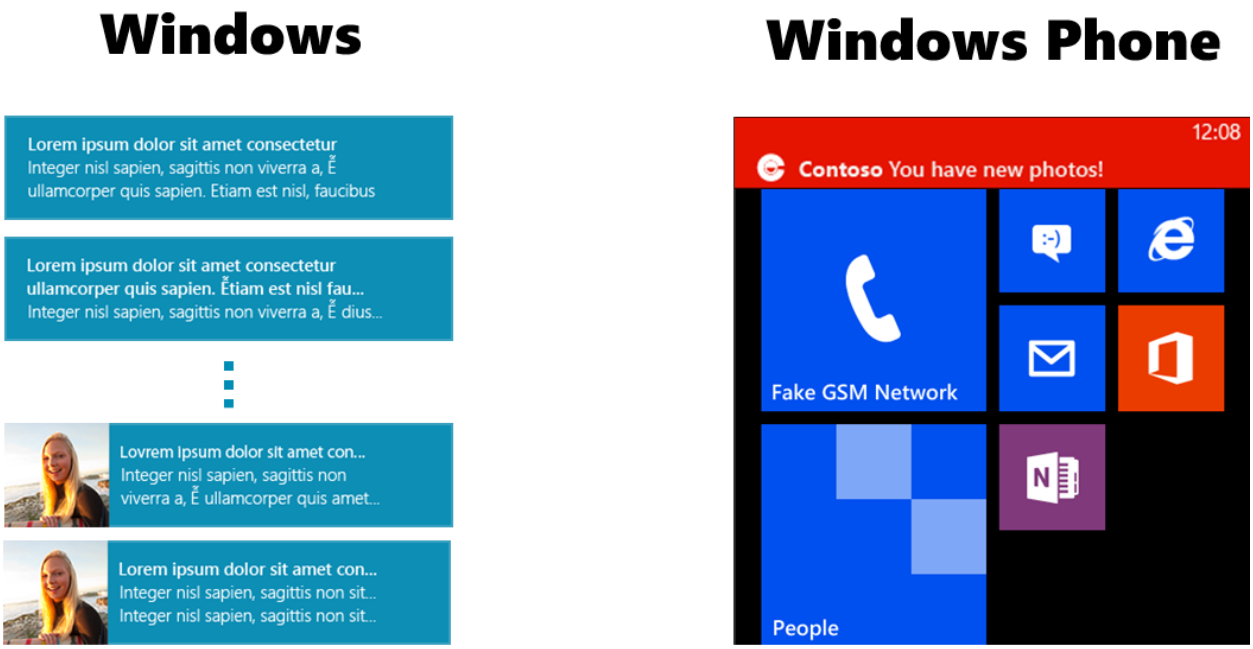


Figure 17: The different ways toast notifications are displayed on Windows and Windows Phone

The first step to receive toast notifications is to enable them in the manifest file. In the **Application** section, under the **Notifications** category, we'll find a dropdown menu labeled **Toast capable**. We need to set it to **Yes** before starting to work with the code.

There are eight templates that can be used to send a toast notifications, and they are all documented in the MSDN documentation at http://s.qmatteoq.com/ToastCatalog. The various templates support different content types: text, images, etc. However, most of these templates work only with Windows. On Windows Phone, we can use only one template, which is composed of the application's logo, a title, and a text (as seen in the previous figure, on the right). You can find this template in the documentation with the name **ToastText02**. However, we can use any template we want on Windows Phone; the information that isn't supported by the platform (like the image) will simply be ignored.

The following sample shows the XML structure of a notification, which template is **ToastImageAndText02:**

```
<toast>
  <visual>
    <binding template="ToastImageAndText02">
      <image id="1" src="image1" alt="image1"/>
```

```
      <text id="1">headlineText</text>
      <text id="2">bodyText</text>
    </binding>
  </visual>
</toast>
```

The main node is called **binding**, and it has a **template** attribute that describes the name of the used template. Inside the **binding** node, we find an element for each property supported by the notification. In this case, we have one **image** element (for the image) and two **text** elements (for the text that is displayed next to the image). The following sample shows how to generate, in code, a toast notification using the previous template:

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    XmlDocument template =
ToastNotificationManager.GetTemplateContent(ToastTemplateType.ToastImageAndTe
xt02);
    XmlNodeList texts = template.GetElementsByTagName("text");
    texts[0].AppendChild(template.CreateTextNode("Title"));
    texts[1].AppendChild(template.CreateTextNode("Text"));
    XmlNodeList images = template.GetElementsByTagName("image");
    ((XmlElement)images[0]).SetAttribute("src", "ms-
appx:///Assets/wplogo.png");
    ToastNotification notification = new ToastNotification(template);
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
}
```

As already mentioned at the beginning of the chapter, every notification is represented using the XML format. Consequently, you'll need to define them in code by manually manipulating the XML structure. However, you won't have to manually create the whole XML that defines the notification. The Windows Runtime offers a class called **ToastNotificationManager**, which, using the **GetTemplateContent()** method, can retrieve the XML structure of a notification. The method requires, as a parameter, one of the values of the **ToastTemplateType** enumerator; each value matches one of the templates that is described in the MSDN documentation.

What we get in return is an **XmlDocument** object with the XML structure. Using the APIs provided by the Windows Runtime to interact with XML documents, we are able to:

- Change the text displayed in the notification, by retrieving a reference to all the **text** elements with the **GetElementsByTagName()** method. Thanks to the **AppendChild()** method, we are able to change the element's values and specify the notification's title and text.
- Change the image displayed in the notification, by again using the **GetElementsByTagName()** method, and by retrieving a reference to the **image** element. As with the HTML image element, we need to change the **src** property of the element to define the path of the image to use. In the previous sample, since we're using the **ms-appx:///** prefix, we're loading an image from the Visual Studio project.

Once we've defined the XML layout, we can create the notification by using the **ToastNotification** class, which requires as parameter the XML document we've just customized. In the end, we need to create a **ToastNotifier** object, which will take care of sending the notification. To create it, we need to use the **CreateToastNotifier()** method of the **ToastNotificationManager** class. In the end, we can show the notification simply by calling the **Show()** method of the **ToastNotifier** class and passing, as parameter, the **ToastNotification** object we've previously created.

Managing the activation from a toast notification

On both Windows and Windows Phone, the user is able to interact with the toast notification; when they tap on it, the application that generated it is automatically opened. However, we can also customize this behavior. Instead of simply opening the application, we can pass a set of activation parameters, which we can retrieve to understand the notification's context and redirect the user to a specific page.

To support this scenario, we need to add, to the main toast element in the XML definition, a new attribute called **launch** with the activation parameters, like in the following sample:

```
IXmlNode toastNode = template.SelectSingleNode("/toast");
((XmlElement)toastNode).SetAttribute("launch", "test");
```

In the previous sample, we're simply passing **test** as activation parameter, but we could have passed any kind of textual information. This time, when the app is opened by the toast notification, we'll get a reference to the parameter in the **OnLaunched()** event of the **App** class. The following sample shows how to manage it:

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation context and navigate to
the first page
        rootFrame = new Frame();
        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            // TODO: Load state from previously suspended application
        }
        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }
    if (!string.IsNullOrEmpty(e.Arguments))
    {
        rootFrame.Navigate(typeof(DetailPage), e.Arguments);
    }
    if (rootFrame.Content == null)
```

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

As you can see, compared to the standard **`OnLaunched()`** method, we've added a new check: if the **`Arguments`** property of the method's parameter isn't empty, it means that the application has been launched by a toast notification. In this case, instead of redirecting the user to the main page, we'll force the navigation directly to another page (in the previous sample, it's called **`DetailPage`**) and we pass, as navigation parameter, the one we've received from the notification. This way, we'll be able to directly load and display to the user the content connected to the toast notification that was displayed.

Sending a toast notification with the NotificationsExtensions library

It's important to understand the XML format that defines a notification, but when you're working with them in code, the operation isn't as straightforward as we've just seen. In all the previous code samples, we were required to manually manipulate the XML document to set the tile's content. To simplify the developer's work, you can find a library on NuGet called **NotificationsExtensions**: http://www.nuget.org/packages/NotificationsExtensions.UniversalApps.

The library acts as a wrapper to the XML document; you'll be able to define a toast notification by using a more familiar approach based on classes and properties. The following sample shows how to send the same notification we've seen before with this library:

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    template.TextHeading.Text = "Title";
    template.TextBodyWrap.Text = "Text";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    template.Launch = "test";
    ToastNotification notification = template.CreateNotification();
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
}
```

The library offers a class called **ToastContentFactory**, which is able to create the notification template: we can find many methods, one for each supported template. In this case, since we're using the template called **ToastImageAndText02**, we use the **CreateToastImageAndText02()** method. In return, we get an object that offers a way to customize the notification content simply by changing the values of its properties. In the previous sample, we set:

- The title, using the **TextHeading** property.
- The text, using the **TextBodyWrap** property.
- The image, using the **Image** property.
- The launch parameters, using the **Launch** property.

In the end, we transform the template into a **ToastNotification** object, by calling the **CreateNotification()** method. From this point, the code is the same as we've seen before: we create a new **ToastNotifier** object using the **ToastNotificationManager** class, and we pass the notification to the **Show()** method.

Scheduling a toast notification

Another option to send toast notifications is to schedule them. By using the **ScheduledToastNotification** class, we'll be able to define and send a notification in the same way we've seen before, with the difference that, this time, we'll have the chance to specify the date and time when the notification will be displayed.

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    if (template != null)
    {
        template.TextHeading.Text = "Title";
        template.TextBodyWrap.Text = "Text";
        template.Image.Src = "ms-appx:///Assets/wplogo.png";
        template.Launch = "test";
        XmlDocument xml = template.GetXml();
        ToastNotifier notifier =
ToastNotificationManager.CreateToastNotifier();
        ScheduledToastNotification scheduledToast = new
ScheduledToastNotification(xml, DateTimeOffset.Now.AddSeconds(10));
        notifier.AddToSchedule(scheduledToast);
    }
}
```

The previous sample uses the **NotificationsExtensions** library we've previously seen. The difference is that we create a new **ScheduledToastNotification** object, which requires as parameter the XML that defines the notification (instead of a **ToastNotification** object). For this purpose, we're going to use the **GetXml()** method offered by the template, which will return the XML file. The other parameter required by the **ScheduledToastNotification** constructor is the date and time to show the notification, represented by a **DateTimeOffset** object. In the previous sample, we schedule it 10 seconds later than the current time. In the end, we add the notification to the scheduler by calling the **AddToSchedule()** method of the **ToastNotifier** class.

The **ToastNotifier** class offers a way to retrieve the list of scheduled notifications, by using the **GetScheduledToastNotifications()** method. We can also remove already scheduled notifications by using the **RemoveFromSchedule()** method. The following sample shows how to remove all the notifications that have been scheduled by the app:

```
private void OnRemoveNotifications(object sender, RoutedEventArgs e)
{
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    IReadOnlyList<ScheduledToastNotification> notifications =
notifier.GetScheduledToastNotifications();
    foreach (ScheduledToastNotification notification in notifications)
    {
        notifier.RemoveFromSchedule(notification);
    }
}
```

Toast notifications and the Windows Phone's Action Center

Windows Phone 8.1 has added a new feature called Action Center, which is activated by dragging the finger downward from the top of the screen. Action Center acts as a notification center; all the toast notifications received by any application will be collected in this area. This way, if the user has missed one or more notifications because they weren't watching their phone, they will be able to recover them.

As developers, we can directly interact with the Action Center by manipulating the notifications created by our application. It's important to highlight that the following APIs are available only on Windows Phone, since Windows 8.1 doesn't offer an Action Center.

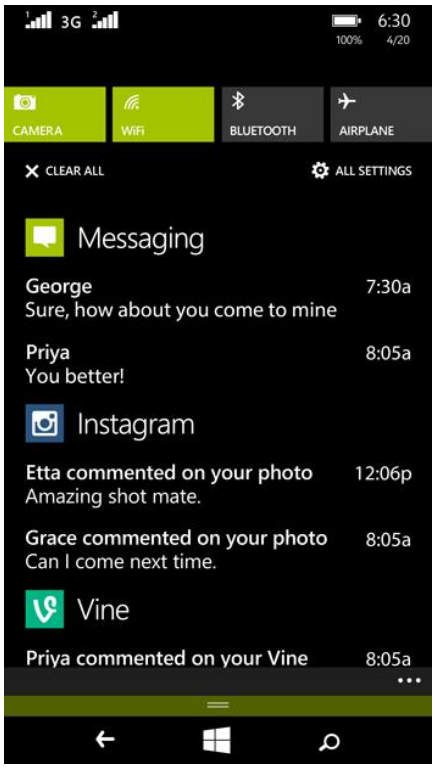


Figure 18: The Action Center in Windows Phone

Sending a silent notification

Silent notifications are toast notifications that aren't intrusive; they don't play any sound, and they don't trigger the usual banner. They are simply stored in the Action Center. The user will notice them by the notification icon that is displayed in the Status Bar. To send a silent notification, we use the **`SuppressPopup`** property of the **`ToastNotification`** class, like in the following sample:

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    template.TextHeading.Text = "Title";
    template.TextBodyWrap.Text = "Text";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    ToastNotification notification = template.CreateNotification();
    notification.SuppressPopup = true;
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
}
```

Setting an expiration date

As default behavior, toast notifications are removed from the Action Center when the user has tapped on them, or after seven days from when they are created. However, we can set a specific expiration date and time, thanks to the **ExpirationDate** property of the **ToastNotification** class. The following sample shows how to send a notification that expires after five seconds:

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    template.TextHeading.Text = "Title";
    template.TextBodyWrap.Text = "Text";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    ToastNotification notification = template.CreateNotification();
    notification.ExpirationTime = DateTimeOffset.Now.AddSeconds(5);
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
}
```

Deleting or replacing a notification

The Windows Runtime on Windows Phone offers a class called **ToastNotificationHistory** that provides access to all the toast notifications generated by our application. Thanks to this class, we are able to delete a notification, delete a group of notifications, or replace an already existing notification.

This scenario requires the developer to find a way to univocally identify a notification. We can do this by using two properties offered by the **ToastNotification** class:

- **Tag**: a string that univocally identifies a notification.
- **Group**: a string that univocally identifies a group of notifications.

Let's analyze the following sample, which generates two toast notifications:

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    template.TextHeading.Text = "Title";
    template.TextBodyWrap.Text = "Text";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    ToastNotification notification = template.CreateNotification();
    notification.Tag = "Notification";
    notification.Group = "MyNotifications";
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
```

```
    IToastImageAndText02 template2 =
ToastContentFactory.CreateToastImageAndText02();
    template2.TextHeading.Text = "Title 2";
    template2.TextBodyWrap.Text = "Text 2";
    template2.Image.Src = "ms-appx:///Assets/wplogo.png";
    ToastNotification notification2 = template.CreateNotification();
    notification.Tag = "Notification 2";
    notification.Group = "MyNotifications";
    notifier.Show(notification2);
}
```

As you can see, we've assigned a different **Tag** to the two notifications, but the same **Group** name. Thanks to these properties, we'll be able to interact with these notifications using the **ToastNotificationHistory** class. For example, we can use the **Remove()** method to remove a single notification, passing its tag as parameter:

```
private void OnRemoveNotificationClicked(object sender, RoutedEventArgs e)
{
    ToastNotificationHistory history = ToastNotificationManager.History;
    history.Remove("Notification");
}
```

Another option is to delete all the notifications that belong to the same group. The following sample shows how to remove all the notifications that belong to a group called **MyNotifications**, which is passed as parameter to the **RemoveGroup()** method:

```
private void OnRemoveGroupClicked(object sender, RoutedEventArgs e)
{
    ToastNotificationHistory history = ToastNotificationManager.History;
    history.RemoveGroup("MyNotifications");
}
```

In the end, we can also replace an already existing notification. We create a new **ToastNotification** object, change the content we want to modify, and then send it again, by assigning the same **Tag** and **Group** value of an already existing notification.

```
private void OnSendToastClicked(object sender, RoutedEventArgs e)
{
    IToastImageAndText02 template =
ToastContentFactory.CreateToastImageAndText02();
    template.TextHeading.Text = "Updated title";
    template.TextBodyWrap.Text = "Update text";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    ToastNotification notification = template.CreateNotification();
    notification.Tag = "Notification";
```

```
    notification.Group = "MyNotification";
    ToastNotifier notifier = ToastNotificationManager.CreateToastNotifier();
    notifier.Show(notification);
}
```

We can also remove all the notifications that belong to our application, simply by calling the **`Clear()`** method of the **`ToastNotificationHistory`** class, like in the following sample:

```
private void OnRemoveAllNotificationsClicked(object sender, RoutedEventArgs
e)
{
    ToastNotificationHistory history = ToastNotificationManager.History;
    history.Clear();
}
```

Tile notifications

We've already highlighted the importance of providing a great tile experience with our application; a good tile is able to keep the user continuously updated on the application's content, without requiring them to open it to perform any task, or to catch their attention and draw them into the app. The weather application is a very good sample of this scenario: users are able to immediately discover the weather forecast in their current position simply by glancing at their start screen. However, if they want to see additional information (like the long-term forecast), they can still open the application.

Tile notifications work a lot like toast notifications, with a rich catalog of available templates, represented using XML. However, an important difference with toast notifications is that we need to provide a default tile. The user, in fact, could decide to pin our application on his Start Screen at any time, even when no notifications have been sent. The default tiles are set in the manifest file, in the **Visual Assets** section. We can define many assets, since Windows and Windows Phone support multiple tile sizes. The required images are:

- **Square70x70** (in Windows) and **Square71x71** (in Windows Phone), the smallest square tile.
- **Square150x150**, the standard square tile.
- **Wide310x150**, the wide rectangular tile.
- **Square310x310**, a big square tile, available only in Windows.

Figure 19: The different tiles in Windows and Windows Phone

For any supported tile size, you'll be able to specify:

- A background image. It's important to highlight that the image should have a maximum resolution of 500x500, and should be smaller than 500 KB. Otherwise, the notification will be discarded.
- A text to show on the tile.
- The text color (dark or light). This option is available only in Windows.
- The background color to apply to the tile. In Windows Phone, you can also use the **`transparent`** value: the tile will automatically adapt to the user's theme, and it will be able to properly support the start background image.
- The default size that is used when the user pins the application on the Start screen. This option is available only in Windows. On Windows Phone, by default, tiles are created using the Square150x150 format.

Once you've defined the default tile aspect, you can start sending tile notifications to update them. The template catalog is very rich, as you can see in the MSDN documentation at http://s.qmatteoq.com/TileTemplatesCatalog. Most of the templates will be automatically adapted to the platform, so that you'll be able to send the same notification payload to Windows and Windows Phone.

There are three kind of tiles in the template catalog:

- **Peek tiles**: These tiles support an animation to display a secondary content. The kind of animation is different according to the platform. On Windows, the main content will slide on the top, and then the secondary one will be displayed; on Windows Phone, the tile will flip to display the secondary content that is placed on the back side.

- **Block tiles**: These tiles are used when the most important information you want to deliver is a number (like the Calendar app, which shows the current day).
- **ImageCollection tiles**: These are used to display a collection of images, by applying a mosaic animation similar to the one used by the People application.

Templates are also grouped according to the tile size, so you'll find specific templates for the following sizes:

- **Square71x71,** the smaller one in Windows Phone, unavailable in Windows.
- **Square150x150**, the standard square tile.
- **Wide310x150**, the wide tile.
- **Square310x310**, the big square tile, supported only by Windows.

Windows also offers a small square tile size, which is called **Square70x70**; however, you won't find any template in the catalog list for this size, since we can only update these files with a badge notification (which we'll see later).

Let's see now how to update a tile. We're going to use, as reference, the template called **TileSquare150x150PeekImageAndText01**, which is defined in the following way:

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150PeekImageAndText01">
      <image id="1" src="image1" alt="alt text"/>
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

The main element is called **`tile`**, and it contains another element called **`visual`**, which includes the visual definition of the tile. The most important element is **`binding`**, which defines, thanks to the **`template`** property, which template the XML is going to define. Inside the **`binding`** element, we're going to find an element for each information of the tile that we can customize; in the previous sample, the template supports one **`image`** and four **`text`** elements. However, as we've seen for toast notifications, we don't have to manually define the XML file. Thanks to the **`TileUpdateManager`** class, we can retrieve the template we need by using the **`GetTemplateContent()`** method and passing as parameter one of the values of the **`TileTemplateType`** enumerator. The following sample shows how to create a **TileSquare150x150PeekImageAndText01** tile, and how to customize its properties:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    XmlDocument template =
TileUpdateManager.GetTemplateContent(TileTemplateType.
    TileSquare150x150PeekImageAndText01);
```

```
    XmlNodeList texts = template.GetElementsByTagName("text");
    texts[0].InnerText = "Tile title";
    texts[1].InnerText = "Text 1";
    texts[2].InnerText = "Text 2";
    texts[3].InnerText = "Text 3";
    XmlNodeList images = template.GetElementsByTagName("image");
    ((XmlElement)images[0]).SetAttribute("src", "ms-
appx:///Assets/wplogo.png");
    TileNotification notification = new TileNotification(template);
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    updater.Update(notification);
}
```

The template is identified by a **XmlDocument** object; using the methods offered by this class (like **GetElementsByTagName()**), we are able to retrieve the various text and image elements and change their values. Once the template is ready, we can create a **TileUpdater** object, by calling the **CreateTileUpdaterForApplication()** method of the **TileUpdater** class. In the end, we can update the tile by creating a new **TileNotification** object (and passing as parameter the template) and passing it to the **Update()** method.

Update multiple sizes with one notification

The previous code has a flaw: we're updating just one of the available tile sizes, but we don't know which size the user is using. In addition, they're able to change the size anytime without prior notice. If, for example, the user resizes the tile to wide instead of using a square one, they won't see the update we've just sent. The correct approach is to include, in the bigger tile size, a notification to also update the smaller versions. For example, in this case, other than sending just a notification using the template **TileSquare150x150PeekImageAndText01**, we need to also update the tile with the **TileWide310x150PeekImageAndText01**, which is used for the wide tile.

To achieve this goal, we can add multiple **binding** nodes in the same notification, like in the following sample:

```
<tile>
  <visual version="2">
    <binding template="TileWide310x150ImageAndText01">
      <image id="1" src="ms-appx:///assets/redWide.png"/>
      <text id="1">This is a notification for a wide tile</text>
    </binding>
    <binding template="TileSquare310x150PeekImageAndText01">
      <text id="1">This is a notification for a square tile</text>
    </binding>
  </visual>
</tile>
```

Consequently, before updating the tile using the **`TileUpdater`** class, we need to define the new template and add it as a child of the visual tag, like in the following sample:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    XmlDocument template =
TileUpdateManager.GetTemplateContent(TileTemplateType.
    TileSquare150x150PeekImageAndText01);
    XmlNodeList texts = template.GetElementsByTagName("text");
    texts[0].InnerText = "Tile title";
    texts[1].InnerText = "Text 1";
    texts[2].InnerText = "Text 2";
    texts[3].InnerText = "Text 3";
    XmlNodeList images = template.GetElementsByTagName("image");
    ((XmlElement)images[0]).SetAttribute("src", "ms-
appx:///Assets/wplogo.png");

    XmlDocument wideTileXml =
TileUpdateManager.GetTemplateContent(TileTemplateType.
    TileWide310x150PeekImageAndText01);
    XmlNodeList wideTileTexts = wideTileXml.GetElementsByTagName("text");
    wideTileTexts[0].AppendChild(wideTileXml.CreateTextNode("Wide tile
tile"));
    IXmlNode node =
template.ImportNode(wideTileXml.GetElementsByTagName("binding").Item(0),
true);
    template.GetElementsByTagName("visual").Item(0).AppendChild(node);
    TileNotification notification = new TileNotification(template);
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    updater.Update(notification);
}
```

After we've defined the tile using the **TileSquare150x150PeekImageAndText01** template as we did before, we also define a new tile using the **TileWide310x150PeekImageAndText01** template. Then, before sending the update, we combine the two templates into a single XML file, and, only at the end, we create a new **`TileNotification`** object.

Sending a tile notification with the NotificationsExtensions library

The NotificationsExtensions library we've previously seen also with tile notifications. In this case, we're going to use the **`TileContentFactory`** to class to create an object based on one of the many available templates, which we will use to easily define the visual layout, thanks to its properties. The following sample shows how to create a tile using the **TileSquare150x150PeekImageAndText01** template from the library:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    ITileSquare150x150PeekImageAndText01 template =
TileContentFactory.CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Tile title";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    TileNotification notification = template.CreateNotification();
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    updater.Update(notification);
}
```

Each template object offers one or more properties, according to the information supported by the tile. In the previous sample, we set the title (using the **TextHeading** property), the various texts (using the **TextBody** properties), and the image (using the **Image** property). After we've customized the template, we can create a **TileNotification** object by using the **CreateNotification()** method. We'll get in return the object required by the **Update()** method of the **TileUpdater** class.

The NotificationsExtensions library can also be used to combine multiple tile updates in the same notification. Every template, in fact, supports a way to define the smaller version to use (for example, the big square template has a property to define the wide one, the wide template has a property to define the small square one, etc.). The following sample shows how to send a notification to update the wide and square tiles at the same time:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    var template =
TileContentFactory.CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Title";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";

    var wideTemplate =
TileContentFactory.CreateTileWide310x150PeekImageAndText01();
    wideTemplate.TextBodyWrap.Text = "Wide tile";
    wideTemplate.Image.Src = "ms-appx:///Assets/wplogo.png";
    wideTemplate.Square150x150Content = template;

    TileNotification wideNotification = wideTemplate.CreateNotification();
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
```

```
    updater.Update(wideNotification);
}
```

We define both the square and wide templates, and we set the **Square150x150Content** property of the wide template with the square one we've previously defined. Then we just need to create a **TileNotification** object starting from the bigger tile notification, which contains the definition of the smaller one (in this case, the small square one). This is the object to pass as parameter of the **Update()** method of the **TileUpdater** class.

Managing a notifications queue

As default behavior, every time we send a tile notification, we override the previous content. We can also enable a notifications queue; the system will be able to store up to five notifications, which will be cycled on the tile. To achieve this goal, we just need to call the **EnableNotificationsQueue()** method of the **TileUpdater** class, passing as parameter the **Boolean** value **true**. Let's see the following sample, which uses the NotificationsExtensions library we've previously seen:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    ITileSquare150x150PeekImageAndText01 template = TileContentFactory.
    CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Title 1";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";

    TileNotification notification = template.CreateNotification();
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    updater.EnableNotificationQueue(true);
    updater.Update(notification);

    ITileSquare150x150PeekImageAndText01 template2 = TileContentFactory.
    CreateTileSquare150x150PeekImageAndText01();
    template2.TextHeading.Text = "Title 2";
    template2.TextBody1.Text = "Text 1";
    template2.TextBody2.Text = "Text 2";
    template2.TextBody3.Text = "Text 3";
    template2.Image.Src = "ms-appx:///Assets/wplogo.png";

    TileNotification notification2 = template.CreateNotification();
    updater.Update(notification2);
}
```

We have create two notifications, but before sending the first one, we've called the **`EnableNotificationsQueue()`** method of the **`TileUpdater`** class. The result of this code is that the second update won't overwrite the first one, and the two notifications will be displayed on the main tile alternatively.

Thanks to the **`Tag`** property, we can also replace a queued notification, as we did to replace toast notifications in the Action Center in Windows Phone. It's enough, in fact, to send a new notification with the same **`Tag`** property of an existing notification:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    ITileSquare150x150PeekImageAndText01 template =
    TileContentFactory.CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Title 1";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";
    TileNotification notification = template.CreateNotification();
    notification.Tag = "MyNotification";
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    updater.EnableNotificationQueue(true);
    updater.Update(notification);
}
```

In the previous sample, we assigned the **`MyNotification`** value to the **`Tag`** property. From now on, if we send new tile notifications with a different tag, they will be simply added to the queue. However, if we send another notification with the same tag, it won't be queued, but it will replace the existing one.

Scheduling a tile notification

Tile notifications can also be scheduled, so that they are sent at a specific date and time. Scheduled tile notifications are created using the **`ScheduleTileNotification`** class, which requires the XML payload of the template, and the sending date and time, which is a **`DateTimeOffset`** object. The following sample shows how to schedule a notification after five seconds:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
{
    ITileSquare150x150PeekImageAndText01 template = TileContentFactory.
    CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Title 1";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
```

```
    template.Image.Src = "ms-appx:///Assets/wplogo.png";

    XmlDocument xml = template.GetXml();
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForApplication();
    ScheduledTileNotification scheduledTile = new
ScheduledTileNotification(xml,
    DateTimeOffset.Now.AddSeconds(5));
    updater.AddToSchedule(scheduledTile);
}
```

We're using the NotificationsExtensions library, so we call the **`GetXml()`** method on the template to get the XML definition of the notification. After we've created the **`ScheduledTileNotification`** object, we schedule the update using the **`AddToSchedule()`** method of the **`TileNotifier`** class.

Periodically updating the tile

All the update methods we've seen so far are also available for toast notifications. However, there's an update method that is available only for tiles: the periodic notification. In this scenario, it's not an external actor that sends the notification (a background task, a push notification server, etc.); it's the application itself that periodically checks an Internet resource to detect tile updates. This way, if we want to update the tile's content, we'll only have to update the XML file that we have published on the Internet.

The first step is to create an XML file with the template definition. For example, if we want to update the tile using the **TileSquare150x150PeekImageAndText01** template, we'll need to create a text file with the following definition:

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150PeekImageAndText01"
fallback="TileSquarePeekImageAndText01">
      <image id="1" src="http://www.qmatteoq.com/logo.png" />
      <text id="1">Title</text>
      <text id="2">Text 1</text>
      <text id="3">Text 2</text>
      <text id="4">Text 3</text>
    </binding>
  </visual>
</tile>
```

The second step is to publish the XML file on the Internet, so that the URL can be reached by the application. It could be a web server, a cloud service, etc.

Now you have two ways to perform periodic updates. The first one is to configure them in the manifest file. This solution is useful when you want to start receiving tile updates immediately after the application has been installed. To set up the periodic updates, you'll need to open the **Application** section of the manifest file. In the **Tile update** section, you can define the update recurrence (from every half hour to daily) and the URL of the XML file with the tile's definition.

The second way to configure periodic updates is by using code. The **`TileUpdater`** class offers a method called **`StartPeriodicUpdate(),`** which requires the URL of the XML file and one of the values of the **`PeriodicUpdateRecurrence`** enumerator to define the recurrence.

```
private void OnSetTileUpdateClicked(object sender, RoutedEventArgs e)
{
    TileUpdater manager =
TileUpdateManager.CreateTileUpdaterForApplication();
    Uri url = new Uri("http://www.qmatteoq.com/tile.xml", UriKind.Absolute);
    manager.StartPeriodicUpdate(url, PeriodicUpdateRecurrence.Hour);
}
```

One of the advantages of managing periodic notifications in code is that we can implement scenarios that are not supported by the manifest file. For example, we can set multiple URLs for the update; this way, at every recurrence, instead of using the same URL, the application will cycle between URLs and pick the next one in the list. To achieve this goal, we need to call the **`StartPeriodicUpdateBatch()`** method, passing as parameter a collection of **`Uri`** objects, like in the following sample:

```
private void OnSetTileUpdateClicked(object sender, RoutedEventArgs e)
{
    TileUpdater manager =
TileUpdateManager.CreateTileUpdaterForApplication();
    List<Uri> uris = new List<Uri>
    {
        new Uri("http://www.qmatteoq.com/tile.xml", UriKind.Absolute),
        new Uri("http://www.qmatteoq.com/tile2.xml", UriKind.Absolute),
        new Uri("http://www.qmatteoq.com/tile3.xml", UriKind.Absolute)
    };
    manager.StartPeriodicUpdateBatch(uris, PeriodicUpdateRecurrence.Hour);
}
```

Another operation that we can perform with the **`TileUpdater`** class is to stop receiving periodic updates, by calling the **`StopPeriodicUpdate()`** method:

```
private void OnStopTileUpdatesClicked(object sender, RoutedEventArgs e)
{
    TileUpdater manager =
TileUpdateManager.CreateTileUpdaterForApplication();
    manager.StopPeriodicUpdate();
```

```
}
```

Secondary tiles

Windows and Windows Phone offer a way to create multiple tiles on the Start screen, which belong to the same application. They are used mostly for two scenarios:

- To provide quick access to inner sections of the application. For example, a newsreader application could provide a way for the user to pin a tile for each news category on his Start screen. When the user taps on this tile, the application will be opened directly on the list of news items that belong to that category.
- To provide independent tile updates. By expanding the previous sample, we'll be able to update a secondary tile connected to a specific news category only with the title of these news items.

In addition, secondary tiles are synchronized using the Microsoft Account of the user, like the roaming folder we saw in Chapter 5. This way, if the user has installed the application on multiple devices, she will automatically find all the secondary tiles she created on her Start screen. To create multiple tiles, we're going to use the same APIs on both platforms. However, there's an important user experience difference between Windows and Windows Phone:

- Windows requires a confirmation from the user before creating the title: a pop-up window will show a tile preview, and the user will have to press the OK button to add it to the Start screen.
- Windows Phone doesn't require any confirmation, but the application is suspended when the tile is created, with the Start screen automatically focused on the new tile.

Both scenarios have been implemented to avoid having an application create an unlimited number of tiles on the user's Start screen without approval.

Secondary tiles are mapped with the **`SecondaryTile`** class, like in the following sample:

```
private async void OnCreateSecondaryTileClicked(object sender,
RoutedEventArgs e)
{
    string tileId = "SecondaryTile";
    string shortName = "Tile";
    string displayName = "This is a secondary tile";
    string arguments = "science";
    Uri logo = new Uri("ms-appx:///Assets/wplogo.png", UriKind.Absolute);
    SecondaryTile tile = new SecondaryTile(tileId, shortName, displayName,
arguments, TileOptions.ShowNameOnLogo, logo);
    bool isPinned = await tile.RequestCreateAsync();
}
```

When we create a new **`SecondaryTile`** object, we need to specify:

- A unique identifier of the tile.

- The tile name (which should be provided both in short and in full length).
- The activation parameters, which are passed to the launching event to identify which tile has been used, so that we can redirect the user to the proper page and display the required information.
- The image to use for the tile.

The tile is created using the **RequestCreateAsync()** method, which will return a **Boolean** value to notify you of the status of the operation. However, this value is useful only on Windows, since the user has the chance to abort the operation by dismissing the confirmation pop-up window.

The **SecondaryTile** class also offers a way to deeply customize the tile's content, thanks to the **VisualElements** property, which offers a way to interact with the same properties we've seen in the manifest file, like the supported tile sizes, the background color, the text, etc.

```
private async void OnCreateSecondaryTileClicked(object sender,
RoutedEventArgs e)
{
    string tileId = "SecondaryTile";
    string shortName = "Tile";
    string displayName = "This is a secondary tile";
    string arguments = "15";
    Uri logo = new Uri("ms-appx:///Assets/wplogo.png", UriKind.Absolute);
    SecondaryTile tile = new SecondaryTile(tileId, shortName, displayName,
arguments,
    TileOptions.ShowNameOnLogo, logo);
    tile.VisualElements.BackgroundColor = Colors.Red;
    tile.VisualElements.ForegroundText = ForegroundText.Light;
    tile.VisualElements.Wide310x150Logo = new Uri("ms-appx:///Assets/wide-
wplogo.png",UriKind.Absolute);
    bool isPinned = await tile.RequestCreateAsync();
}
```

Updating a secondary tile using a notification

Secondary tiles can also be updated using notifications. The required code is the same we've seen for the main tile. The only difference is that, this time, to create the **TileUpdater** object, we're going to use the **CreateTileUpdaterForSecondaryTile()** method. It requires as parameter the unique identifier of the tile we want to update (which is the first parameter we passed when we created the **SecondaryTile** object).

After that, we can proceed as usual to create the notification starting from one of the templates, and then immediately update the tile (using the **Update()** method), schedule the notification (using the **AddToSchedule()** method), or set the periodic update (using the **StartPeriodicUpdate()** method). The following sample shows how to update a secondary tile:

```
private void OnUpdateTileClicked(object sender, RoutedEventArgs e)
```

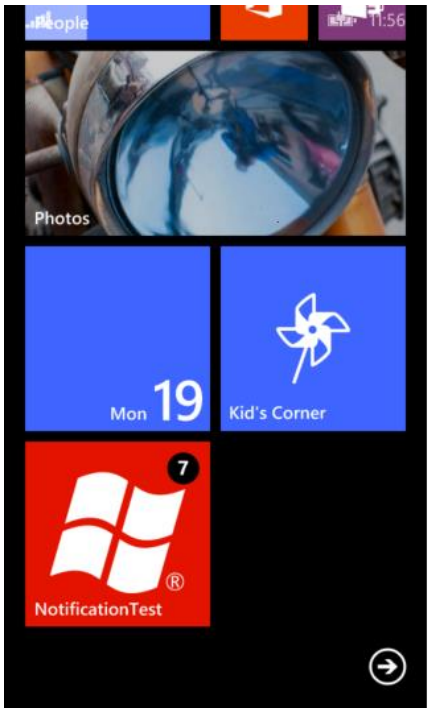
```
{
    var template =
    TileContentFactory.CreateTileSquare150x150PeekImageAndText01();
    template.TextHeading.Text = "Title";
    template.TextBody1.Text = "Text 1";
    template.TextBody2.Text = "Text 2";
    template.TextBody3.Text = "Text 3";
    template.Image.Src = "ms-appx:///Assets/wplogo.png";

    TileNotification notification = template.CreateNotification();
    string tileId = "SecondaryTile";
    TileUpdater updater =
TileUpdateManager.CreateTileUpdaterForSecondaryTile(tileId);
    updater.Update(notification);
}
```

Badge notifications

Badge notifications are connected to tiles; they are symbols that can be displayed over the tile to draw the user's attention. There are two kind of badges:

- **Numbers**
- **Symbols**, to warn the user that something happened in the application that requires his attention.



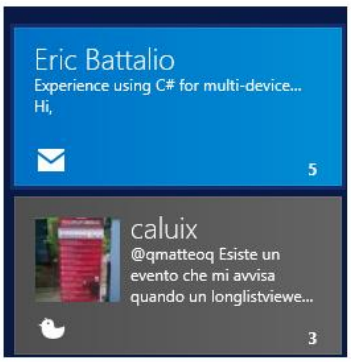


Figure 20: Numeric badge notifications in Windows and Windows Phone

All the available symbols are listed in the MSDN documentation (http://s.qmatteoq.com/MSBadgeCatalog), but it's important to highlight that Windows Phone supports only the **alert** and **attention** symbols.

Badge notifications act like toast or tile notifications; they're defined by an XML file that, in this case, is much simpler, since it contains only the badge value. The following sample shows how to define a numeric badge:

```
<badge value="1"/>
```

Here is, instead, a symbol badge:

```
<badge value="attention"/>
```

Compared to toast or tile notifications, it's also easier to manipulate the XML document to create a template, as you can see in the following sample:

```
private void OnUpdateBadgeClicked(object sender, RoutedEventArgs e)
{
    XmlDocument template =
BadgeUpdateManager.GetTemplateContent(BadgeTemplateType.BadgeNumber);
    XmlElement badgeElement =
(XmlElement)template.SelectSingleNode("/badge");
    badgeElement.SetAttribute("value", "7");
    BadgeNotification notification = new BadgeNotification(template);
    BadgeUpdater badgeUpdater =
BadgeUpdateManager.CreateBadgeUpdaterForApplication();
    badgeUpdater.Update(notification);
}
```

The approach is always the same: we use a class called **BadgeUpdateManager** to retrieve the template, by using the **GetTemplateContent()** method, which requires as the parameter one of the two values of the **BadgeTemplateType** enumerator (**BadgeNumber** or **BadgeGlyph**). Then we retrieve the node called badge (using the **SelectSingleNode()** method), and we change the value attribute (using the **SetAttribute()** method). In the end, we use the XML template to create a **BadgeNotification** object, which is sent using the **Update()** method of the **BadgeUpdater** class.

Sending a badge notification using the NotificationsExtensions library

The NotificationsExtensions library supports badge notification by providing two different classes, according to the content we want to deliver: **BadgeNumericNotificationContent** for numbers, and **BadgeGlyphNotificationContent** for symbols.

It's enough to create one of these two objects (passing as parameter the number or the symbol to display) and then call the **`CreateNotification()`** method to turn the template into a **`BadgeNotification`** object, which can be sent using the **`BadgeUpdater`** class.

```
private void OnUpdateBadgeClicked(object sender, RoutedEventArgs e)
{
    BadgeNumericNotificationContent template = new
BadgeNumericNotificationContent(7);
    BadgeNotification notification = template.CreateNotification();
    BadgeUpdater updater =
BadgeUpdateManager.CreateBadgeUpdaterForApplication();
    updater.Update(notification);
}
```

If we need to send a symbol badge, the library offers a useful enumerator called **`GlyphValue`**, with a list of all supported symbols. The following sample shows how to send a badge notification with the attention symbol:

```
private void OnUpdateBadgeClicked(object sender, RoutedEventArgs e)
{
    BadgeGlyphNotificationContent template = new
BadgeGlyphNotificationContent(GlyphValue.Attention);
    BadgeNotification notification = template.CreateNotification();
    BadgeUpdater updater =
BadgeUpdateManager.CreateBadgeUpdaterForApplication();
    updater.Update(notification);
}
```

Display notifications on the lock screen

In Windows and Windows Phone, the developer can interact with the lock screen, which is the screen that is displayed when the device is not being used. Displaying notifications on the lock screen is very easy, since it doesn't require a set of special APIs; these notifications rely on the existing tile and badge notifications we've just seen. The only step is to properly set up the manifest file. Every time we're going to send a notification to update our tile, if the application has been configured to interact with the lock screen, the notification's content will also be displayed on the lock screen.

There are two kinds of notifications that are supported:

- **Numeric badge notifications**: We can display the same counter on the tile and on the lock screen. You can have up to five applications on the Windows Phone lock screen, and up to seven on the Windows lock screen.
- **Text notifications**: The text that is displayed as title of the wide template can also be displayed on the lock screen. You can have only one application of this type.

Figure 21: Lock screen notifications in Windows Phone

To support lock screen interaction, the first step is to edit the **Application** section of the manifest file. In the **Notifications** section, you'll find a dropdown menu called **Lock screen notifications**. You'll be able to choose if you support **Badge** notifications (the numeric value) or **Badge and Tile text** notifications (both of them). As soon as you enable one of the two options, you'll see an error icon displayed in the **Visual Assets** section of the manifest It's required, in fact, to also set the logo to use in the lock screen, near the number, in the **Badge logo** section. You'll have to use a white image that is 24x24 pixels, with a transparent background. You can't use colors in a lock screen icon.

On Windows, you'll also get an error in the Declarations section of the manifest: Windows applications, in fact, are required to be connected to a background task, which type should be Timer, Location, Control channel, or Push Notification.

It's important to highlight that the user has full control over which lock screen notifications to display. We will always able to send them, but it's the user that needs to define how many, and which applications to display on the lock screen.

Push notifications

All the samples we've seen in the previous section are useful when the notifications are sent directly by the application or a background task. However, in some cases we can't rely on these scenarios. For example, we need to support real-time notifications, but background tasks have some constraints that make them not suitable for this scenario.

Push notifications are the answer to this problem, since they rely on an external service that sends the notifications, like a website, a desktop application, or a cloud service. When a specific event occurs, the service will send a notification to the device, which will receive and process it. To improve reliability and security, a service can't directly send a notification to the device, but it needs to use a service offered by Microsoft, called Windows Notification Service (WNS). WNS is able to gather all the notifications and to redirect them to the proper device.

What is the push notification workflow? You can see a summary in the following image:

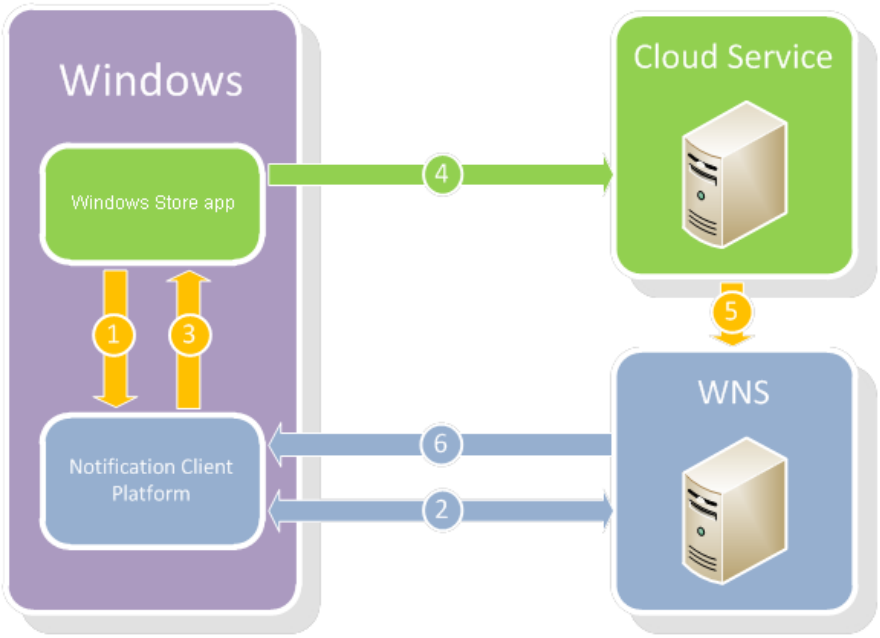


Figure 22: The push notification workflow

When a Windows Store application registers with the WNS, it receives a URL that univocally identifies the push notification channel. It's a standard HTTP URL, which is exposed on Internet (here a real channel sample: https://db3.notify.windows.com/?token=AgYAAABbCcIgWtH2CggvXoBateheW%2b0tKoEYCv1XIl16GrBjNFUta4S4YT1JhEOCh6JKij4CuSOXOcg%2b4auoEw8chyp%2fHzogm7ieUMzWc9WCZEK2age%2fR1Zl3CTh0rKFsZE1pEE%3d).

Now it will be easier to understand why all the notifications we've seen so far are identified by an XML file. Sending a push notification simply means preparing a HTTP request with, as content, the XML definition of the notification and sending it, using a POST operation, to the channel's URL.

The typical push notification workflow is:

- When the application is opened, it registers for a notification channel.
- The application sends to our service the URL. Typically, it's saved inside a database, to be used every time the service has to send a notification to the registered devices.
- The service sends a toast, tile, badge, or raw notification with a POST command on the channel. In return, it receives a response that the service can use to understand the request status.

One of the most important advantages of this architecture is that it's based on standard technologies, like HTTP or XML. Consequently, we are not forced to use a Microsoft technology to create our backend, but we can easily integrate a push notification service for Windows Store applications in our already existing backend service, no matter if it's written in Java, PHP, or Node.js.

Sending a push notification

In this section, we're going to see how to send a push notification from a server application. As already mentioned, you'll be able to use any technology. However, since you're reading a book about developing applications for Windows and Windows Phone, it's likely that you already know Microsoft technologies. Consequently, as a server sample, we're going to build a WPF application. To perform the network operations required to interact with the WNS, we're going to use the **`HttpClient`** class we've already seen in Chapter 6. Since this class isn't natively available in WPF, you will have to install a special NuGet package (http://www.nuget.org/packages/Microsoft.Net.Http/), which adds **`HttpClient`** support to any technology based on the .NET framework.

The first step to send a notification is to manage authentication. The server application needs to be authenticated with the WNS, so that only an authorized application can send push notifications to your application. To manage the authentication, you'll have to register your application on the Windows Dev Center, even if it's a Windows Phone app; right now, the Windows Phone Dev Center doesn't support the push notifications setup. Consequently, to support push notifications in your applications, you'll need a valid developer account. Without it, you'll be able to send notifications only by using the specific tool that is provided with the Windows Phone emulator.

To proceed with the push notifications configuration, you'll have to start a new application submission on the Windows Dev Center. We're going to see the procedure in detail in Chapter 12, but for now, let's take a look at the step required to properly set up the authentication.

The first step is to reserve a name for the application, which is the one that will displayed to the user on the Store. You can do it by right-clicking on your Windows project in Visual Studio and choosing, in the **Store** section, the **Associate App with the Store** option. You'll start a wizard, which will guide you through the procedure: the first step is to log in with the Microsoft Account connected to your developer account. Then, a window will display a list of all your applications that are registered on the dashboard. If it's a new application, you can reserve a new name by filling the **Reserve a new app name** field and pressing the **Reserve** button. Otherwise, simply choose one name from the list and press the **Next** button.

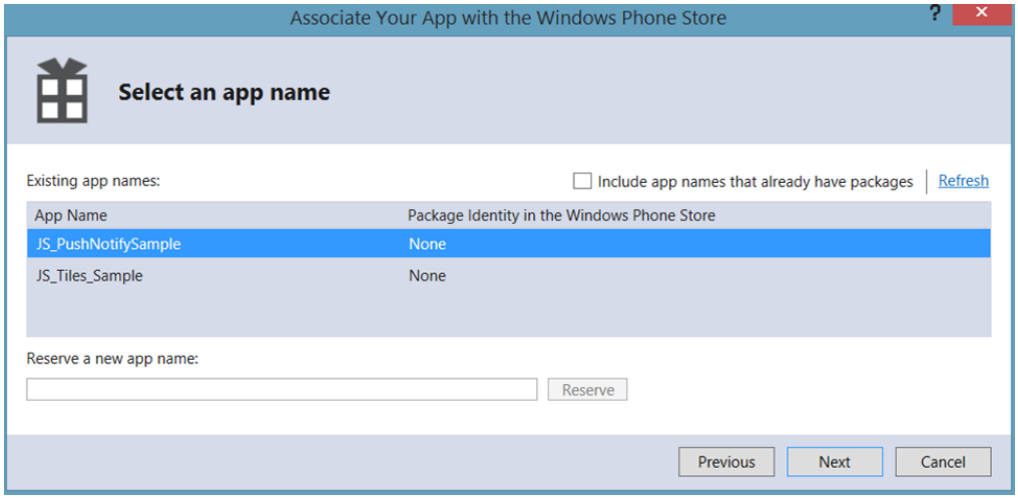


Figure 23: The Visual Studio's screen to reserve a name for the application

Now you're ready to start the submission. Go to the website http://dev.windows.com, log in with your Microsoft Account, and click on the Dashboard link. You'll be asked which Dev Center you want to use: choose the Windows Store. Once you've landed on the Windows Dev Center, start the submission procedure by clicking on the **Submit new app link**. The procedure will start by displaying all the required steps to complete the submission. We'll ignore them for the moment, and just complete the minimum set of steps required to get the authentication credentials for the push channel.

The first step is called **App name**, and it's used to define the application name. Since we already did this operation in Visual Studio, we can just choose it from the dropdown menu. After completing this step, we'll return to the list; let's jump directly to Step 3, called **Services**. It's the one we need to set up the push notifications channel. Since we're using our own backend, we need to click **If you have an existing WNS solution or need to update your current client secret, visit the Live Service site**. Finally, you'll land on a page that will display the information you're looking for, which is the **Package SID** and the **Client Secret**, as displayed in the following image.

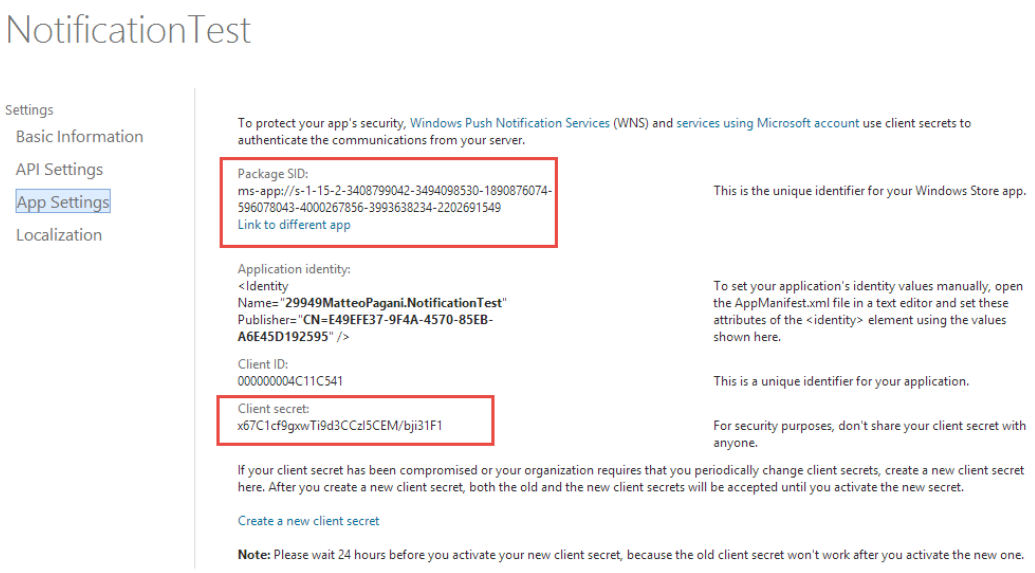


Figure 24: The required information to set up authentication for push notifications

Now that we have this information, we can start to write some code. To perform authentication, we need to perform a POST request with a package that contains the Package SID and the Client Secret we previously retrieved. The following sample shows how to perform a proper authentication:

```
private async void OnAuthenticateClicked(object sender, RoutedEventArgs e)
{
    string sid = "ms-app://s-1-15-2-3408799042-3494098530-1890876074-
596078043-4000267856-3993638234-2202691549";
    string secret = "x67C1cf9gxwTi9d3CCzI5CEM/bji31F1";
    string encodedSid = HttpUtility.UrlEncode(sid);
    string encodedSecret = HttpUtility.UrlEncode(secret);
    string body =
string.Format("grant_type=client_credentials&client_id={0}&client_secret={1}&
scope=notify.windows.com", encodedSid, encodedSecret);
    HttpClient client = new HttpClient();
    StringContent content = new StringContent(body);
    content.Headers.ContentType = new MediaTypeHeaderValue("application/x-
www-form531urlencoded");
    HttpResponseMessage response = await
client.PostAsync("https://login.live.com/accesstoken.srf", content);
}
```

The content of the HTTP request is stored inside the variable called **body**. It's a string with a set of parameters:

- **grant_type**, which is a fixed value, and has to be set to **client_credential**.
- **scope**, which is a fixed value, and has to be set to **notify.windows.com**.

- **client_id**, which is the Package Id we've previously retrieved.
- **client_secret**, which is the Client Secret we've previously retrieved.

Since the package contains only text, we can define the content using a **StringContent** object. The only special configuration to highlight is that we need to set the **ContentType** of the request to **application/x-www-form531urlencoded**.

Now we can send the request using the **PostAsync()** method of the **HttpClient** class to the URL https://login.live.com/accesstoken.srf, passing as parameter the **StringContent** object we've previously defined. If we did everything correctly, we'll get in return a JSON response with the access token that will be needed to perform all the other operations with the WNS. This is a sample JSON response:

```
{
  "token_type":"bearer",

"access_token":"EgAaAQMAAAAEgAAAC4AAzqwf/hYpeD3oG5Lj9SGc53j5U1OtVV30jwHasHVBr
xTWcUF1

zy/1pNnyn2OUUh+bCjs690ELAHQAgH7O94l6EKWoX0oK0oAIf1E0dUk3pf8kRw6VuddENz8n00JZQ
tpNlS3H

SdbvEAugraYKiYfp+Uvi677rSS7/HPhu4tvXN3KJAFoAiQAAAAAAQcURTJ18jFOdfIxT60gEAAsAO
TMuMzYu

MC40MQAAAAAAXgBtcy1hcHA6Ly9zLTEtMTUtMi0zNDA4Nzk5MDQyLTM0OTQwOTg1MzAtMTg5MDg3N
jA3NC01
  OTYwNzgwNDMtNDAwMDI2Nzg1Ni0zOTkzNjM4MjM0LTIyMDI2OTE1NDkA",
  "expires_in":86400
}
```

The following compete sample shows how to send the authentication request, and how, by using the Json.NET library we talked about in Chapter 6, we are able to retrieve the access token:

```
private async void OnAuthenticateClicked(object sender, RoutedEventArgs e)
{
    string sid = "ms-app://s-1-15-2-3408799042-3494098530-1890876074-
596078043-4000267856-3993638234-2202691549";
    string secret = "x67C1cf9gxwTi9d3CCzI5CEM/bji31F1";
    string encodedSid = HttpUtility.UrlEncode(sid);
    string encodedSecret = HttpUtility.UrlEncode(secret);
    string body =
string.Format("grant_type=client_credentials&client_id={0}&client_secret={1}&
scope=notify.windows.com", encodedSid, encodedSecret);
    HttpClient client = new HttpClient();
    StringContent content = new StringContent(body);
    content.Headers.ContentType = new MediaTypeHeaderValue("application/x-
```

```
wwwform-urlencoded");
    HttpResponseMessage response = await
client.PostAsync("https://login.live.com/accesstoken.srf", content);
    if (response.IsSuccessStatusCode)
    {
        string json = await response.Content.ReadAsStringAsync();
        JObject jObject = JObject.Parse(json);
        string accessToken =
jObject.GetValue("access_token").Value<string>();
    }
}
```

If the request is successful (we check the **IsSuccessStatusCode** property of the response), we read the response's content. Since it's a string, we use the **ReadAsStringAsync()** method offered by the **Content** property. Then, using the **JObject** class of the JSON.NET library, we are able to retrieve the value of the **access_token** element. It's important to save this information locally, since we're going to include it in every request we will use to send a push notification. The access token is valid for 24 hours; after this period, we will have to repeat the authentication procedure to acquire a new one.

Now we have all the information required to send a push notification. Let's see how to compose the HTTP request that defines a notification:

- The request content is the XML file that defines the notification, according to what we've seen in this chapter.
- The request should contain a header called **X-WNS-Type**, with the type of notification that we want to send. The supported values are:
    - **wns/toast** for toast notifications
    - **wns/tile** for tile notifications
    - **wns/badge** for badge notifications
    - **wns/raw** for raw notifications
- The content type of the request is different according to the notification's type: for toast, tile, and badge notifications, it's **text/xml**; for raw notifications, it's **application/octect-stream**.
- We need to specify the size of the content, by setting the **ContentLength** header.
- We need to include a header called **Authorization**, with the access token we've previously retrieved.

The following sample shows how to send a toast notification by using the **ToastImageAndText01** template:

```
private async void OnSendNotificationClicked(object sender, RoutedEventArgs
e)
{
    string xml =
        String.Format(
            "<?xml version='1.0' encoding='utf-8'?><toast><visual><binding
```

```
template=\"ToastImageAndText01\"><image id=\"1\"src=\"{0}\" alt=\"Placeholder
image\"/><text id=\"1\">{1}</text></binding></visual></toast>",
            "World", "Title");
    StringContent content = new StringContent(xml);
    content.Headers.Add("X-WNS-Type", "wns/toast");
    content.Headers.ContentType = new MediaTypeHeaderValue("text/xml");
    content.Headers.ContentLength = xml.Length;
    HttpRequestMessage message = new HttpRequestMessage();
    message.Method = new HttpMethod("POST");
    message.Content = content;
    message.Headers.Authorization = new AuthenticationHeaderValue("Bearer",
accessToken);
    message.RequestUri = new Uri(ChannelUri.Text, UriKind.Absolute);

    HttpClient client = new HttpClient();
    HttpResponseMessage response = await client.SendAsync(message);
}
```

Since the notification's content is a string, we define a **`StringContent`** object with the XML template as body. Then we add the required header: the notification type (**`wns/toast`**, in this case), the content type (**`text/xml`**), and the content length (which is the length of the XML file). We also set the **`Authorization`** header with a new **`AuthenticationHeaderValue`** object, which requires as parameter the authentication's type (which is Bearer) and the previously retrieved access token. The last step is to define the channel URL in the **`RequestUri`** property. In this sample, we're assuming that our WPF application has a **`TextBox`** control where the user can manually insert the channel URL.

In the end, we send the notification by calling the **`SendAsync()`** method and passing as parameter the **`HttpRequestMessage`** object we've just defined. In return, we will get an **`HttpResponseMessage`** object, which can be used to determine the operation status. The status can be identified by the **`StatusCode`** property, which contains the HTTP status code of the response. The most common values we can get in return are:

- **`200`** if the request has been sent successfully.
- **`401`** if the request has not been authorized, so we need to request a new access token.
- **`410`** or **`404`** if the channel where we sent the notification isn't valid anymore, so we can safely remove this URL from our database.

You can find a complete list of all the error codes in the MSDN documentation at http://s.gmatteog.com/PushStatus.

Receiving a notification: the application

It's now time to see how to manage notifications from the other side of the channel: the Windows Store application. The first step is to contact the WNS to require a new channel. We can do it by using the **`PushNotificationManager`** class, which is included in the **`Windows.Networking.PushNotifications`** namespace.

```
private async void OnRequestChannelClicked(object sender, RoutedEventArgs e)
{
    PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsy
nc();
    Debug.WriteLine(channel.Uri);
}
```

The operation is very simple: just call the **CreatePushNotificationChannelForApplicationAsync()** method offered by the **PushNotificationChannelManager** class. You'll get, in return, a **PushNotificationChannel** object, with all the information about the channel. The most important information is the channel's URL, which is stored in the **Uri** property. In the previous sample, we just display it in the Visual Studio's Output Window. In a real application, we would send this URL to our backend service, so that it can be used to send push notifications.

However, creating a push notifications channel can raise some errors; the most common scenario is when we don't have an active Internet connection. To catch errors, wrap the **CreatePushNotificationChannelForApplicationAsync()** method into a **try** / **catch** statement, like in the following sample:

```
private async void OnRequestChannelClicked(object sender, RoutedEventArgs e)
{
    try
    {
        PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsy
nc();
        Debug.WriteLine(channel.Uri);
    }
    catch (Exception exc)
    {
        Debug.WriteLine("Error creating the channel");
    }
}
```

By using the **CreatePushNotificationChannelForApplicationAsync()** method, we create the default push notification channel for the application. If we send a tile notification to this channel, we'll update the main tile. If we want to update secondary tiles with a push notification, we can create additional channels by using the **CreatePushNotificationChannelForSecondaryTileAsync()**, which requires as parameter the unique identifier of the secondary tile.

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

```
    {
        string id = "MySecondaryTile";
        PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForSecondaryTileA
sync(id);
    }
    catch (Exception exc)
    {
        Debug.WriteLine("Error creating the channel");
    }
}
```

Intercepting a notification when the application is running

The primary use case of push notifications is when the application is not running. However, since we can't determine when a push notification is going to be received, they can be sent also when the application is running.

We are able to intercept this scenario so that we can perform a custom operation (for example, immediately refreshing the data currently displayed). We can implement this feature by subscribing to the **PushNotificationReceived** event offered by the **PushNotificationChannel** class, like in the following sample:

```
private async void OnRequestChannelClicked(object sender, RoutedEventArgs e)
{
    try
    {
        PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsy
nc();
        channel.PushNotificationReceived += channel_PushNotificationReceived;
    }
    catch (Exception)
    {
        Debug.WriteLine("Error creating the channel");
    }
}


private void channel_PushNotificationReceived(PushNotificationChannel sender,
PushNotificationReceivedEventArgs args)
{
    string content;
    switch (args.NotificationType)
    {
        case PushNotificationType.Toast:
            content = args.ToastNotification.Content.GetXml();
            break;
```

```
        case PushNotificationType.Tile:
            content = args.TileNotification.Content.GetXml();
            break;
        case PushNotificationType.Badge:
            content = args.TileNotification.Content.GetXml();
            break;
        case PushNotificationType.Raw:
            content = args.RawNotification.Content;
            break;
    }
    args.Cancel = true;
}
```

In the event handler, we get a parameter with a property called **`NotificationType`**, which contains one of the values of the **`PushNotificationType`** enumerator. We can use it to discover which kind of notification has been received. The parameter also contains a property for each notification's type; once we've identified which notification we have received, we can get its content by using the **`Content`** property. When it comes to toast, tile, or badge notifications, we can simply retrieve the XML template by using the **`GetXml()`** method. Since raw notifications don't have a fixed structure, we simply retrieve the value of the **`Content`** property.

Intercepting a push notification with the **`PushNotificationReceived`** method doesn't prevent the system from managing it. It means that, even if the application is opened, a toast banner will be displayed, or the tile will be updated. If we want to avoid this behavior (for example, if the user is already using the app, it doesn't make sense to show a toast notification), we can simply set the **`Cancel`** property of the handler's parameter to **`true`**.

Testing push notifications on Windows Phone

The Windows Phone emulator offers a tool for testing push notifications. This way, you won't have to use a real backend service; the tool itself will send the notifications to the running application. The tool is called **Notifications**, and you can find it by opening the **Additional tools** panel of the emulator.

The first is to enable it by checking the **Enabled** option in the **Simulation** section. Now you can deploy your application and run the code that will create a new push notification channel. It's important to do it after enabling the tool, or it won't work. If you did everything correctly, you'll find your application in the **AppId** dropdown menu. When you choose it, all the fields with the channel information will be automatically filled.

Sending a notification is really easy: just select from the **Notification type** menu the kind of notification you want to send, and select the template to use from the **Templates** dropdown menu. In the **Notification payload** area, you'll automatically find the XML definition of the template. Just change the different properties with the content you want to simulate, and press the **Send** button to send it. You can also save custom templates for later usages, by using the **Save** and **Load** buttons.

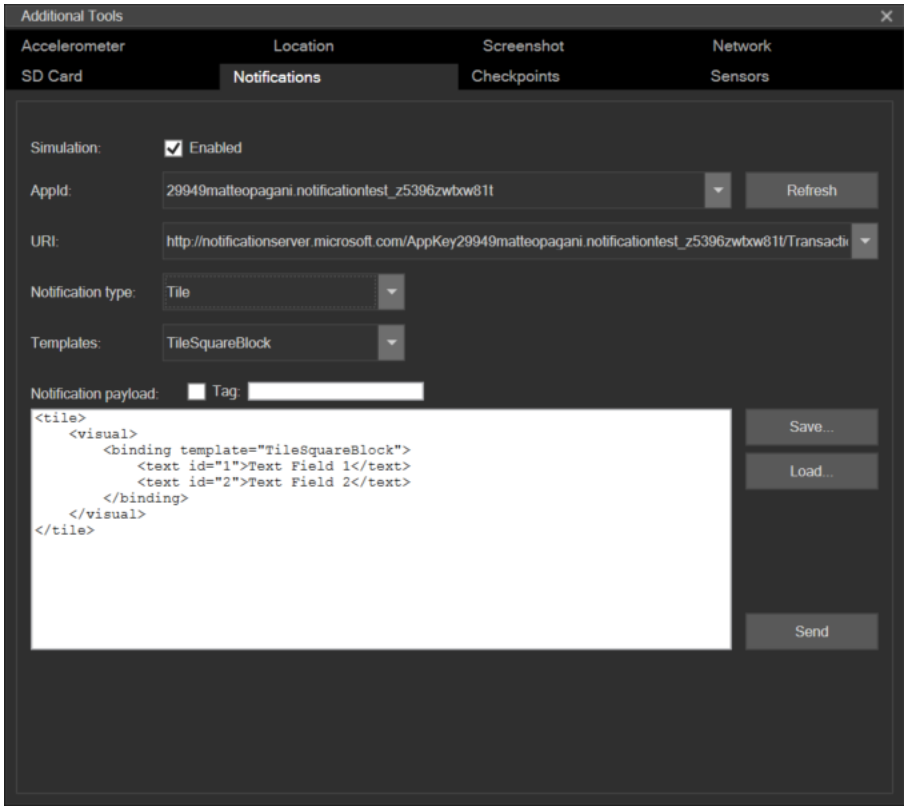


Figure 25: The Notifications tool

Chapter 7 Supporting Background Operations

In Chapter 4, we learned about the lifecycle of a Windows Store app. One of the most important concepts is suspension: when the application is moved into the background, it's no longer able to perform any operations. This approach optimizes performance and battery usage, but it can be a serious limitation to the developer; the application always needs to be open to perform any operation.

To overcome this limitation, the Windows Runtime has introduced background tasks. They are separate projects that are part of the solution, and contain some code that is executed when the application is not running, according to different conditions. Background tasks are a good balance between flexibility and user experience—they can perform operations in the background, but without affecting the performance and the battery life of the device.

Background tasks work in the same application's context; they use the same capabilities of the main application, and they share the same storage. Background tasks are based on two important concepts:

- **Triggers**: A specific event can trigger a background task. The Windows Runtime offers many trigger types, like system events, push notifications, timers, etc.
- **Conditions**: In this case, a background task can be executed only when a set of specific conditions is satisfied. For example, if the task requires an Internet connection to work properly, we can add a specific condition to make sure that it's executed only when the Internet is available.

Consequently, an application is able to register multiple background tasks, which are connected to different triggers.

Triggers

As already mentioned, triggers are the events that can invoke a background task. Here is the list of supported triggers:

- **`SystemTrigger`**: These triggers are connected to system events, like a change in the network connectivity, the lock screen activation, etc.
- **`TimeTrigger`**: These triggers are periodically executed. The minimum time interval between one execution and the next is 15 minutes for Windows, and 30 minutes for Windows Phone.
- **`LocationTrigger`**: These triggers are connected to the geofence APIs we saw in Chapter 7. They are triggered every time the user enters or exits from an area.
- **`MaintenanceTrigger`**: These are similar to the **`TimeTrigger`** category, but they used for time-consuming operations. Consequently, they are executed only when the device is connected to a power source.

- **PushNotificationTrigger**: In Chapter 10, we learned how to use push notifications. One of the supported notification types is raw, and its content can be freely defined by the developer. This trigger is invoked every time a raw notification is received by the phone and needs to be processed.
- **ControlChannelTrigger**: With this trigger, you can open a TCP/IP channel, which is also maintained when the application is closed, so that the application can continue to send and receive messages. This trigger is available only in Windows.

Additionally, there is a set of triggers that are used to interact with external devices, like **RfcommConnectionTrigger** (when a connection is established using the RFCOMM protocol), **DeviceChangeTrigger** (when the connection with a device is established or closed), **BluetoothSignalStrengthTrigger** (when the strength of the Bluetooth signal changes) or **GattCharacteristicNotificationTrigger** (used to interact with Bluetooth Low Energy devices).

Interacting with the lock screen

As we saw in Chapter 10, applications are able to display notifications on the lock screen. On Windows, lock screen support is strictly connected to background tasks. There are, in fact, some triggers (**ControlChannelTrigger**, **TimerTrigger**, **PushNotificationTrigger**, and other triggers that are part of the **SystemTrigger** category) that can be used only if the application is set to display notifications on the lock screen. You'll notice this requirement as soon as you declare, in the manifest file, a background task with one of these types. A series of errors will be displayed in the manifest editor, since you'll have to add the visual assets required to fully support the lock screen integration.

This limitation doesn't apply to Windows Phone; you can use any trigger, without being forced to enable lock screen integration.

Conditions

Conditions are used to trigger a background task only when one or more requirements are satisfied. They are not required like triggers (when you register a background task, you must specify the trigger's type, but conditions are optional), but they are useful to avoid wasting resources and executing a task when we know, in advance, that the operation can't be completed successfully.

Conditions are represented by the values of an enumerator called **SystemConditionType**. Here are the main values:

- **UserPresent** / **UserNotPresent**: This condition detects if the user is using the device.
- **InternetAvailable** / **InternetNotAvailable**: This condition detects if the device is connected to the Internet.
- **SessionConnected** / **SessionDisconnected**: This condition detects if there's an active session.
- **FreeNetworkAvailable**: This condition detects if the device is connected to Internet using a free network (like the office or home network), and not a paid connection (like a public hotspot).

Background task constraints

As we already mentioned, background tasks are a good balance between flexibility and attention to performance and battery life. Consequently, there's a set of constraints that we need to keep in mind when we create a background task.

CPU usage

The background can use the CPU for a limited time:

- One second for Windows applications that are not integrated with the lock screen.
- Two seconds for Windows applications that are integrated with the lock screen.
- Two seconds for Windows Phone applications.

It's important to highlight that these timings refer only to the CPU time, and not to the total task time. For example, if your background tasks download some data from Internet, the download time is not counted against this constraint. CPU is used only when the data has been download and needs to be processed.

Network usage

Another constraint is related to network usage, even if these limitations are applied only when the device isn't connected to a power source. The constraints change depending on whether or not the application supports lock screen integration:

Type	Usage per execution	Daily usage
Lock screen application	0.469 MB	45 MB
Standard application	0.625 MB	7.5 MB

It's important to highlight that these parameters are not fixed, but change according to the average speed of the current network connection. The values displayed in the previous table refer to a 1 Mbit connection; real values need to be multiplied for the average speed of your connection. For example, if the device has a 10 Mbit connection, you'll have to multiply the previous values by ten (so the maximum daily usage for a lock screen application would be 450 MB, and not 45 MB).

Memory usage (Windows Phone only)

Windows Phone has an added set of memory constraints, since typically smartphones offer less memory than tablets or computers. Consequently, background tasks need to respect a memory cap. If they use more memory than the one assigned to the task, they are immediately terminated. The following table shows the various memory caps; they change based on the device's memory and the task type.

Task type	512 MB devices	1 GB devices	2 GB or more devices
Location	16	30	40
Bluetooth	16	16	16
Other task types	16	30	40
Debugging limits	30	40	50

If you want to track the memory usage of a background task in real time, you can use a class called **`MemoryManager`**, which belongs to the **`Windows.System`** namespace. However, this class can be used only in a Windows Runtime Component specific to Windows Phone, since it's not available on Windows. The **`MemoryManager`** class offers a property called **`AppMemoryUsage`**, which contains the current amount of memory used. The following sample shows how to display this information in the Visual Studio Output Window:

```
private void OnCheckMemoryClicked(object sender, RoutedEventArgs e)
{
    Debug.WriteLine(MemoryManager.AppMemoryUsage.ToString());
}
```

Number of installed apps constraint (Windows Phone only)

On Windows Phone devices with 512 MB of RAM, you can only have a limited number of apps with a background task installed. If you exceed this number, the app will be regularly installed, but the background task will never be triggered. As developers, we can detect this scenario: the task registration operation (which we will see later in detail) will fail.

Battery Saver (Windows Phone only)

Windows Phone includes an application called Battery Saver, which is able to show the user the battery consumption of every app. When the battery goes below 20 percent, Battery Saver can automatically disable any background operation, including background tasks. The background tasks will resume normally when the phone is connected to a power source. In addition, the user can manually disable the background tasks connected to a specific application.

Adding a background task

Let's see how to implement a background task. The first step is to add a new project to our solution. We need to choose the **Windows Runtime Component** template, which will create a project with an empty class called **`Class1`**. You can delete it and create a new one, or rename it; it's important just to give a more meaningful name to the class.

The next step is to implement, in our class, the **`IBackgroundTask`** interface. It will require us to add a method called **`Run(),`** which is the method that is performed every time the background task is executed by the system. The following sample shows a basic implementation:

```
public sealed class BackgroundTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        //code to run in background
    }
}
```

The **`Run()`** method offers a parameter, which type is **`IBackgroundTaskInstance`**, that contains information about the current task. The most important method offered by this object is **`GetDeferral()`**. We've already seen the deferral approach, which is used to execute asynchronous operations inside a task, so that it isn't terminated by the system until all the operations are completed. As you can see in the following sample, the approach is the same we've seen in other scenarios (like the sharing contract):

```
public sealed class BackgroundTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();
        //code to run in background
        deferral.Complete();
    }
}
```

We get a **BackgroundTaskDeferral** object by using the **GetDeferral()** method. After all the operations have been completed, we call the **Complete()** method, so that the task can be terminated.

Registering the background task in the application

In addition to creating a background task's project, we have to register it in the main application. We do it in two steps: by editing the manifest file, and by writing some code. Let's start with the manifest file. In the **Declarations** section, you'll find an item in the **Available declarations** called **Background Tasks.** Once you've added it, you have to:

- Set, in the **Properties** section, which kind of triggers you want to support.
- Set, in the **Entry point** section, the full qualifier of the background task, which is the full namespace with the class name. For example, if we have created a project called **MyBackgroundTask** and we've defined the task in a class called **BackgroundTask**, the full qualifier would be **MyBackgroundTask.BackgroundTask**.

As we already mentioned, when you add a background task in Windows and you enable a specific trigger type, the manifest editor will display some errors, since you need to properly configure the lock screen integration.

Now that the manifest is properly configured, we can register the background task in code using the **BackgroundTaskBuilder** class. Typically, this operation is performed when the application starts.

```
private void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name
    == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new SystemTrigger(SystemTriggerType.UserAway,
false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundTaskRegistration taskRegistration = builder.Register();
    }
}
```

Since we can't register two background tasks with the same name, the first step is to check if the task is already registered in the system. We can do this by enumerating the **AllTasks** collection exposed by the **BackgroundTaskRegistration** class, and looking for a task with the name we have chosen as an identifier (in the previous sample, it's **Test task**).

If the task isn't already registered, we create a new **BackgroundTaskBuilder** object and define the main properties: **Name** (the identifier) and **TaskEntryPoint** (which is the same entry point we've specified in the manifest). Then we need to define the kind of trigger we want to associate to the task, which should be the same as we've defined in the manifest. We call the **SetTrigger()** method and by passing, as parameter, an **IBackgroundTask** object, which is the base interface that is implemented by every trigger in the Windows Runtime. In the previous sample, we register a **SystemTrigger**, specifying as type the **UserAway** value of the **SystemTriggerType** enumerator. We'll see more details about using system triggers later in the chapter.

Conditions are added using the **AddCondition()** method, which requires as parameter a **SystemCondition** object. The condition is defined by using one of the values of the **SystemConditionType** enumerator. Now that the task definition is completed, we can register it in the system by calling **Register()** method.

Registering a lock screen's background task in Windows

For background tasks that use a trigger that requires lock screen integration, we use different code. Before registering the task, in fact, we need to get the user's permission, by calling the **RequestAccessAsync()** method of the **BackgroundAccessStatus** class, like in the following sample:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = "Task";
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new TimeTrigger(30, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundAccessStatus status = await BackgroundExecutionManager.
        RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration taskRegistration = builder.Register();
        }
    }
```

```
}
```

The base code is the same, but before calling the Register() method, we check the value of the **`BackgroundStatus`** enumerator returned by the **`RequestAccessAsync()`** method. We proceed with the registration only if we get a value different than **`Denied`**, which is the value that we get if the user has denied the registration.

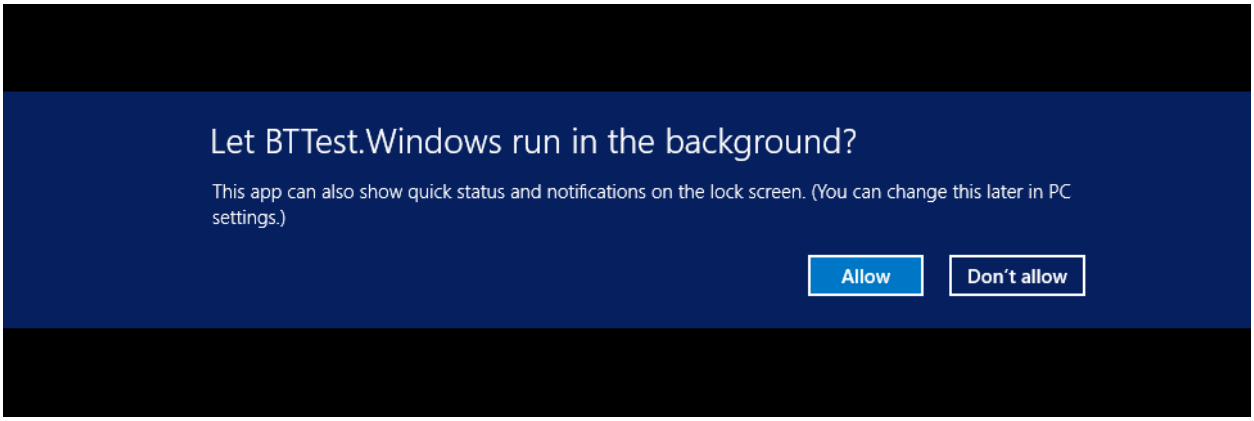


Figure 26: The pop-up message that requires permission from the user to register the background task

It's important to highlight that the previous code should be used only when the background task is using of the triggers that requires lock screen integration. If you call the **`RequestAccessAsync()`** method to register a standard background task, you'll get an exception.

Registering a background task in Windows Phone

As we already mentioned, Windows Phone doesn't make any difference based on the trigger type; you won't need to register your application with the lock screen in order to use any kind of triggers. However, in Windows Phone, it's always required to call the **`RequestAccessAsync()`** method, no matter which trigger are you using. The difference is that, on Windows Phone, no permissions will be asked to the user; the task will be silently registered. However, this method can also return a **`Denied`** status, if you're launching the application on a 512 MB device that has reached the maximum number of active background tasks, or if the user has disabled the task in the Battery Saver app.

Consequently, to properly register a background task in Windows Phone, we need to use the same code we've seen for registering the lock screen's background tasks in Windows:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
```

```
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = "Task";
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new TimeTrigger(30, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundAccessStatus status = await BackgroundExecutionManager.
        RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration taskRegistration = builder.Register();
        }
    }
}
```

Interacting with the background task in the application

Thanks to the **BackgroundTaskRegistration** class, we can find out whether the background task has been executed when the application is running, so that we can perform additional operations. This scenario is achieved by registering to the **Completed** event, which is triggered when the task has completed. The following sample shows the user a pop-up message when the task is completed:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = "Task";
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new TimeTrigger(30, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundAccessStatus status = await BackgroundExecutionManager.
            RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
            task.Completed += task_Completed;
        }
    }
}
```

```
private async void task_Completed(BackgroundTaskRegistration sender,
    BackgroundTaskCompletedEventArgs args)
{
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
    {
        MessageDialog dialog = new MessageDialog("Background task executed");
        await dialog.ShowAsync();
    });
}
```

Note that the **`Completed`** event is executed on a background thread, so if we want to interact with the user interface, we need to use the **`Dispatcher`** class.

Testing a background task

Not all tasks are easy to test. For example, a background task that uses the **`TimeTrigger`** class would require the developer to wait 15 or 30 minutes (based on the platform where they're testing the application) to see if the code is correct. Luckily, Visual Studio helps us in the testing phase, by providing a way to launch a background task connected to the application anytime we need to.

Once the application has successfully registered a background task, you'll find it in the dropdown menu labeled **Lifecycle events**, which is the one we used in Chapter 4 to test the application's lifecycle events. Remember that, by default, the toolbar that contains this dropdown menu is disabled in Visual Studio. To display it, you need to right-click in an empty space in the toolbar area, and enable the **Debug location** item. When you click on the task's name, it will be immediately executed. Then, we'll be able to test the code, like with a regular application.

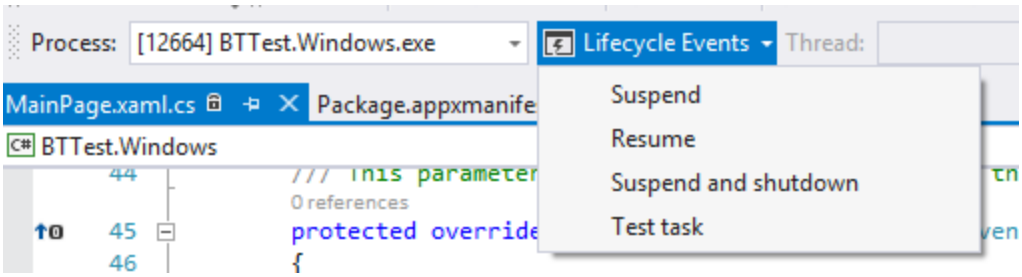


Figure 27: The Visual Studio option to manually trigger a background task

Background task samples

In the following section, we'll see how to implement some common background task types, so that it will be easier to understand how they work.

System trigger

System triggers can be used to execute a background task when something changes in the system status. Here is a list of all the available triggers:

- **`UserPresent`** / **`UserAway`**: It's triggered when the user stops using the device or resumes using it after a while. Typically, it's executed when the device is locked or unlocked. In Windows, this trigger requires us to enable the lock screen integration.
- **`NetworkStateChange`**: It's triggered when the connectivity status changes.
- **`ControlChannelReset`**: It's triggered when the TCP/IP channel is reset. In Windows, In Windows, this trigger requires us to enable the lock screen integration.
- **`InternetAvailable`**: It's triggered when the device is connected to Internet.
- **`LockScreenApplicationAdded`** / **`LockScreenApplicationRemove`**: This trigger is available only in Windows, and it's executed when the application is added or removed from the lock screen.
- **`TimeZoneChange`**: il's triggered when the time zone of the current device is changed. The trigger is invoked only when the time zone change causes the current time to shift.

The following sample shows how to register a background task that is connected to a system trigger, in this case **`InternetAvailable`**:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name== taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new
SystemTrigger(SystemTriggerType.InternetAvailable, false));
        BackgroundAccessStatus status = await
BackgroundExecutionManager.RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
        }
    }
}
```

The previous code can register, for example, a background task that (using the APIs we saw in Chapter 10 and the NotificationsExtensions library) sends a toast notification every time an Internet connection is detected:

```
public sealed class BackgroundTask : IBackgroundTask
```

```
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();
        ToastNotifier notifier =
ToastNotificationManager.CreateToastNotifier();
        var toast = ToastContentFactory.CreateToastText01();
        toast.TextBodyWrap.Text = "The device is connected to Internet";
        ToastNotification notification = toast.CreateNotification();
        notifier.Show(notification);
        deferral.Complete();
    }
}
```

Time trigger

Time trigger tasks are used to perform operations that should be performed periodically while the device is turned on. For example, a newsreader application could use a background task to periodically check for the latest news items. When you register a time trigger task using the **`TimeTrigger`** class, you'll need to provide two parameters:

- The second one is the trigger type: one shot (the task is executed just once) or traditional (the task will be periodically executed).
- The first one is a time value, and it's connected to the trigger type. If it's a one-shot trigger, it defines how much time must pass before the task is executed. If it's a standard time trigger, it defines the interval between one execution and the next. It's important to remember that this value can be more than 15 minutes for Windows, and more than 30 minutes for Windows Phone.

The following sample is used to register a background task that is executed every 60 minutes:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new TimeTrigger(60, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundAccessStatus status = await
BackgroundExecutionManager.RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
```

```
        }
    }
}
```

In the previous sample we've also added a condition, by using the **AddCondition()** method: the task will be executed only if an internet connection is available. To show a background task sample connected to a time trigger, let's reuse the newsreader app sample. The following task, by using the **SyndicationClient** class we saw in Chapter 5, retrieves the number of news items from an RSS feed and displays it on the main tile using a badge.

```
public sealed class BackgroundTask : IBackgroundTask
{
    public async void Run(IBackgroundTaskInstance taskInstance)
    {
        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();

        SyndicationClient client = new SyndicationClient();
        SyndicationFeed feed = await client.RetrieveFeedAsync(new
Uri("http://feeds.feedburner.com/qmatteoq", UriKind.Absolute));
        int numberOfArticles = feed.Items.Count;

        BadgeNumericNotificationContent content = new
BadgeNumericNotificationContent();
        content.Number = (uint) numberOfArticles;
        BadgeNotification badgeNotification = content.CreateNotification();
        BadgeUpdater updater =
BadgeUpdateManager.CreateBadgeUpdaterForApplication();
        updater.Update(badgeNotification);

        deferral.Complete();
    }
}
```

There's also another kind of time trigger, called **MaintenanceTrigger**; it's periodically executed, like the **TimeTrigger**, but only when the device is connected to a power source. Consequently, unlike the **TimeTrigger**, it doesn't require interaction with the lock screen. If the device is disconnected from the power source while the task is running, it will be interrupted, and it will restart only when it is reconnected. The following sample defines a background task using a **MaintenanceTrigger** that is executed every 60 minutes:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
```

```
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new MaintenanceTrigger(60, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.InternetAvailable));
        BackgroundAccessStatus status = await
BackgroundExecutionManager.RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
        }
    }
}
```

Geofencing triggers

In Chapter 7, we saw that the Windows Runtime provides a set of APIs to implement geofencing, which is a way to intercept when the user enters or exits from a specific geographic area. We also learned how to manage this scenario when the application is running. The same approach can also be implemented in a background task, so that we can detect when the user reaches a certain area, and trigger an operation. The following sample shows how to register a geofencing task in the main application:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new
LocationTrigger(LocationTriggerType.Geofence));
        BackgroundAccessStatus status = await
BackgroundExecutionManager.RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
        }
    }
}
```

It's enough to create a new **LocationTrigger** object, passing as parameter the value **Geofence** of the **LocationTriggerType** (which is the only available one).

Inside the background task, we can reuse the code from Chapter 7:

```
public sealed class BackgroundTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();
        var reports = GeofenceMonitor.Current.ReadReports();
        foreach (GeofenceStateChangeReport report in reports)
        {
            GeofenceState state = report.NewState;
            string status = string.Empty;
            switch (state)
            {
                case GeofenceState.Entered:
                    status = "The user entered the area";
                    break;
                case GeofenceState.Exited:
                    status = "The user exited from the area";
                    break;
            }
            IToastText01 toast = ToastContentFactory.CreateToastText01();
            toast.TextBodyWrap.Text = status;
            ToastNotification notification = toast.CreateNotification();
            ToastNotifier notifier =
ToastNotificationManager.CreateToastNotifier();
            notifier.Show(notification);
        }
        deferral.Complete();
    }
}
```

Compared to the code from Chapter 7, there are two main differences:

- In the main application, we registered to an event called **GeofenceStateChanged**, which is triggered every time a geofence status changes. The event handler contained a parameter, of type **Geomonitor**, which we used to identify which geofence has been triggered. Since, in this scenario, we're using a generic background task, we have to use another way; we need to manually access the current instance of the **Geomonitor** class, which is available with the **Current** property. This way, we can proceed like we did in the main application. We call the **ReadReports()** method to get a **GeofenceStateChangeReport** object for every geofence that has been triggered, and thanks to its properties (like **NewState**), we are able to identify the event that occurred.

- In the main application, we displayed a pop-up message to notify the user that the event was triggered. In this scenario, since we are in a background task and we can't display a pop-up message, we send a toast notification using the APIs from Chapter 10.

Push notifications triggers

In Chapter 10, we learned how push notifications work, but we only covered tile, toast, and badge notifications. I've since mentioned that there's another type of notifications, called raw notifications. Unlike the others, they don't have a fixed XML structure, but rather contain an arbitrary text.

Unlike the other types of notifications, raw notifications require a background task to be managed (unless you want to manage them only when the application is running). Every time the backend sends a raw notification to our application, the background task is executed and retrieves the notification's content, so that we can process it. The following sample shows how to register a background task that uses a **`PushNotificationTrigger`** object:

```
private async void OnRegisterTaskClicked(object sender, RoutedEventArgs e)
{
    string taskName = "Test task";
    bool isTaskRegisterd = BackgroundTaskRegistration.AllTasks.Any(x =>
x.Value.Name == taskName);
    if (!isTaskRegisterd)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = taskName;
        builder.TaskEntryPoint = "BTask.BackgroundTask";
        builder.SetTrigger(new PushNotificationTrigger());
        BackgroundAccessStatus status = await
BackgroundExecutionManager.RequestAccessAsync();
        if (status != BackgroundAccessStatus.Denied)
        {
            BackgroundTaskRegistration task = builder.Register();
        }
    }
}
```

As you can see, no parameters are required when you create a new instance of the **`PushNotificationTrigger`** class. Now we need to setup our application so that is able to receive push notifications, by creating a channel:

```
private async void OnSubscribeNotificationsClicked(object sender,
RoutedEventArgs e)
{
    PushNotificationChannel channel =
        await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsy
nc();
```

```
}
```

This code is just a basic sample; to better understand how to register and handle a notification channel, see Chapter 10.

Now that the application is configured to receive notifications, every time the backend sends a raw notification, the system will run the background task we've just registered. Let's take a look at a background task's sample code:

```
public sealed class BackgroundTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();
        RawNotification rawNotification = taskInstance.TriggerDetails as
RawNotification;
        IToastText01 toast = ToastContentFactory.CreateToastText01();
        toast.TextBodyWrap.Text = rawNotification.Content;
        ToastNotification toastNotification = toast.CreateNotification();
        ToastNotifier notifier =
ToastNotificationManager.CreateToastNotifier();
        notifier.Show(toastNotification);
        deferral.Complete();
    }
}
```

When the background task is triggered by a raw notification, its content is stored in the **`TriggerDetails`** property of the **`IBackgroundTaskInstance`** parameter. However, since **`TriggerDetails`** is a generic object (it can also be used for other scenarios), it first needs to be converted into a **`RawNotification`** object.

Now we can access the notification's content, thanks to the **`Content`** property. In the previous sample, we take this content and display it to the user with a toast notification.

Update tasks (Windows Phone only)

Windows Phone supports a special background task category, called Update Task. It's a standard background task (so it's a separate project in the same solution, with a class that implements the **`IBackgroundTask`** interface and defines the **`Run()`** method), but it's automatically triggered every time the application is updated from the Store.

It's particularly useful when your update deeply changes some technical details of the application (like the way the data is stored, for example) and you need to migrate your data before the user launches it again.

This background task has two main differences from the other tasks:

- It's not a trigger type. To register it in the manifest file, we won't add the **Background Tasks** item in the **Declarations** section, but rather the **Update Task** item. However, we'll have to set the **Entry point** field with the full qualifier of the class that implements the **`IBackgroundTask`** interface.
- It's automatically registered using the manifest declaration; it doesn't required us to register it using the **`BackgroundTaskBuilder`** class.

Except for these two differences, it behaves like a regular task; you'll have to include, in the **`Run()`** method, the code needed to update or migrate your data.