Community Experience Distilled

Windows Phone 8 Application Development Essentials

A practical guide to creating a Windows Phone 8 application using C#, XAML, and MVVM

Tomasz Szostak

[PACKT] PUBLISHING

Windows Phone 8 Application Development Essentials

A practical guide to creating a Windows Phone 8 application using C#, XAML, and MVVM

Tomasz Szostak

[PACKT]
PUBLISHING
BIRMINGHAM - MUMBAI

Windows Phone 8 Application Development Essentials

First published: October 2013

Production Reference: 1101013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-676-0

`www.packtpub.com`

Cover Image by Jarosław Blaminsky (`milak6@wp.pl`)

Credits

Author
Tomasz Szostak

Reviewers
Deep Shah
Melania Andrişan (Danciu)
Waldemar Sudoł

Acquisition Editor
Saleem Ahmed

Commissioning Editors
Maria D'souza
Llewellyn F. Rozario

Technical Editors
Vrinda Nitesh Bhosale
Amit Shetty

Copy Editors
Lavina Pereira
Mradula Hegde
Brandt D'Mello
Gladson Monteiro

Project Coordinator
Joel Goveya

Proofreader
Stephen Copestake

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Tomasz Szostak is a Senior Software Developer in an international corporation. On a daily basis, he delivers software for nuclear facilities; however, in his spare time, he becomes a mobile-application-fascinated developer. He is in love with the best practices in creating software.

He has been working on the development of Windows Phone application since the very first version of WP SDK was released. He is the author of tens of Windows Phone market applications with some successes; he runs a dev blog and actively works on self-development.

I would like to thank my wife Monika for the patience and support she showed while I was writing this book.

About the Reviewers

Deep Shah is a cofounder and director at InformationWorks (`www.informationworks.in`). He received his B.E. in Computer Engineering from Pune University, Pune, in 2008 and his M.S. in Information Management from Syracuse University, New York, in 2011. He was also a research assistant to Prof. Kevin Crowston on the Flossmole project (`http://flossmole.org/`) and has taught enterprise technologies, such as Mainframes, as a teaching assistant at Syracuse University.

Deep is a passionate web and mobile app developer. During his stay in the United States, he had interned and worked as a web developer for the likes of J.P. Morgan Chase, Sidearm Sport (`http://www.sidearmsports.com/`), Think60 (`http://www.think60.net/`), and so on.

He is now a cofounder at InformationWorks and is the lead developer of Android and iOS app:

- CATapp (`https://play.google.com/store/apps/details?id=in.informationworks.app.CATapp`)
- Vedic Math India (`https://play.google.com/store/apps/details?id=in.informationworks.app.vedicmath`)
- What's The Word (`https://play.google.com/store/apps/details?id=in.informationworks.app.WTW`)

Deep has been a .NET developer for more than five years and has worked on various .NET technologies such as WPF, XAML, LINQ, WinForms, ASP.NET, and so on. He actively follows the Windows Phone development technologies and is in the process of developing all their existing applications for Windows Phone 8.

Melania has worked using Microsoft technologies since being a student in college and with mobile products since the introduction of Windows Phone to the market. She worked on many projects as part of the product development and architecture teams for some large companies in Europe. Since being in college, she's started activating in the community having presented at some of the best known regional conferences: DevReach, DevSum, and ITCamp.

Waldemar Sudoł is a senior software developer running his own business and cooperating with international corporations. He develops software for many OSes in .NET technology, mostly specialized in mobile and web solutions.

He develops apps for Windows Phone application from the beginning of the platform. He is the author of tens of Windows Phone market applications. He also developed the Windows Store apps. He's interested in .Net technologies and Windows Store apps.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Mobile applications are one of the fastest growing markets. There are hundred billion dollar businesses on the stage looking for employees as well as individual developers to create bestseller applications.

The Windows Phone market is growing fast, but there are still not as many applications as there should be. This means that there is a lot of room for improvement. XAML and C# are a well-conceived pair that indulges most developers. New features in the latest version of Silverlight provide mechanisms that accelerate and simplify ordinary tasks.

C# 4.5 features such as dynamic programming, asynchronous methods, and optional parameters are just some of the topics described here that developers find useful.

What this book covers

Chapter 1, *XAML in Windows Phone*, introduces XAML, controls, containers, element bindings, and Value Converters.

Chapter 2, *App Design – Best Practices*, covers the best practices that should be used while creating UI in Windows Phone 8 applications.

Chapter 3, *Building a Windows Phone 8 Application using MVVM*, explains what the MVVM pattern is, how to use it, and describes the best practices for creating a testable application.

Chapter 4, *Integrating with Windows Phone*, covers application integration with phone features, such as notifications, storage, camera, and so on.

Chapter 5, *Integrating with Twitter and Facebook*, describes how to implement basic social support in the application.

What you need for this book

Users that want to create Windows Phone 8 applications need to use the Windows 8 operating system because of Windows Phone 8 SDK requirements. Another thing that will be needed is Visual Studio 2012 with SDK 8 installed. Optionally (to simplify UI creation), Expression Blend would be very helpful.

Who this book is for

This book is designed for people who want to get into mobile development. Some C# background may be very useful to achieve full understanding. The book shows the advantages of using modern patterns instead of the traditional way of programming.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "`PhoneApplicationFrame` can contain pages and implements navigation by calling the `Navigate` method or setting the `Content` property".

A block of code is set as follows:

```
public class SampleModel : INotifyPropertyChanged
{
  private string _sampleProperty;

  public string SampleProperty
  {
    get { return _sampleProperty; }
    set
    {
      _sampleProperty = value;
      RaisePropertyChanged();
    }

}
```

Any command-line input or output is written as follows:

```
PM> Install-Package TweetSharp
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—may be a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the **Errata** section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
XAML in Windows Phone

This chapter covers the basic things that can be done using the XAML language and the presentation layer that it depends on:

- XAML introduction
- Basic WP8 controls
- Windows Phone Toolkit
- Working with data

XAML (**Extensible Application Markup Language**) is an important part of the .NET platform that allows us to build rich and beautiful applications. There are a few ways to build a **Windows Phone** application; we decided to follow the XAML with C# path because this technology, despite its richness, is very comfortable to use and relatively easy to handle. The multiplicity of controls and features that we get out of the box with **Visual Studio** is enough to build and publish Windows Phone 8 applications.

As XAML is grammar based on XML, anyone who has a web developer's background will find it intuitive. The main goal of using this markup language is to simplify the cooperation between graphic designer and developers, but it doesn't mean that you need to have designer skills to create an XAML application and vice versa; if a graphic designer wants to help with the application design, it is not necessary to know C# (but it can be helpful). If you are still afraid of XAML code and want to try it yourself, take help of a graphic designer and almost do not touch the GUI code. You should try the Designer tool that is a part of Visual Studio or the fancier Expression Blend. Graphic designers prefer to use Blend because they can create a whole user interface, animations, and so on without writing XAML code. But we will write the code because we are developers!

The XAML specification defines the rules of mapping the .NET classes to XAML objects; for example, the following is how we define a button in XAML:

```
<Button Name="BtnMyButton" Click="BtnMyButton_OnClick" Content="Click
me!">
</Button>
```

It means the same as:

```
System.Windows.Controls.Buttonbutton = new
  System.Windows.Controls.Button();
button.Name = "BtnMyButton";
button.Content = "Click Me!";
button.Click += BtnMyButton_OnClick;
```

But we will never do that! Defining controls, containers, and so on in C# code is against patterns and good practices, and causes my soul to suffer.

Types of XAML objects

XAML objects can be grouped into 6 groups as follows:

1. Navigation
2. Containers
3. List controls
4. Common controls
5. User controls
6. Third-party controls

Navigation

Navigation controls ensure that the Windows Phone application works in a similar way to page navigation in a web browser.

PhoneApplicationFrame

`PhoneApplicationFrame` is a top-level container and only one can be defined for the entire application. `PhoneApplicationFrame` can contain pages and implements navigation by calling the `Navigate` method or setting the `Content` property. I suggest using the `Navigate` method because it adds entry to the `BackStack` enumerable (the list of entries in the navigation history). In this way, it allows the back button to be automatically handled by getting the history elements from the history stack.

PhoneApplicationPage

`PhoneApplicationPage` is a container for UI components that can be used as many times as we need. This is a good and fairly common practice to build a page-based application.

Containers

Containers are very important in application design. It allows us to organize content the way we want to.

Canvas

`Canvas` is one of the `Panel` containers and can contain child elements. Child objects within `Canvas` have their position defined by x, y (`Canvas.Left`, `Canvas.Top`) properties, which specifies the distance in pixels from the left (x) to the top(y) corner of `Canvas`.

Border

`Border` is a container that delivers border or/and background around other controls and can have only one child.

Grid

`Grid` is a `Panel` type container. Within `Grid`, we can create grid columns and rows, and then position objects by the `Row` and `Column` property on the control. Distribution of spaces is really simple using star sizing (means stretching to place that is split equally) or Auto value. `Grid` contains elements that are drawn in the order that they are defined in the container . Using the `ZIndex` property on the elements of `Grid`, we can define which element will appear at the front and at the back.

Panorama

`Panorama` is a long horizontal canvas that contains `PanoramaItem`. It is commonly used in Windows Phone. `Panorama` is an ideal place to put long horizontal content into as it provides a better user experience. The best thing is, we don't need to handle any gestures (swipe, flick) because it does these automatically.

Pivot

`Pivot` is another container that can be used instead of the `Panorama` control. It contains `PivotItems` that provide a quick way to manage pages or other views, just as `Panorama` has gesture handling implemented by default.

ScrollViewer

`ScrollViewer` is a container that hasn't got its own UI. It fits into the integrated area and provides scrolling functionality for UI items, that are present in it.

StackPanel

`StackPanel` is a simple layout panel that arranges controls into a single line or column (depending on the `Orientation` property). It works like stack; new elements go after the old ones.

List controls

Most applications display some data, often lists of items. When we need to show a list, we can use one of the list controls.

ListBox

`ListBox` is a control is used to display a list of items. We can customize the look of each item in `ListBox` (it will be described later). If needed, we can use `LongListSelector`, which has many features in comparison to `ListBox`, such as grouping and scrolling to a specific element.

LongListSelector

`LongListSelector` is a control that gives us flexibility in displaying data. Defining how data will appear is set by the `LayoutMode` property. Furthermore, it supports templates and grouping, and is often used because it performs better than `ListBox`.

Common controls

Mobile applications, though they are completely different, have many common elements. There are a number of controls that are available for developers in the Visual Studio toolbox.

ApplicationBar

`ApplicationBar` is a control that contains icon buttons. The bar is placed at the bottom of the application page and can contain buttons that launch default actions on the page, such as add, delete, save, and edit. The application bar can be shown in the following three states/sizes:

- **Mini size**: Where only three dots are visible and the user can slide up the bar
- **Default size**: Where buttons and ellipsis are visible
- **Full size**: Where buttons, ellipsis, and labels are visible

This control is used in many places in Windows Phone 8 and is really intuitive.

Button

`Button` is one of the most frequently used controls. When the user touches the `Button` control, it causes the execution of the `Click` event method handler that is assigned to it. What is interesting is that we can set when the event will be fired by changing `ClickMode`. For example, if we set `ClickMode` to `Press`, the event handler will be executed when the user taps on it. A release setting (a default setting) action will be launched when the user releases the button.

CheckBox

`CheckBox` represents a Boolean value, a choice between two opposite states, and is often used in a group.

HyperlinkButton

`HyperlinkButton` works and looks like any web hyperlink. Tapping on the hyperlink opens mobile IE or a web view in our application.

Image

Using the `Image` control, we can show standard web format images in our application. Images respond to typical gestures in WP, such as tap, double-tap, pan, flick, and pinch-and-stretch.

A good practice is to display a picture that has a similar resolution to our image control.

MediaElement

`MediaElement` has no interface of its own; that is, it just shows media (audio/video) content. It can be customized by its properties to display the video on a full-screen, and if there is an audio playing, the background sound can be muted. What is important is that only one `MediaElement` control can be used at a time.

MultiScaleImage

`MultiScaleImage` in used in **Deep Zoom** technology. This control allows the user to open a multiresolution collection of pictures or a single picture. Methods for panning and zooming are not implemented by default but should be utilized.

PasswordBox

`PasswordBox` is used to input passwords (for logging in or registering purposes). It is displayed like `TextBox`, but the difference is that the entered letters show only for the moment and are normalized to bullets. The number of displayed bullets is not equal to the number of letters in the `Text` property.

Popup

`Popup` displays content that overlays current content. Using the `IsOpen` property, this control shows or hides the pop up. We can use the `MessageBox` control, which is a less-flexible component.

MessageBox

`MessageBox` is a modal dialog that shows at the top of the application page. `MessageBox` can be displayed by calling the static `Show` method on the `MessageBox` class. Because of that, such controls should not be used since they create the dependency of the user interface in the code.

RadioButton

`RadioButton` like `CheckBox` represents a Boolean value but can be grouped with that, which provides mutual exclusivity.

RichTextBox

`RichTextBox` represents a rich text-editing control that allows hyperlinks, images, formatted texts, and HTML elements.

Slider

`Slider` displays a range of values along the track that the user can select from. The selected position is represented by the `Value` property.

TextBlock

`TextBlock` is one of the most common controls in Windows Phone 8. It displays read-only text, which can be set by the `Text` property.

TextBox

`TextBox` enables the user to input text: it can be a short, single, or multi-line input.

ToggleButton

`ToggleButton` is very similar to `CheckBox` and `RadioButton` and is used to switch states. It is most commonly used in the Settings page.

WebBrowser

`WebBrowser` displays HTML in our application. This control is based on the Internet Explorer 10 engine. Using `WebBrowser`, we are able to display the web content by using the `Source` property. It is good to remember that the control has disabled scripts by default; if we want to use some JavaScript, we have to switch the `IsScriptEnabled` flag on it.

User controls

User controls provide encapsulation of some functionality. Imagine a situation where you created some form and would like to use it on another page—don't even think about copy and paste! The best solution for this will be to create a user control.

User controls are easy to create—just add the new item to the solution, select the WP user control, and put controls into it.

```
<UserControl x:Class="MyCustomControls.UsrControl"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel HorizontalAlignment="Center">

    <TextBlock Text="Hello from User Control!"></TextBlock>

  </StackPanel>
</UserControl>
```

Now imagine you can put a user control into `ListBox` or other list control. For sure, using user controls will save your time, and the code will look clearer.

Third-party controls – Windows Phone Toolkit

Windows Phone Toolkit is based on the Microsoft Public License, so you can use it on the terms of this license. The toolkit library provides many controls that are way better than the common ones and there are some new one with nice features. For sure, using this toolkit will provide a rich user experience to our application, making users feel better about using it longer. The following is a list of some of the most popular controls used in WP Toolkit:

- `AutoCompleteBox`
- `ContextMenu`
- `CustomMessageBox`
- `DatePicker`
- `HubTile`
- `MapExtensions`
- `ExpanderView`
- `LongListMultiSelector` - List layout
- `LongListMultiSelector` - Grid layout
- `ListPicker`
- `ToggleSwitch`

This is not the end! Windows Phone Toolkit also contains a test framework that will be used later when the **Model-View-ViewModel** (**MVVM**) pattern will be described. Why should unit testing be important for developers? Because it is not only a way to test the application; it is a way to create software—writing a code. Agile development practices require us to write and execute unit tests.

Two ways to get Windows Phone Toolkit are as follows:

- Downloading from CodePlex by browsing to: `http://phone.codeplex.com/`
- Executing the Package Manager Console command `PM> Install-Package WPtoolkit`

Working with data

We now know about many controls and containers; now it is time to populate them with some data. As always, there are multiple ways to put data— textbox, button, listbox, and so on—but, following the modern best practices, I will go with data binding. Data binding defines the way the data is linked to the properties of `Model` and interacts with the user. Using this mechanism, data is separated from the code that manages data. There are a few things that are important to know when starting to work with the data in Windows Phone. We will be dealing with some of them in the following section.

Binding expressions

Imagine you have just created a WP8 solution, dragged-and-dropped the `TextBlock` control into the page, and now you need to tell the textbox what it has to display. As always, there are multiple ways to do that. First is the binding expression that is used in XAML. Binding expressions should be defined between curly braces "{" and "}".

To visualize this method, let's define the class `Customer`, which will be used further in this chapter, and call it `Model`.

```
public class Customer
{
  public string Name { get; set; }
  public string Lastname { get; set; }
  public string Email { get; set; }
  public DateTime DateOfBirth { get; set; }
  public bool IsMarried { get; set; }
  public Country Location { get; set; }
}
```

Consider a situation where you want to bind the `Customer` class property to the `TextBlock` control that you have already defined. Using simple binding can be done in a few ways, as seen in the following code:

```
<StackPanel>
  <TextBlock Text="{Binding Path=Lastname}"/>
  <TextBlock Text="{Binding Name}"/>
  <TextBlock Text="{Binding Country.IsoCode}"/>
</StackPanel>
```

The preceding example shows how to link particular properties from `Model` (the `Customer` class) with the `Text` property of the `TextBlock` control. The first and second code lines have the same meaning; the path keyword can be omitted in simple expressions. The third example is a bit more sophisticated and binds the `IsoCode` property from an embedded object in the `Customer` model.

DataContext

Most controls have the `DataContext` property. `DataContext` is accessible from the code and setting it will tell the XAML object which model it should use. Binding expressions that we have defined previously while resolving the value will look for the data context from leaf to root, meaning from `TextBlock` to `StackPanel` and next in any parent container.

```
var customer = new Customer()
{
  Name = "John",
  Lastname = "Smith",
  DateOfBirth = new DateTime(1988, 1, 3)
};
StckContainer.DataContext = customer;
```

Now, the model is linked to the container but it can be any other parent container or `UserControl`.

Element-to-element data binding

During development, we will often face a situation where one element will depend on another element's property.

```
<StackPanel Name="StckContainer">
  <Slider x:Name="SliderSize" Minimum=
    "10" Maximum="72" Value="12" ></Slider>
  <TextBlock Text="{Binding Path=Lastname}"
    FontSize="{Binding Value, ElementName=SliderSize}"/>
</StackPanel>
```

The preceding example shows a `Slider` control that manages the font size of `TextBlock`. Using this type of binding, we have to initialize the `ElementName` property with a name and path to attribute of dependent control. This is a very simple demonstration; such bindings are used in much more advanced situations such as hiding UI elements or changing the control's content.

Binding mode

There are three binding modes that can be defined in the binding expression as follows:

- `OneTime`: This is the default binding mode. It populates a property with data when loading for the first time.
- `OneWay`: This should be used for read-only controls. When the model changes, the control (UI) property will be updated.
- `TwoWay`: This should be used within the editable form, where a user has to input some data. For example, the `Name` and `Lastname` textboxes. It updates the model and UI automatically.

The `OneWay` and `TwoWay` modes need the `INotifyPropertyChanged` interface to be implemented in a model that is linked to the control.

INotifyPropertyChanged

Why do we need the `INotifyPropertyChanged` interface? It implements events in our model class that will update the UI when the model changes (for example, when the `Name` property in the `Customer` class changes). This behavior is really useful when the application updates data in the background; changes will be directly seen in the UI.

More about the `INotifyPropertyChanged` interface implementation and binding to property will be covered in *Chapter 3*, *Building a Windows Phone 8 Application using MVVM*.

Value converters

Sometimes, developers need to display show data differently from how it is stored in a model or database. If you ever experience this, remember value converters. `IValueConverter` is an interface that implements the following two methods:

- `Convert`: It gets the property and processes and returns the converted values
- `ConvertBack`: It works in the opposite manner to the `Convert` method

A good example of converting the data is the `DateTime` converter. To define the converter, we have to remember two steps. First, create a class that implements `IValueConverter`, and then use it in XAML in the control resource. Out of experience, I suggest creating such converters in one location or library to keep our code clear and reuse it them in many projects.

List binding

Another very important topic is binding a list. It is hard to even imagine an application without lists. Lists or collections, which are pretty much the same, can be linked to list control using a binding mechanism. It is very easy to implement in a markup definition.

```
<ListBox Name="ListCustomers" ItemsSource="{Binding}"
  DisplayMemberPath="Name">
</ListBox>
```

The trickiest part goes in code. Remember when we were talking about `INotifyPropertyChanged`? We can create a list of any objects, bind it to the list, and the list will automatically handle updating of the UI. `ObservableCollection` is a generic collection that provides notification when the items get removed, or added, or when refreshed. Look at the following code and see how simply I did that!

```
ObservableCollection<Customer> customers = new
  ObservableCollection<Customer>();
ListCustomers.DataContext = customers;
customers.Add(customer);
```

In this example, the collection was created, linked to the `ListBox` control, and one object was added to the collection in code. Because of the features of the observable collection, the UI gets updated because of implicit binding.

Summary

In this chapter we were talking about XAML controls, updated UI, and took the first few steps in working with data presentation. Readers should now know which and how controls can be used, what data binding is, and know about value converters.

The next chapter will be less technical; there will be more about planning, inventing, and designing our application.

2

App Design – Best Practices

This chapter explains how to create a UI for a Windows Phone application. The following things will be explained in this chapter:

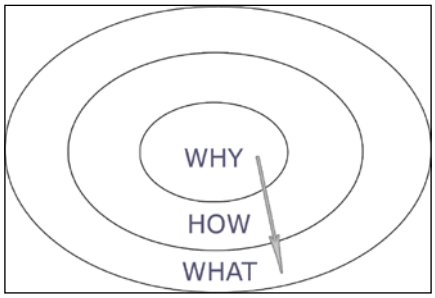
- First impression
- Golden circle
- Planning
- Commands and navigation
- Touch
- Branding
- Principles for user interface (UI) / user experience (UX)

First impression

Many projects end tragically or will never be in use because of poor user experience. Why is it so important? It is scientifically proven that people form an opinion after a few seconds of using an application. If the application looks complicated, uploads too slowly, or seems old fashioned, a user will uninstall it and the application gets a 'one star' rating in App Hub (the Virtual Microsoft store with applications, games, music, and movies). Even if an application does magic on the inside but feels uncomfortable to use—only a few users will patiently continue to work with it. We must think about how a user thinks and feels when we launch an application for the first time. If the first impression fails, then we won't have a chance to tell the users how great our application is.

The golden circle – people don't buy what you do, people buy why you do it

Simon Sinek had a great talk on **TED** (**Technology, Entertainment, Design**) about how to build our products and how to make them better. He invented this concept called the Golden circle, which illustrates the phrase: "People don't buy what you do, people buy why you do it."



We know what we want to do, we know how do it, but do we know why we do it? Why do we build an application and what is the goal? For sure, the answer is not revenue; it is only the result. Look at the circle and how it relates to what a producer usually thinks: "I've got software, it works great, is well designed, buy it". However, great leaders think: "I believe my software will change your life for the better with its great interface, simplicity, and beauty" (Simon Sinek, "How great leaders inspire action", TED, 17 September 2009). The goal is to create something that people feel they want to use; we need to talk to their feelings.

Why? how? what? – planning

Many people usually omit this topic, but we want to be professional so we plan our work on our application. The first thing that we have to think of is how the user will feel using our application and why he needs it. Spend a few hours thinking about who will use our application and why will he/she do that?

For example, let's think about an application that will help you to keep fit, lose weight, and track progress.

Who will use it?

- Sportsmen
- Plus-size people
- People who want to lose some weight

Why will they use it? Because they believe that our application will help them to:

- Lose weight
- Record their progress
- Highlight the weak and strong points of their training
- Create goals that they can achieve

Now decide what our application will be great at. Prepare many user scenarios and focus on the users' activities; what are the things that they will do most often? At this stage, we should not think about features that our application offers but about what the users will be able to do with it. Brainstorming or mind-mapping techniques (a mind map is a diagram that helps to visualize concepts and ideas) are a nice way to generate ideas and processes we should consider.

An example of a user's story could look like the points listed as follows:

1. A user creates a goal that he wants to achieve in the next month.
2. He/she enters information about the exercises he/she does.
3. He/she enters information about his/her weight.
4. Watches his/her progress.
5. Reviews suggestion about his training.

From these 5 points, let's decide what activities a user will do regularly on a daily basis. For sure, he will enter information about exercises and about his weight, and that is the point that will affect our application's UI and navigation. At this moment, we know why we want to create the application (to help a user keep fit) and what we want to do. But there is a problem, we don't know how to do it. Let's look at a few possible ways to plan the application's most important part: UI and UX.

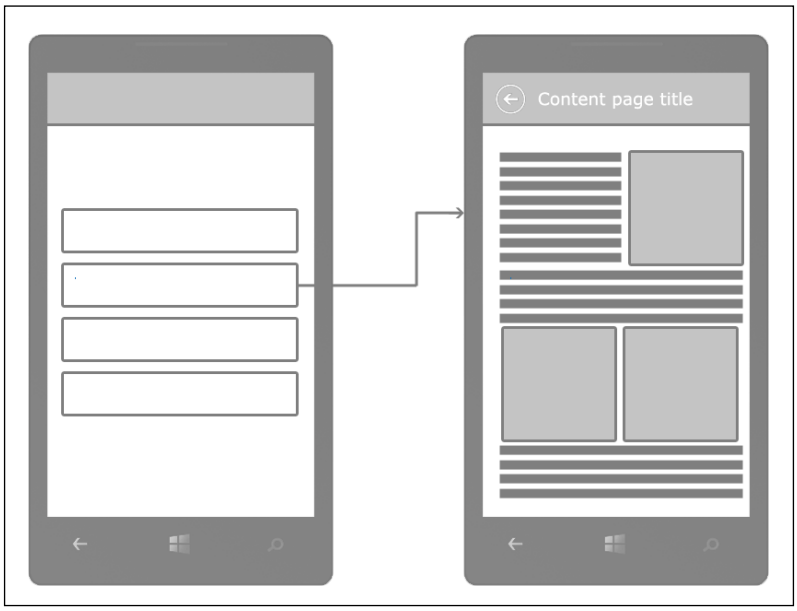
Commands and navigation

Organization of the application's content is very important, especially when the available space is very limited. Using proper navigation patterns will cause our application to contain fewer controls and be more intuitive. Users have to focus on the content and not think about navigating from one point to another within our application. Another thing to discuss is consistency. Our navigation must be consistent to prevent the users from getting confused. If we plan to use a navigation system (even if it is completely customized), we should apply it to all places in our application.

Here are a few patterns that describe how the Modern UI application should navigate:

Flat navigation

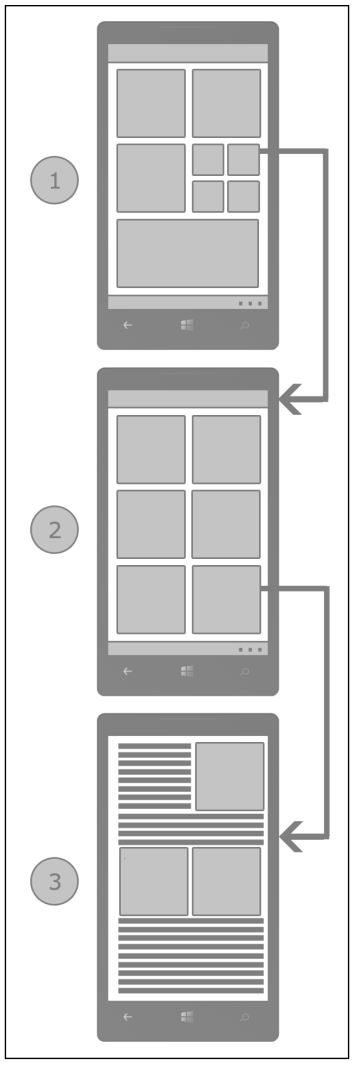
The flat navigation system is the simplest one that can be used to navigate with simple data content. If our application does not require data with complex structures, then the flat navigation system can be a good solution. This type of navigation system is very simple; it contains an application menu that can be presented as a top menu or a main menu on a separate page.



The preceding image shows a sample use of the flat navigation system. In the preceding figure, we can see the main menu that navigates to some content. It is really simple, but believe me it is very common because it provides intuitiveness. A good example of an application that uses flat navigation can be our weight loss app, because all we need within this application is to add weight check, enter exercises, show progress/statistics, and define goals. All of those things can be separated on the main menu and point to the proper content pages.

Hierarchical navigation

The hierarchical navigation system is another very common navigation system ; to be honest, most Windows Phones use this system. This type of navigating in a complex application with its rich content is fast and fluid, while still being easy to use and intuitive. The hierarchical navigation system helps the user to navigate from the main menu by grouped parts of content to details. A sample hierarchical navigation system is defined as follows:



Hub controls

After the application launches, we can see the entry point of the application. This is a known view in Windows Phone. Using large tiles, we can suggest more important places in the application or the places that will be used more often. In the preceding figure, we can see that the tiles are limited with an application bar at the bottom and a header at the top. Modern UI applications use this page schema and, for sure, it will be intuitive to the users. Considering this type of navigation in the sample application, we can imagine an application that aggregates data from social portals (Facebook and Twitter). The entry point to such an application can contain tiles with a number of new messages, contacts, photos, or events.

Groups or section tiles

The second level of the hierarchical navigation system can be presented in a way that best represents/groups the content of the application. It is not the most detailed part of the application, thus needs to contain only basic shortened content. Each of the tiles presented in this view has its own detail page. If there are many tiles needed we can expand the canvas using `Panorama` or `ScrollViewer`. The example sections can contain our last conversations ordered by date, represented by tiles with a contact photo along with the tile's background color, which depends on whether the message has been read.

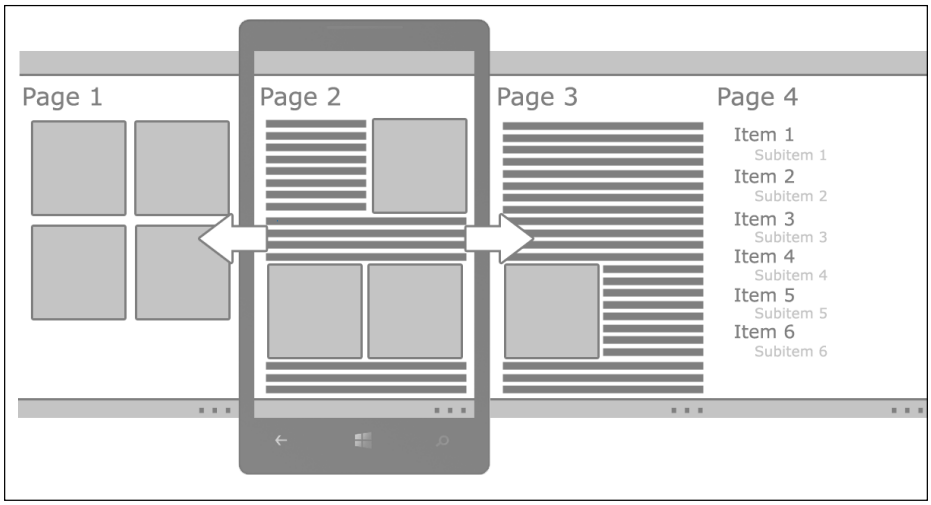
Details

The third level of the application (but it doesn't have to be last one) should contain detailed information about the objects that we chose before. It can include a single picture, movie, or article. Moving forward with the social aggregator idea and message path, this is the place for a whole conversation with a particular person. This page consists of simple content as described earlier, but this is the place for more complicated and complex functionality as well.

A rich data application that contains plenty of pictures and text, for sure, will need scrolling and according to content type we can use `Pivot`, `Panorama`, `ScrollViewer`, or any other control that would help us to organize our content.

Pivot

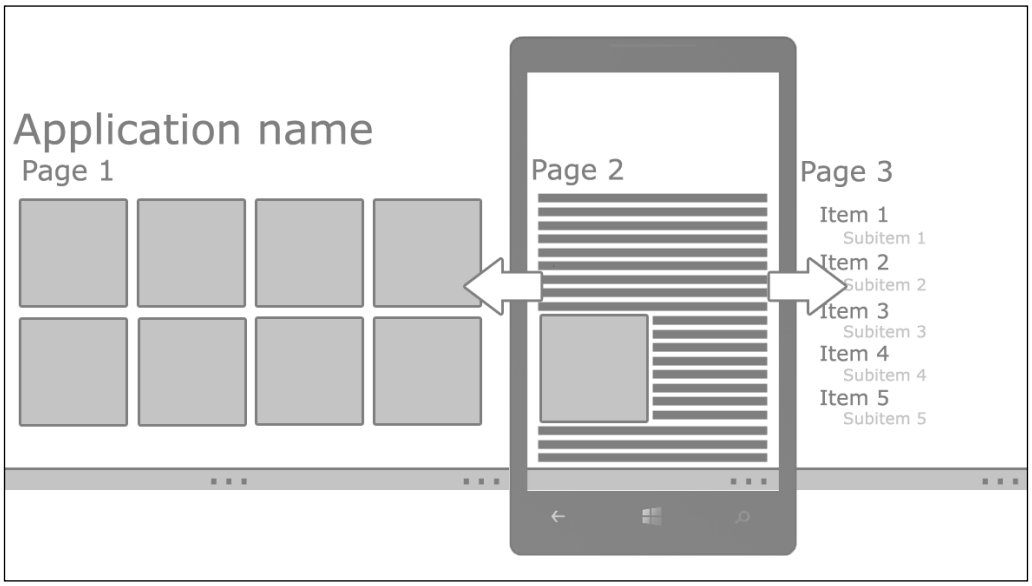
`Pivot` navigation is the next navigation system that is commonly used in Windows Phone 7 (this is a older version of this mobile system and is still very popular in Windows Phone 8). As the name suggests, it is based on the `Pivot` control that defines the application's entry points and provides access to the application's functionality.



As we can see in the preceding screenshot, the `Pivot` type navigation provides out of the box functionality; it is only required to create a project with the `Pivot` application, using the Visual Studio template. Each page is a separate `PivottItem` container and can be easily swiped/flicked between each other. `PivotItem` is a container that can be defined only within the `Pivot` control and it can contain only one element that is another container usually. There is no problem to embed the user control within `PivotItem`. Some developers complain that `Pivot` and its `PivotItems` are defined in one page and all the code that creates controls and actions must be located on one page. A solution for this problem can be using user controls for each `PivotItem`, as it will reduce the number of lines of code for one page and will split it to different places. From a maintenance point of view, it is very important to keep the code well organized. Of course, this is the entry point to the application, so navigating to any other pages is fine as long as we maintain the consistency.

Panorama

`Panorama` is the last navigation system described in this book. The `Panorama` control provides a unique ability to view long horizontal content in a very attractive format and it is actually a long horizontal canvas that contains `PanoramaItems`. Those items are hosts for some content sections and the user can navigate between these using supported touch gestures.



The `Panorama` control gives a nice effect when having a defined background; every Windows Phone user has seen the photo gallery on his or her phone, that is based on `Panorama`. In another example, a user can contain `PanoramaItems` with a list of messages while another has a set of thumbnail images. As with the `Pivot` navigation, `Panorama` can be mixed with other navigation types, depending on whether the content is using a hierarchical or flat navigation system.

The `Pivot` and `Panorama` controls look very similar but have a different purpose. The `Pivot` control is suggested to be used in an application that contains a lot of data, where the content can be split-up into several chunks with a similar or slightly similar look. The difference is that `Panorama` is made to present the content in a more artistic way; it makes the user interested enough to flick and see what the next item is. Content within the `Panorama` items should not look the same; the more different the better.

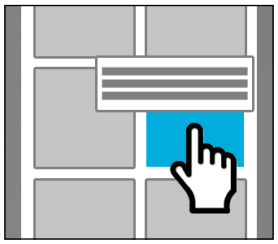
Application bar

We have many places in our application where we can define commands. As we can see in most of the pictures here, there is an application bar implemented. Application bar by default is a row of icons at the bottom of the application canvas. It has defined animation for swiping and orientation change; for example, if the user changes the landscape and orientation of the phone, the application bar adjusts automatically by moving to the side of the screen vertically. Basically, we use the application bar to display a user's on-demand commands that are related to the current context. For example, if one of the pages within an application contains a set of recently downloaded content from our Facebook wall, the application bar should contain basic actions that the user can perform such as add, upload a picture, and refresh. Many applications have the same icons for basic commands in Windows Phone 8 because there are some icons provided with the SDK and located at: `C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v8.0\Icons`.

As we saw in *Chapter 1, XAML in Windows Phone*, the application bar may be displayed in three states, that is, mini, default, and full-size. In following example, we can see a full-size state that was reached by swiping the application bar from the bottom. Additionally, we can add text-based menu items to the icon buttons. Actions defined in menu items should be used in less-frequently-used scenarios. The application bar does not support data binding; thus, if we want to change the content of the application bar, we need to populate it in C#. Using an application bar provides consistency across the whole application, which is very important.

Context menu

The next thing that is good to have in most applications is context menus. Press-and-hold on an item should cause a content menu to show up. The system provides basic action on texts and hyperlinks, but we can define commands that will work for other elements. For example, on an image thumbnail we can define commands such as share, delete, and crop.



Touch in the Windows Phone 8 application

Windows Phone users have no hardware such as a keyboard and everything is done by touching a screen. We need to ensure that the user will feel good using our application; he or she needs to be able to complete the core actions by using touch. Let them manipulate the content of our application; it should be easier than commands.

Touch and gestures

For sure, our users will appreciate the better user experience. Here are some common touch events that an application usually handles in its controls as follows:

- **Tap on item**: It causes invoking of primary actions such as going to a full-size image or message details
- **Press-and-hold on item**: It shows a context menu or tooltip (if such is defined)

- **Slide**: It is used for moving items, switching radio buttons, or moving between items in `Pivot` or `Panorama`
- **Swipe for application commands**: It shows an application bar that contains commands depending on the context

Target size guidelines

The next thing that is very important in touch displays is the elements' size. Imagine a situation where we want to tap on an element but it is too small or has too thin a margin and we miss the click. Isn't it frustrating? We are not going to make our users suffer because of these bad practices. There are a few guidelines for the elements' size as follows:

- 30 x 30 pixels elements size is the absolute minimum and can be applied only to the elements that are not very important.
- 40 x 40 pixels is the minimum from a good-practice point of view; it gives the best resolution for selecting elements.
- 40 pixels height is used for Windows phone buttons.
- 50 x 50 pixels is used for the most important and critical actions. Size reduces the probability of touching a wrong target.
- A 10 pixels gap between elements is extremely important when using 5 x 5 pixels elements; if you want to reduce a gap it is your choice but, for sure, the users will not feel comfortable using such small elements.

It is very important to follow these rules in every case. For instance, if we cannot place a button in the application bar and we need to put it into content, we have to remember about padding between the button and content (text, image, and so on).

Branding in the Windows Phone application

Let's talk about creativity. All of these design patterns, guidelines, and good practices can make you think that there is no place for creativity. Wrong! Even if the application is well-planned and the developer has used all the design patterns, still his or her application cannot be popular or even seen. The real value of each application is in its uniqueness and recognizability. The main elements that we have to provide are unique logotypes that will be associated with our application. Our goal is to make the users think about our application when they see similar shapes or colors in their environment.

How to do that? Our brand can be expressed by several visual elements as follows:

- **Colors**: This visual element is the key of each brand. For example, look at the Facebook logo, it is very simple but the Facebook blue color is highly recognizable.
- **Logo**: This is the visual element that we are going to use to identify our application and make it recognizable.
- **Graphics**: It is the custom elements that will help to make our application unique but a thing to remember is that too many graphics will confuse our users.
- **Layout**: Arrangement of controls and other elements in our application should be a result of the type and the most frequent actions that a user executes.

Simplicity is not bad!

Simplicity is not bad; I can say it is an advantage. Every worldwide brand goes simple. I mentioned that Facebook was simple and clear from the beginning; Windows simplified its logotype in Windows 8. Look at Visual Studio 2012; it is monochromatic and simple even though it has plenty of functionalities.

Principles for UI/UX

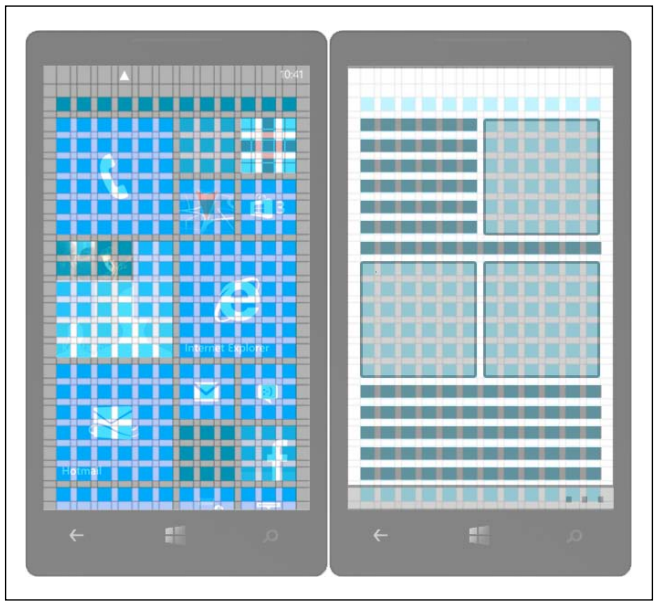
There are many dos and don'ts in Windows Phone UI development. It is good to remember that we have a screen with a limited size where the elements should be placed comfortably.

Being fast and fluid

This is the main goal of Windows Phone UX. Starting with the phone menu, we can switch between tiles and application lists; if we touch an element action it is performed very fast. We have to remember while creating our own software about providing responsivity to user interaction; even if we need to load some data or connect to a server, the user has to be informed about the loading. Going through our application, the user should feel that he or she controls and can touch every element. Many controls provide animation out of the box and we need to use it; we need to impress our users while seeing motion in our application. The application has to be alive!

The grid system

The following design pattern provides consistency and high UX to Windows Phone users. It helps in organizing the application elements. A grid system will provide unity across applications and will make our application look similar to the Windows Store application. Talking about the grid layout, we mean arranging the content using grid lines.



As we can see, the grid is where the application design starts. Each square is located 12 pixels from the other square. A single square has 25 x 25 pixels. In order to provide the best user experience to our application, we need to ensure that the elements are arranged appropriately. Looking at the left-hand side of the image, we find that each big tile contains 6 x 6 squares with 12-pixel margins.

Windows is one

When using a grid system and other design patterns for Windows Phone, porting to Windows Store will be possible and can be done more easily. Unfortunately at the moment, there are no common markets for Windows Phone application and Windows Store (applications for Windows 8 and Windows RT). The current solution is to create the application in a way that allows portability to other platforms, such as separating the application's logic from UI. Even then some specific changes have to be done.

Controls design best practices

`Button`	• Use a maximum of two words in the `Button` content; it can be dynamically set but should never be more than two words • To define content use system font • Use a minimum height of 40 pixels
`CheckBox`	• A `CheckBox` text should present a clear choice to the user • If there are many choices, use the `ScrollViewer` and `StackPanel` control • It is recommended to use single or a maximum of two lines of content text • If the meaning of a checkbox is not clear, use `RadioButton` or `ListBox`
`HyperlinkButton`	• Don't place two hyperlinks close to each other because it will make it difficult to select either one • Hyperlink text should not be longer than two words • If some action can be reached by taping in text content, use `HyperlinkButton` instead of `Button`
`Image`	• Size (resolution) of the picture has to be really close or identical to the `Image` control size • Use gestures if it is possible • The `Image` controls displays JPEG or PNG images • Provide good-quality images—non pixelated
`ListBox`	• Use `ListBox` with a long list of items; for smaller lists use `RadioButton` • `ListBox` should be responsive so that it provides an immediate visual reaction for the user action • Use at least 12 pixels height with sans-serif font
`MediaElement`	• Do not show too many controls on one page • Do not allow for a situation when more than one media (audio or video) is playing simultaneously

`Panorama`	• The `Panorama` control handles only portrait orientation • If you are using `ApplicationBar` with `Panorama`, make sure that the `ApplicationBar` control is in minimized mode • Do not use too many sections (`PanoramaItems`); use a maximum of 5 sections to avoid torturing the user with swiping • `Panorama` title can contain image and/or text • Use system font; an exception is when your brand uses custom font • First `PanoramaItem` should contain a left-aligned title • For section titles use plain text • Do not use `Pivot` in `PanoramaItem` and `Panorama` in `PivotItem` • Horizontal scrolling can be difficult because of the default behavior of `Panorama`; use vertical scrolling instead • Good quality background photography makes for a very good users' impression but ensure that all content is readable • Background should be low contrast and dark • Background image can be maximally 1024 x 800 pixels and minimally 480 x 800 pixels to provide good performance
`Pivot`	• Do not place `Pivot` in `PanoramaItem` • Use a maximum of 4 `Pivot` items • Each `PivotItem` should not contain completely different actions; their functionality should be related • Do not use input controls because it disrupts the `Pivot` flick, if you want to create a form with editing controls, use a separate page that will contain the form • Do not use controls that need swipe or scroll inside `Panorama` such as Map, `ToggleButton`, or `Slider` because the `Pivot` default gesture handling can make it difficult for the user to use them

`ProgressBar`	• It is highly recommended to use a label that describes the state of the `ProgressBar` control—loading, launching, and failed
`RadioButton`	• Avoid customization • Provide 2 short words as content text, with at least 12 pixels height maximally • Use `ListBox` if this control has more than 10 decisions to make • Bring the user's attention to the selected `RadioButton` control by making inactive those that are not selected
`ScrollViewer`	• Avoid creating a situation where a user has too much content to scroll
`Slider`	• Make sure that the `Slider` control is the appropriate size for comfortable use • Do not use `Slider` as a progress indicator • Do not put `Slider` too close to the edge of the screen because it can be difficult to select a value • It shouldn't be used within `Panorama` or `Pivot` because `Slider` uses dragging gestures whose usage is difficult in those containers
`TextBlock`	• Make sure that the text in the `TextBlock` control looks clear and is readable
`TextBox`	• To improve the user's experience, update the application content when the user enters text into `TextBox`; for example while filtering data • It is not possible to scroll rich text inputs • When it is likely that the user will enter a new value into `TextBox`, select all content on focus
`LongListSelector`	• `LongListSelector` is used for a large number of data items • Use at least 12 pixel Sans serif font

Fonts

From the GUI and UX point of view, it is critical to make text clear and legible. Entire text should be properly contrasted to the background. The font used in Windows Phone is Sans serif, which is widely used in web and mobile applications. The Sans serif font type has many fonts; which one should we use? There is no simple answer for that, it depends on the control and context in which we use the text. However, it is recommended to use the Segoe UI, Calibri, or Cambria font types.

- **Segoe UI**: This is best used in UI elements such as buttons, checkboxes, datepickers, and others such as these.
- **Calibri**: This is best used for input and content text such as e-mail messages. When using this font it is good to set the font size to 13.
- **Cambria**: This font type is recommended for articles or any other big pieces of text; depending on the content section, we can use 9, 11, or 20 pt Cambria font.

An example of each of the these fonts is as follows:

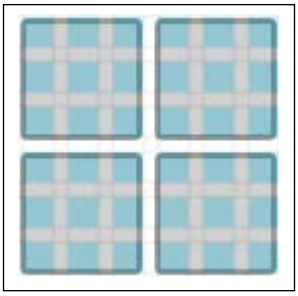
- 11 pt Segoe UI font
- 13 pt Calibri font
- 11 pt Cambria font

Tiles and notifications

My friend, who is the owner of a company that creates mobile applications, once said to me that 50% of an application's success depends on how nice and good looking the icon/tile is. After thinking about it, I imagine he was right. The tile is the first thing that a user sees when starting to use our application (remember what I said about first impression?). The Windows Phone tile is not only a static icon that represents our application, but it can be live and show some simplified content as well. A default tile is a static icon and can display some information such as a message count after it updates, and it returns to the default tile only after another update.

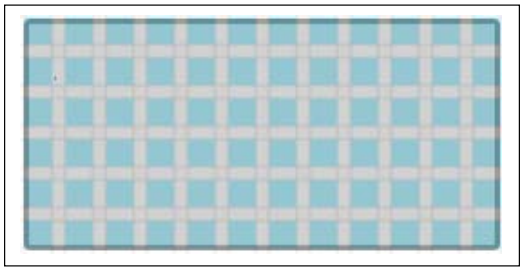
An application tile can be pinned to the Start screen and set in one of 3 sizes:

1. A small tile is built from 3 x 3 grid units. It is half the width and height of medium tile. It was introduced in Windows Phone 8 and the 7.8 update of Windows Phone 7.



2. Only a medium-size tile was available in Windows Phone 7. Every application pinned to the Start screen starts with a medium tile.

3. Along with the small tile, a large tile was introduced in Windows Phone 8.



Why we should use the tile that is updating (live tile)? Because the tile is the front door of our application and if it gives some notification to user it really says, "Hey! Come here and see what I've got for you!". An updated tile assures the users that our application is fresh and active, even if it has not been run for some time.

- If we want to allow the user to pin our application in a large/wide tile, we should provide a wide image, point it in the application manifest, and mark the **Supports wide size** option. There are some good practices that are worth following.
- If our application has content that is refreshed at least once every few days and can be interesting for the user, we should enable our application to use wide tile; if not it is not necessary.
- Forget about the wide tile if the application doesn't use notifications.

There are three available tile templates that will help us to create tiles as follows:

1. Iconic template that is mainly used for e-mails, messaging, RSS, and social networking apps.
2. Flip templates that are used in an application give the user a lot of information such as a weather application.
3. Cycle template for galleries and photo applications that cycles 1 to 9 images in a tile—applicable only for medium and wide tile.

Summary

In this chapter, we have seen how to create a fast and fluid design of a mobile application. One of the most important things we got to know is planning and how to introduce harmony and cohesion in our application. Creating complex applications that will be intuitive and simple to use is really hard without the knowledge of the UI and UX principles that were described in this chapter.

The next chapter will cover the technical point of view of creating a Windows Phone application, how to use MVVM pattern, and more advanced WP8 programming techniques will also be explained.

3

Building a Windows Phone 8 Application using MVVM

In this chapter, we will put our attention to more technical cases than those that were described in the two previous chapters. As a result of this chapter, we will start working on an application that will be extended in the next chapters. **Model-View-ViewModel** (**MVVM**) is a pattern that separates a view's logic from an application's logic and allows us to create highly maintainable and testable applications. It sounds good, doesn't it?

An easy (and lazy) way of creating Silverlight applications was by dragging the control from the toolbox to the design surface and then handling user interactions in the code behind. It can be a good solution for very small one screen applications with poor business logic. Problems can appear when we will like to extend such applications to improve and introduce features or just make some changes to the layout. When we have the view separated from application logic, we can modify the user interface easily and not affect the application logic. What can be very useful from the manager's point of view—in some part of the application, the designer and developer can work in parallel and the developer doesn't need to know how the application should look like but only how it should work. I mentioned about testability; that is, because we have individual layers of an application separated, we are able to introduce unit tests into our project. There are people who think that creating tests for applications is a waste of time and resources, but such an attitude is not applicable in projects created and managed by professionals. For projects, which will either be extended in future or just be under maintenance automatic tests are critical. Unit testing-enabled projects are costly at the beginning, but a lot less time is wasted in looking for bugs and fixing them in the future. If you think about introducing testing in your project, I will say you should, and it is really worth it.

- The project structure
- View

- ViewModel
- Model
- Bindings
- MVVM Communication
- Data Templates
- Converters
- MVVM Toolkit
- Unit testing
- Building Social Aggregator application—SociAgg

The project structure

After creating the project in Visual Studio from the default template, we get Solution including one project. This project is ready to be launched in the emulator or deployed to the device — but has no functionality; it is our job to change this. From the beginning, the project contains the `App.xaml` file.

The `App.xaml` file and its code behind take care of the following:

- Declaration of global resources
- Handling application lifecycle events
- Unhandled exception detection

Global resources declarations can be defined in `App.xaml` under the `<Application.Resources>` node, and their main advantage is accessibility—they can be reached from any place in the project. Global resources can be implemented as control styles, setters, and animations. However, we should use our common sense because application-level resource could impact the performance of our application. Why? Because global resources are loaded before any page is loaded while the splash screen is showing. If we will have tons of animations and styles as a global resource, we can notice that our application is loading, and loading, and loading.

Lifecycle event handlers are defined in `App.xaml.cs` but are left empty. It is our task to make some changes here. The `Application_Launching` handler method is called each time when the application is launched and `Application_Closing` is called when the user decides to close the application. The `Application_Activated` method is called when the application is resumed. This appears at the foreground and is freezed. The last method, that is, `Application_Deactivated`, is executed when the application appears at the background and is freezed. What can we put in these methods? We can read/save the application state and data or make some initializations.

Exception detection is done by handling the `UnhaltedException` event. `Application_UnhandledException` is called each time our application raises an exception and we need not care about handling it in our code. This method can be (or even should be) implemented in a way that, even if the exception occurs, the application is able to recover and give feedback to the user.

Folder structure

Proper organization of files within the project is necessary to keep things clear. If we come back to our project after a year or two, we will know where to find each part of the application. The first few steps after the creation of the new project are the best time to add the three folders—Model, View, and ViewModel. Each of them will contain the corresponding part of MVVM.

View

A view in an application defines a layout, positioning of controls, animations, and everything that a user can see on the screen. The main goal for a view is to represent data and functionality in a user-friendly way. The ideal situation is when a view is defined completely in XAML and very briefly in the code behind. Some code-behind code is required, so we cannot completely omit it.

Windows Phone applications have some typical views that are usually used, such as page or user controls. You can ask, if we have the view separated from other layers, how can we introduce interactions or populate data within an application? Here comes **binding**—a mechanism for connecting interface with data or logic sources and commanding for invoking methods. Each view has a `DataContext` property that is a way to reference `ViewModel` to a view and allow using exposed properties from `ViewModel` and invoking methods on it.

As I said earlier, view defines how data is presented; however, when we want to show data differently than what is exposed from `ViewModel`, we can use `ValueConverters` that will make the conversion in a way we want to.

There are situations when code-behind can define UI; for example, when it is hard to define some login in XAML, or it needs to refer to other UI controls.

Model

Model is a description of the domain model that contains all characteristics that may describe some objects. In short, model represents the data. The main goal of creating models is to encapsulate data, validation, and business rules within classes. It is independent from `ViewModel`, `View` and is not designed to store any visual information, but it can be a data entity; for example, taken from database or web service resources.

Model classes use the `INotifyPropertyChanged` interface in notifications about changing a property value. All we need to do is to implement the `INotifyPropertyChanged` interface in our `Model` class and raise an event each time the property is set.

```
public class SampleModel : INotifyPropertyChanged
{
  private string sampleProperty;

  public string SampleProperty
  {
    get { return sampleProperty; }
    set
    {
      sampleProperty = value;
      RaisePropertyChanged();
    }
  }

  private void RaisePropertyChanged(
  [CallerMemberName] string caller = "")
  {
    if (PropertyChanged != null)
    {
      PropertyChanged(this, new PropertyChangedEventArgs(caller));
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;
}
```

In this example of a simple model, we defined one property that will be exposed. Model implements the `INotifyPropertyChanged` interface. The setter of `SampleProperty` raises an event that updates the UI. A sample situation of usage is as follows:

1. Bind `SampleProperty` to the **Label** text.
2. Update `SampleProperty` in code (for example, when a web service call is completed).
3. After the `PropertyChanged` event is raised, **Label** gets updated.

This is one of the ways of communicating between Model, ViewModel, and View.

If our model contains a collection of another object that will require notifications, we must use `ObservableCollection<T>`.

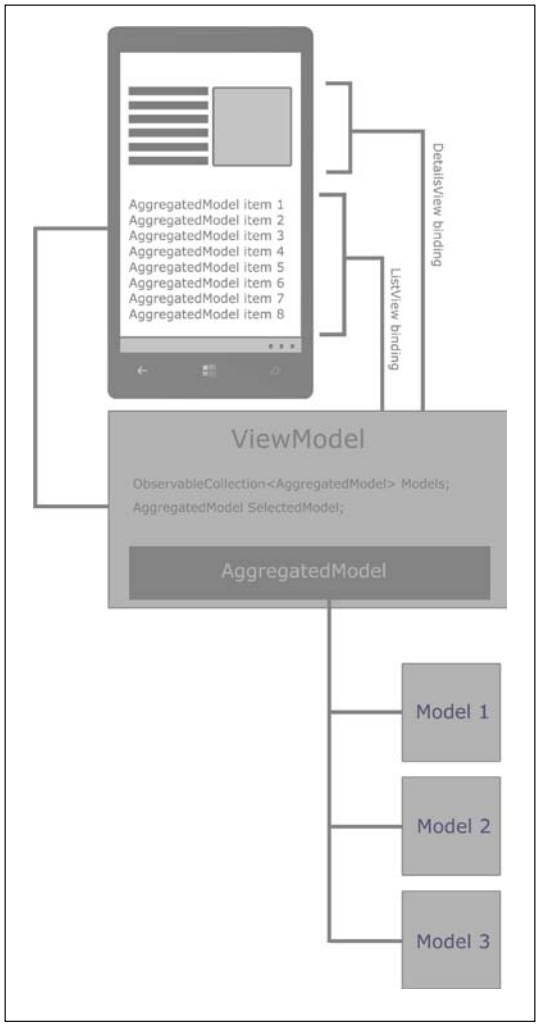
```
public class SampleModel
{
  public ObservableCollection<string> SampleObservableList;
}
```

Each time an item will be added or removed from the `ObservableCollection <string>` `SampleObservableList`, the notification will be provided and UI gets updated with new values. As we can see, it is very simple to use this collection where all notification functionality is encapsulated in the `Observable` class.

ViewModel

`ViewModel` is a connector between `Model` and `View` and provides data and stateto `View` that is bound. It is also known as **View's Model** (**VM**) because it exposes public properties and commands to a view. `ViewModel` takes the responsibility and transforms the data by manipulating `Model` or even aggregates many models and exposes them to view. A possible situation is when view will need some property that not exist in any Model; for example, when loading data, the `IsLoading` property will be needed that will tell the view that loading is in progress. You may ask why not expose a property that directly says to hide the element? Because `ViewModel` is designed to be independent; if we want to handle exposed properties in the view, we have to implement `ValueConverter`.

Another thing for which `ViewModel` is very useful is loading the data. I cannot imagine a better place for loading and populating model data than VM methods; for instance, it may be implemented by loading data from web services. VM executes methods for loading and sets the model's properties, aggregates, transforms, and does required stuff.



The preceding diagram shows how View-ViewModel and Models can be connected and related in the sample screen. This is just one of the options where view could be related to `ViewModel`.

A big advantage of VM is transforming the data. We can use transforming not only to wrap or modify `Models` but also for aggregating properties.

```
public class SampleTransformDataViewModel :
  INotifyPropertyChanged
{
  public string FullNameOfSelectedItem
  {
    get
    {
      if (SelectedItem != null)
      {
        return string.Format("{0} {1}", SelectedItem.Name,
          SelectedItem.LastName);
      }
    }
  }
  private SampleModel selectedItem;
  public SampleModel SelectedItem
  {
    get { return selectedItem; }
    set
    {
      if (Equals(value, selectedItem)) return;
      selectedItem = value;
      OnPropertyChanged();
      OnPropertyChanged("FullNameOfSelectedItem");
    }
  }
  public event PropertyChangedEventHandler PropertyChanged;
     (…)
}
```

Handling data transformation is done in `ViewModel` by exposing the `FullNameOfSelected` value property. The `SampleModel` class in this case has two fields—name and last name—and our exposed property just aggregates these values in the getter property. A curious thing that should be remembered is calling `OnPropertyChanged` with the name of the string property in the `selectedItem` setter. Why we are doing this? Imagine when we set`selectedItem` in the code, the view gets updated because of calling `OnPropertyChanged` but another property that is related to `selectedItem` also needs to send notification about its state change. As you can see, we can call as many notifications as we need.

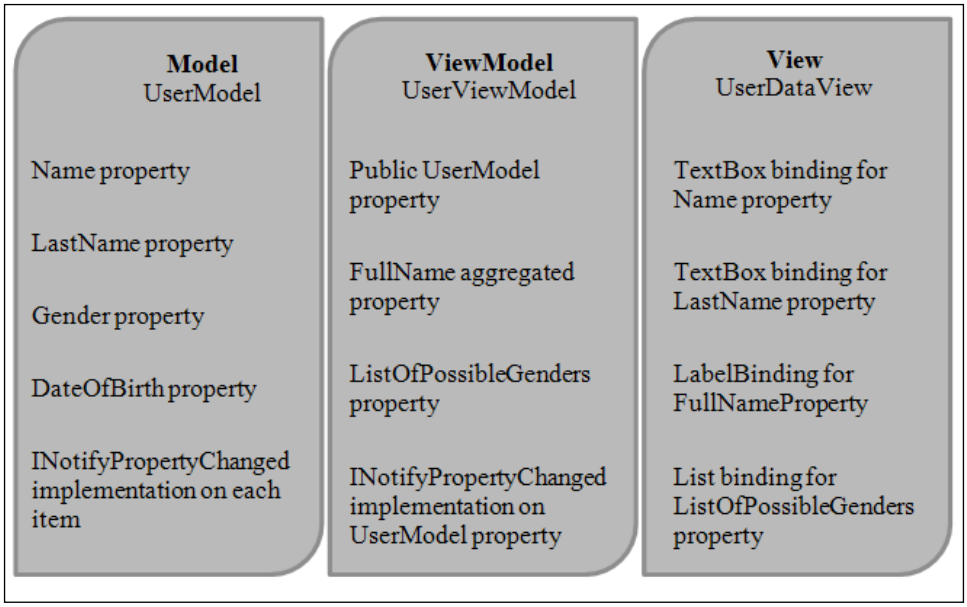
Typically, one `ViewModel` is for one `View`, and this is the most common situation but not the only one allowed. A curious example could contain one view for adding and editing item data and many `ViewModel` objects that handle functionality in views depending on the context. Contrariwise, many views could be bound to one `ViewModel` (for example, details view and a view of listed data that points to one VM).

Bindings

Basically, binding is a mechanism that the whole MVVM pattern relies on. It is very powerful and is the "glue" between the view and the exposed fields and commands.

In *Chapter 1, XAML in Windows Phone*, we learned about a mechanism called binding and saw what binding expressions are. The current chapter, being more practical, will show how to implement such bindings.

The functionality that we want to implement is shown in the following diagram:



Model

We are going to use the `Model` class for storing data about users.

```
public class UserModel:INotifyPropertyChanged
{
```

```
    private string name {get;set;
    public string Name { get;
      set
      {
        name = value;
        RaisePropertyChanged();
      }
    }
    public string LastName { get; set{(…)} }
    public string Gender { get; set{(…)} }
    public DateTime DateOfBirth { get; set{(…)} }

    public event PropertyChangedEventHandler PropertyChanged;
    (…)
}
```

One thing that we still have to implement is the property changed event subscription for all fields within this `UserModel` class.

ViewModel

Now, `UserModel` will be wrapped into `UserViewModel`. Our `ViewModel` will also implement the `INotifyPropertyChanged` interface for updating `View` when the `VieModels` object changes.

```
public class UserViewModel:INotifyPropertyChanged
{
  public UserModel CurrentUser { get; set; }

  public string FullName
  {
    get
    {
      if (this.CurrentUser != null)
      {
        return string.Format("{0} {1}",
          this.CurrentUser.Name, this.CurrentUser.LastName);
      }
      return "";
    }
  }
  public List<string> ListOfPossibleGenders
     = new List<string>(){"Male","Female"};
   (…)
}
```

As we can see, `UserModel` is wrapped in two ways: the first is wrapping the entire `Model` and some properties are transformed into `FullName`. The `FullName` property contains `Name` and `LastName` that come from the `UserModel` object.

View

I'm going to create a view, piece by piece, to show how things should be done. At the beginning, we should create a new `UserControl` object called `UserView` in the `Views` folder. We will do almost everything now in XAML; so, we have to open the `UserView.xaml` file and add a few things.

In the root node, we have to add a namespace for our `ViewModel` folder.

```
xmlns:models="clr-namespace:MyMVVMApplication.ViewModel"
```

Because of this line, our `ViewModels` will be accessible in the XAML code.

```
<UserControl.DataContext>
    <models:UserViewModel/>
</UserControl.DataContext>
```

It sets `DataContext` of our view in XAML and has its equivalent in `DataContext`. If we wish to set `DataContext` in the code behind, we have to go to the constructor in the `.cs` file.

```
public UserView()
{
  InitializeComponent();
  var viewModel = new SampleViewModel();
  this.DataContext = viewModel;
}
```

A better approach is to define the `ViewModel` instance in `XAML` because we have intellisense support in binding expressions for fields that are public in `ViewModel` and the exposed model.

As we can see, `ViewModel` contains the `CurrentUser` property that will store and expose the user data to the view. It has to be public; the thing that is missing here is the change notification in the `CurrentUser` setter. However, we already know how to do that so it is not described.

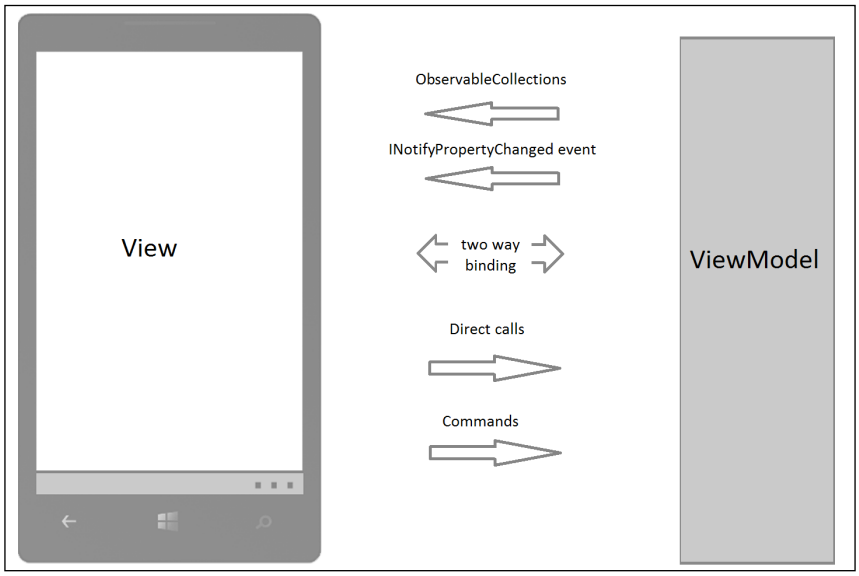
```
<StackPanel>
  <TextBlock Text="Name"></TextBlock>
  <TextBox Name="txtName" Text="{
    Binding CurrentUser.Name, Mode=TwoWay }">
  </TextBox>
```

```
    <TextBlock Text="Lastname"></TextBlock>
    <TextBox Name="txtLastName" Text="{
      Binding CurrentUser.LastName , Mode=TwoWay}">
    </TextBox>
    <TextBlock Text="Gender"></TextBlock>
    <ListBox Name="lstGender" ItemsSource="{
      Binding ListOfPossibleGenders}"
      SelectedItem="{Binding CurrentUser.Gender, Mode=TwoWay}">
    </ListBox>
  </StackPanel>
```

The preceding example shows how to set binding using the exposed model as well as the list or property that was implemented directly in `ViewModel`. We made all these things without any code behind the line! This is really good because our sample is testable and is very easy to write unit tests. We use the `TwoWay` binding mode that automatically populates the control value and then the edit control value of the `ViewModel` property also gets updated.

MVVM communication

Looking at the examples that were shown before, we can wonder about how we can execute some action in our application. For example, the **Save** or **Confirm** button should execute some action such as saving to the local database or making a service call. The traditional way suggests to set the `Click` (or another event such as `SelectionChange` and `KeyDown`) property on control and handle it in code behind. It works well in a legacy application, but it should not exist in `MVVM`.



As we can see in the preceding diagram, there are several ways to enable communication between the view and `ViewModel` and this is not only calling the piece of code when some event occurred but also communicating by changing property values or changing a list of items. Communication can also be understood as a change of state or exposing the model.

The following examples will show how we can create communication in view and `ViewModel`.

Wrapping model/property changes

The following example shows how to wrap the model and update the UI whenever the `Model` object in our `ViewModel` changes:

```
public class SampleViewModel:INotifyPropertyChanged
{
  public bool IsLoading
  {
    get { return isLoading; }
    set
    {
      if (value.Equals(isLoading)) return;
      isLoading = value;
      OnPropertyChanged();
    }
  }

  private SampleModel selectedItem;
  public SampleModel SelectedItem
  {
    get { return selectedItem; }
    set
    {
      if (Equals(value, selectedItem)) return;
      selectedItem = value;
      OnPropertyChanged();
    }
  }
  public ObservableCollection<SampleModel> Models;
  private bool isLoading;

  public event PropertyChangedEventHandler PropertyChanged;
```

```
    [NotifyPropertyChangedInvocator]
    (…)
}
```

This `ViewModel` is designed to wrap a list of `SampleObjects`, and it also exposes the `selectedItem` object. It implements the `INotifyPropertyChange` interface for updating UI when its properties change. The properties that we want to track are:

- `IsLoading` : This could be used for storing information while loading of data (for example, from web service) is in progress
- `selectedItem`: This will be used for keeping the selected item data for the details view, for instance

As we can see, our properties call the `OnPropertyChanged` method in their setters. You can ask, why we set the properties in a way that each change should be notified? Because our sample view will contain a list of selectable items; if we set the `selectedItem` property in code, it will also be selected on the view side. The `IsLoading` property may be bound to a view, which would display an indicator while data loading is in progress, depending on the `IsLoading` value.

The next item that exists within our `ViewModel` is `ObservableCollection`, a collection of `SampleModel` objects. As you can see, not only can simple objects be stored in such updatable collections but also complex ones, and this is one of the elements that could be bound from `ViewModel` to `View`.

This type of communication works in two ways; the following code shows how to implement and handle this `ViewModel` in a `View`.

```
<ListBox
  ItemsSource="{Binding Models}"
   SelectedItem="{Binding selectedItem, Mode=TwoWay}">
</ListBox>
```

No doubt that this part of the view uses VM but how is it communicating with VM? When users pick items from `ListBox`, then the `selectedItem` property is automatically set to this item. That's how the `TwoWay` mode works!

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Exposing commands

`Command` is another object within `ViewModel`. The `Command` objects implement the `ICommand` interface and are associated with the function that performs some action. On the view, the side command should be bound, for instance, to the button.

```
public class SampleCommandViewModel
{
  public ICommand SampleCommand { get; set; }
  public SampleCommandViewModel()
  {
    SampleCommand = new Command(ExecuteCommand,CanExecuteCommand);
    }
  private void ExecuteCommand(object parameter)
  {
    //do something here
  }
  private bool CanExecuteCommand(object parameter)
  {
    return true;
  }
}
```

The `ICommand` interface defines two methods:

- `CanExecute`: This method indicates if the `Execute` method can be performed
- `Execute`: This method invokes a delegate passed in the Command constructor.

Why is it done by using the interface? This is because of unit testing. If we define the `Command` object that inherits from `ICommand`, we can put whatever we want in this object and place it within our unit tests. If we don't feel like doing implementation of these interfaces, we don't have to do that. The Command functionality is already implemented in MVVM Light Toolkit, which will be described later.

Of course, the command has to be activated in some way. `Button` is a great example of using commands as shown in the following code:

```
<Button
  Command="{Binding SampleCommand}"
  Content="Click me!">
</Button>
```

As we can see, binding is a great mechanism, and we can use it in multiple ways. Using `ViewModel` that exposes the `Command` object, we are able to bind it to the view. Additionally, we could add the `CommandParameter` attribute to the button control and point which element or property we want to pass to the command.

Direct method calls

As we know, `ViewModel` should be completely separated from `View` but a situation can occur when we want to show `MessageBox` (that may be defined in code) .The `MessageBox` object is completely allowed and is even required when the application has to draw the user's attention.

The first thing we should create is the interface that will contain the `ShowMessage` method. You may say—interface? Really? Why? The answer is simple—tests that we are going to write don't know how to confirm reading messages, and we have to mock such functionality.

```
public interface IMessage
{
  void ShowMessageBox(object messageContent);
}
```

A simple interface that will be wrapped in `ViewModel` is as follows:

```
public IMessage Message { get; set; }
```

As long as it is public, we are able to refer to it in the code behind. Remember we were talking about reducing code behind as much as possible ? There is one such situation when some code behind is needed.

```
public partial class SampleView : UserControl, IMessage
{
  public SampleView()
  {
    InitializeComponent();
    ((SampleViewModel) DataContext).Message = this;
  }

  public void ShowMessageBox(object messageContent)
  {
    MessageBox.Show(messageContent.ToString());
  }
}
```

Here we go with the implementation of the `IMessage` interface where we show the message box. Now we can see we cannot do that in our `ViewModel` because it sets elements that belong to view.

Data templates

As the name suggests, data templating is creating a presentation of the data passed to `View`. Many Windows Phone 8 controls are enabled to use such data presentations. One can wonder why we should use data templates, but doing so gives us great flexibility in manipulating `View`. Out of experience, we will notice that data templates are mainly used in list controls such as `ListBox` and to create our own control. If our `ViewModel` contains only simple types such as `string` or `int`, there is no need to define the data template in list control; setting the `DisplayMemberPath` property will be enough. However, if the bound list contains some complex objects, such as model instances, we have to define how data should be presented.

Data templates could be defined as any ordinary control. One thing is very important—the `DataTemplate XAML` tag can contain only one child. So, if we want to put more than one control in `DataTemplate`, we should wrap those controls in the container.

A real-life example is as follows:

```
public class UserModel
{
  public string Name { get; set; }
  public string LastName { get; set; }
  public DateTime DateOfBirth { get; set; }
}
  public class FriendsViewModel
  {
    public List<UserModel> MyFriendsList { get; set; }
  }
```

The first thing to create is `ViewModel` that exposes the list of complex objects. Then, we are going to define the representation of these data in a view.

```
<UserControl.DataContext>
  <local:FriendsViewModel/>
</UserControl.DataContext>
  <ListBox Name="LstFriends" ItemsSource="{
    Binding MyFriendsList}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
      <TextBlock Text="{Binding Name}"></TextBlock>
      <TextBlock Text="{Binding LastName}"></TextBlock>
```

```
        <TextBlock Text="{Binding DateOfBirth}"></TextBlock>
        </StackPanel>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
```

This example shows `ListBox` that is bound to the `MyFriendsList` property of `ViewModel`. As long as bound items are complex objects, we need to create `DataTemplate`. List controls usually contain a node called `ItemTemplate`, and it is used to wrap `DataTemplate`. In turn, `DataTemplate` contains controls that are directly responsible for displaying data. As we can see, our data is presented by `TextBlocks` bound to each property of the complex Model object wrapped in the `ViewModel` list.

It is more common when we define the data template in Windows or control resources. There is a big advantage here; we can re-use such templates in many places. The first thing we should try to avoid is"inheritance by clipboard", that is, copy-paste practices. That is why wrapping data templates in resources is a good idea.

```
<UserControl.Resources>
    <DataTemplate x:Key="FriendProfileTemplate">
        (…)
    </DataTemplate>
</UserControl.Resources>
```

And we simply use the number of times we want to:

```
<ListBox Name="LstFriends"
         ItemsSource="{Binding MyFriendsList}"
         ItemTemplate="{StaticResource FriendProfileTemplate}">
</ListBox>
```

Item Template now is bound to a static resource (every user control or Windows resource is static). The code becomes more readable and such defined templates can be used many times.

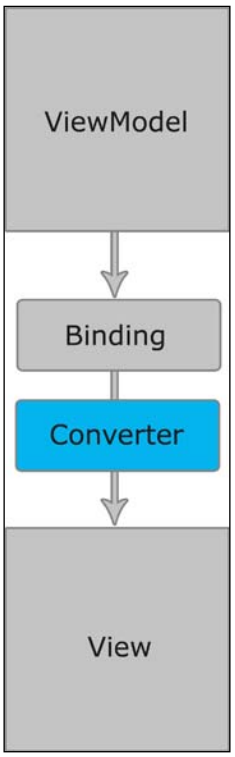
Data templating is a powerful mechanism, which enables us to control how data is presented, and it is completely a must-have in XAML programming as well as value converters.

Value converters

The preceding examples have the `IsLoading` property. At the moment, XAML controls don't support the Boolean state of visibility and have to be casted to one of the following values:

- `Visibility.Collapsed`
- `Visibility.Visible`

As we already know, the MVVM pattern advises against setting View-related values in `ViewModel`, but this is not a problem as long as we have Value Converters. Value Converter is a piece of code that converts the source (`ViewModel`) value to the target type value while binding. Conversely, the bound control property changes, goes to value converter and sets the `ViewModel` property.



At the beginning, we have to implement `IValueConverter` to our converter class. The interface implements two methods:

- `Convert`
- `ConvertBack`

As the names suggest, these methods are used to handle conversions in both ways.

```
public class BoolToVisibilityConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
  object parameter, CultureInfo culture)
  {
    bool result;
    if (value != null)
    {
      Boolean.TryParse(value.ToString(), out result);
    }
    return result ? Visibility.Visible : Visibility.Collapsed;
  }

  public object ConvertBack(object value, Type targetType,
  object parameter, CultureInfo culture)
  {
    return null;
  }
}
```

This example of the converter shows that converter methods take few arguments. The most important argument from our point of view is the `value` parameter. `Value` is the bound property value that has to be converted or converted back. Sample implementation returns the value of different types and then is passed to converter. We give Boolean and the code returns `System.Visibility`. It is allowed because the return type is set to `object`.

View implementation is not rocket science and only a few steps are enough to get rid of converters.

```
<UserControl.Resources>
  <converters:BoolToVisibilityConverter x:Key="BoolToVisibility"/>
</UserControl.Resources>
```

Converter instance can be defined in view resources. A good practice is to use one naming convention such as the `converters` namespace in all value converters. One important thing is to set the key for the static resource to be able to refer to it in controls.

```
<Image
    Name="ImgIndicator"
    Visibility="{Binding IsLoading,
    Converter={StaticResource BoolToVisibility}}">
</Image>
```

The `Visibility` property exists in most of the controls, so Boolean to visibility conversion is one of the most common ones.

This was the basic usage of converter, but its use is not limited to converting simple values such as strings and also includes objects whose complexity is very high. Even the whole model object can be converted, which doesn't look too good but is possible.

The value converter is a great mechanism but we have to remember that it consumes processing power. We can experience performance issues if we have a lot of value converters.

The MVVM Light Toolkit

MVVM Light Toolkit is known as one of the best (if not the best) light toolkits for the MVVM pattern. It significantly improves working with `MVVM` but still gives the user the freedom to customize and design the project. Toolkit supports many project types such as `WinRT`, `WPF`, and of course `Siverlight`.

Getting the MVVM Light Toolkit

MVVM Light Toolkit is available in the nugget package manager in Visual Studio or using the following nugget command:

```
PM> Install-Package MvvmLight
```

If you don't like automation and would like to manually add all references and required tools, you should go to http://mvvmlight.codeplex.com/ and download the package with all the stuff. After downloading and unzipping the package, we can find several extras as listed here:

- **Project templates for Visual Studio**: This includes templates that allow creating MVVM Light-enabled projects from Visual Studio's **New Project** window
- **Item templates for Visual Studio**: This includes templates for **New Item** in the project solution explorer
- **Binaries**: This includes DLLs that can be referenced by our application
- **Code snippets**: These are very useful helpers that accelerate our application development

The biggest disadvantage in MVVM pattern, in my point of view, is the lack of the `Command` class that will handle bound actions. When developers decide to not use any framework, they are forced to implement some mechanisms on their own. In turn, MVVM Light has this huge advantage as it provides the `RelayCommand` class out of the box. This class is responsible for representing commands in our `ViewModel`, and it implements and handles the `ICommand` interface features. Why is it so good? Because we can create any command we want in a very convenient way without going under a glass and thinking of very deep implementation.

The `RelayCommand` class expects to pass through its `constructor` object of the `Action` class (that defines a delegate to the method) and a predicate that will tell if a command can execute. There is also a generic implementation of this class that allows passing arguments to a method. This argument could be simple such as `int` or `string`, but can also be a complex type; this enables us to pass even the `Model` class through `Commands`. The following example shows how simple and fast is the implementation of `Commands` using MVVM Light Toolkit.

```
public class CommandViewModel:ViewModelBase
{
    public const int BaseNumber = 5;
    public int Sum { get; set; }
    private RelayCommand<int> addNumberCommand { get; set; }

    public RelayCommand<int> AddNumberCommand
    {
        get
        {
            if (addNumberCommand == null)
            {
                addNumberCommand = new RelayCommand<int>(x =>
                    {
                        this.Sum = BaseNumber + x;
                        RaisePropertyChanged("Sum");
                    });
            }
            return addNumberCommand;
        }
    }
}
```

In this example, we can see that our `CommandViewModel` inherits from `ViewModelBase`. It is the next feature of the MVVM toolkit that provides the `INotifyPropertyChanged` implementation, among other things. This way we can use the `RaisePropertyChange` method without actual implementation of it. Looking at this example, we can see that two fields of the `RelayCommand` object are created—one private and one public. We don't want to mess around in our `ViewModel` constructor and initialize the `Command` object; so, we prefer to initialize the `RelayCommand` private object in the public object's getter. The interesting thing can be creating an instance of `RelayCommand` because we use the lambda expression to create anonymous methods that will handle our command with the parameter.

```
        <StackPanel>
            <TextBox Name="txtNumber"></TextBox>
            <Button Command="{Binding AddNumberCommand}"
                    CommandParameter="{Binding Text,
                                       Converter={StaticResource
    StringToIntConverter},
                                       ElementName=txtNumber}"
                    Content="Add 5" x:Name="btnCount">
            </Button>
            <TextBlock Text="{Binding Sum}"></TextBlock>
        </StackPanel>
```

This XAML piece of `View` shows how to use our command and bind parameters to it. Additionally, we show converting data from `string` to `int` in runtime. Bindings in this view refer to `AddNumberCommand` and to the `Sum` property that is computed while the `btnCount` button is clicked (command-fired). Workflow is designed to update the `Sum` property when the button is clicked and a new value of the sum is calculated.

As we have such solutions, we can improve them. For example, we don't want to use any button, just enter a number and, when the content of the textbox changes, the value should be calculated automatically. We are able to do it using triggers under our control node in XAML. We can do this quickly because of MVVM Light .

```
        <StackPanel>
            <TextBox Name="txtNumber">
                <Interactivity:Interaction.Triggers>
                    <Interactivity:EventTrigger EventName="TextChanged" >
                        <Command:EventToCommand Command="{Binding
    AddNumberCommand}"
    PassEventArgsToCommand="True"
                         CommandParameter="{Binding Text,
    Converter={StaticResource StringToIntConverter},
    ElementName=txtNumber}"
                                                 />
```

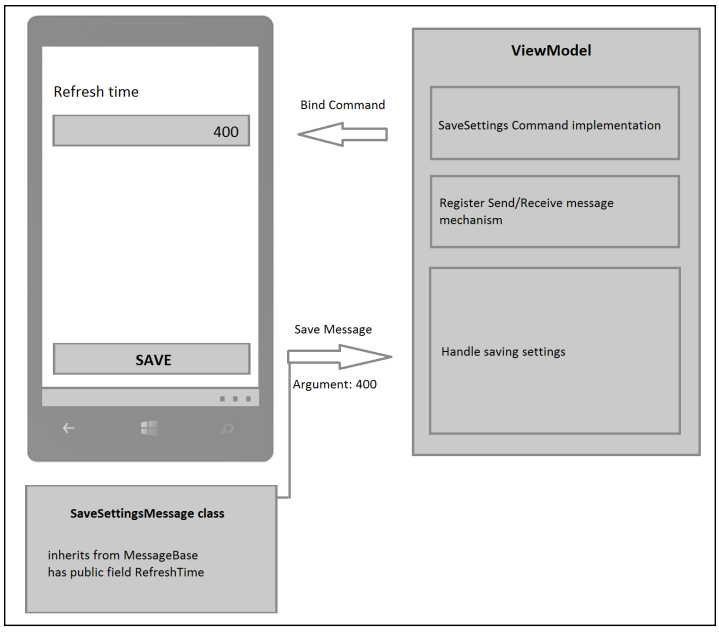
```
            </Interactivity:EventTrigger>
        </Interactivity:Interaction.Triggers>
    </TextBox>
    <TextBlock Text="{Binding Sum}"></TextBlock>
</StackPanel>
```

We don't have the button that fires a command. Now, user action causes the execution of the event handler even more importantly, with no code behind!) It shows how powerful MVVM is since we changed `View` without changing a single line of our `ViewModel` class.

Messaging

The `Messenger` class is a part of MVVM Light Toolkit and was intended to be a provider of communication not only between ViewModels and also any class but could be used as communication channel by any objects. Messenger is a static class that is not a part of MVVM but is one of the helpers in the toolkit. The class provides mechanisms for receiving and sending communication messages that could be simple or complex depending on how we implement them. An interesting thing is, we can send a message that contains a callback function, which allows two-way communication.

We are going to create the application's simple settings page that will use messaging to save the settings data. Though the sample will look trivial, generally the mechanism used here can be used in a more advanced way with minor changes . The following diagram depicts this plan:



As we can see in the preceding diagram, we are going to use the `MessageBase` class to create our own message that will carry items from `View` to `ViewModel`. After clicking on the **Save** button, the bound command will send a message to `ViewModel`, which has a registered receiving mechanism, and then will handle saving data.

The implementation of `SaveSettingsMessage` looks as follows:

```
public class SaveSettingsMessage:MessageBase
{
    public int RefreshTime { get; set; }
    public SaveSettingsMessage(int refreshTime)
    {
        this.RefreshTime = refreshTime;
    }
}
```

As we can see, it looks like an ordinary model; the only difference is that the class inherits from `MessageBase`, which makes it passable in a message-taking argument. The more sophisticated implementation has our `ViewModel`.

The first thing is to create the `SettingsViewModel` class that inherits from `ViewModelBase`. The non-parameterized constructor of our class registers the receiving mechanism and points the handler to take messages of a specified type. The keyword `this` means that the current `ViewModel` will take care (will contain the method) of handling the message as follows:

```
        #region ViewModel Contructor
        public SettingsViewModel()
        {
            Messenger.Default.Register<SaveSettingsMessage>(this,
HandleSaveSettings);
        }
        #endregion
```

The initialization of `RelayCommand` will be bound to the **Save** button. As we can see, it is very similar to the example of `RelayCommand` that was shown before. One difference is that this command sends the message using Messenger with the passing argument of our `Message` class.

```
        #region Save settings command
        private RelayCommand<int> saveSettingsCommand;

        public RelayCommand<int> SaveSettings
        {
            get
            {
```

```
                if (saveSettingsCommand == null)
                    saveSettingsCommand = new RelayCommand<int>(
                     x =>
                     {
                         Messenger.Default.
Send<SaveSettingsMessage>(new SaveSettingsMessage(x));
                     });

                return saveSettingsCommand;
            }
        }
        #endregion
```

The `Save` method can call some business logic or service interface to perform saving the data. An argument that is passed from the Messenger gets handled in this method.

```
        #region Handle Saving

        public void HandleSaveSettings(SaveSettingsMessage settings)
        {
            this.RefreshTime = settings.RefreshTime;
            //do local storage call to save
        }

        #endregion
```

Messaging is the next feature that shows separation from `View`. In this example,the `View` XAML code was not listed because it looks exactly the same as others with the `RelayCommand` usage and this is another proof of the power of separation; we changed logic with no changes in the application's appearance.

Page navigation with MVVM

One of the most important things in MVVM application development is navigation between pages. We are going to create it using the Messenger helper. Navigation should be as clear as it can be and should follow the MVVM pattern with no code behind!

The first thing that we will do to improve simplicity and accelerate our development is to create a base class for each page in our application. Let me introduce you to `ApplicationPageBase`, which will inherit from `PhoneApplicationPage`. This will mean, we will extend the basic functionality of this class. The most important method of our new class is `NavigateTo`.

```
public void NavigateTo(Uri uri)
{
    if (uri.ToString() == "/GoBack.xaml")
        NavigationService.GoBack();
    else
        NavigationService.Navigate(uri);
}
```

This method handles `NavigationService` and `GoBack` behavior—we all know how important it is in mobile applications.

The next two methods in our base class register and unregister the `Messenger` mechanism for incoming navigation requests.

```
        protected override void OnNavigatedFrom(System.Windows.
Navigation.NavigationEventArgs e)
        {
            Messenger.Default.Unregister<Uri>(this);
            base.OnNavigatedFrom(e);
        }

        protected override void OnNavigatedTo(System.Windows.
Navigation.NavigationEventArgs e)
        {
            Messenger.Default.Register<Uri>(this, "NavigationRequest",
NavigateTo);
            base.OnNavigatedTo(e);
        }
```

This is only a few lines of code but they help a lot during development especially when our applications contain a lot of pages. Now we should only change the `MainPage` class to inherit from our new base class, but it is worth remembering that the `MainPage` class is a partial class, and we also have to change the XAML code of inheritance. It means we have to change the root node in the XAML file. In this example, it will be:

```
<CustomNavigation:ApplicationPageBase
…

</CustomNavigation:ApplicationPageBase>
```

What should we do next? Our application still doesn't know how to handle navigation. So, we have to create a new command in `MainViewModel`, which will be bound to button or `HubTile` and will send messages by Messenger with `NavigationRequest`.

```
        public RelayCommand GotoSettingsPage { get; set; }
        public MainViewModel()
        {
            GotoSettingsPage = new RelayCommand(gotoSettingsPage);
        }
        private void gotoSettingsPage()
        {
            Messenger.Default.Send(new Uri("/Views/SettingsPage.xaml",
UriKind.Relative), "NavigationRequest");
        }
```

This example shows handling of the `GotoSettingsPage` command. This command is bound to the `Hubtile` settings in a view. When the tile is tapped, the messenger sends messages with a navigation request; because our base class has registered the receiving mechanism, it simply proceeds with such requests and forces `NavigationService` (build in service) to navigate to the page with specified URI path.

ViewModel locator

Another thing that will provide improvement to our application will be the `ViewModel` locator. If we create such locators and somebody asks us if we know any patterns, we can say—yes! I know the `ServiceLocator` pattern! This class will contain all instances of `ViewModels` in our application. Why is it a good idea to keep instances? Imagine a situation when our application, while loading some page, needs to load and populate lots of data into `ViewModel`. We don't want to load it again and again while going to this page; it is enough to do it once.

Our `ViewModeLocator` class looks as follows:

```
    public class ViewModelLocator
    {
        public ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.
Default);
            SimpleIoc.Default.Register<MainViewModel>();
            SimpleIoc.Default.Register<SettingsViewModel>();
        }

        public MainViewModel MainViewModel
```

```
            {
                get
                {
                    return ServiceLocator.Current.
    GetInstance<MainViewModel>();
                }
            }

            public SettingsViewModel SettingsViewModel
            {
                get
                {
                    return ServiceLocator.Current.GetInstance<SettingsVie
    wModel>();
                }
            }
            public static void Cleanup()
            {
                // TODO Clear the ViewModels
            }
        }
```

The constructor initializes `ServiceLocator`—it is a kind of a bag for our `ViewModels`. This service will keep instances, and we can simply get a `ViewModel` instance by using the generic method called `GetInstance`. Our application for now contains only two `ViewModels`, so only those two are included in `ViewModelLocator`.

To use locator, we need to edit the `App.xaml` file and add the following line:

```
<vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" />
```

From now on, we are able to use `ViewModelLocator` instance in our `Views` as a static resource object. The `Locator` instance holds all `ViewModel` instances, so setting `DataContext` in our `Views` never will be simpler. It is enough to add the following line as the root node property.

```
DataContext="{Binding MainViewModel, Source={StaticResource Locator}}"
```

It means that the application will look for the `MainViewModel` property in the `Locator` resource. `Locator` is the name of `ViewModelLocator` set in the `App.xaml` file.

I hope you enjoyed the short description of MVVM Toolkit, which is very rich in its simplicity, improves understanding of MVVM , and accelerates developers' work! Because of many features such as messages or `ViewModelLocator`, we can improve user experience because of holding an instance of the loaded `ViewModels` and as a result making the application run fast.

Unit testing

A unit test can be defined as piece of code that executes another piece of code and compares the returning value with the expected value. If it is equal, the test is passed; if not, it fails. Unit tests are not a fad as some people think. Possibly, at the beginning of development, it is more costly to cover the application with unit tests. However, when it progresses and functionality and the number of dependencies increase, it is very helpful to keep things clean and properly working.

Unit tests force developers to use a style of coding that enables them to cover their code with tests, only with dependencies that are testable. As the name suggests, unit test is the code that tests atomic functionality. Such atoms collectively enable the application to be fully functional.

How to use unit test in a Windows phone?

Microsoft has created the Windows Phone Toolkit test framework that is dependent on Windows Phone Toolkit. Test Framework can be installed by executing the following nugget command:

```
PM> Install-Package WPToolkitTestFx
```

And now we can start creating tests for our application.

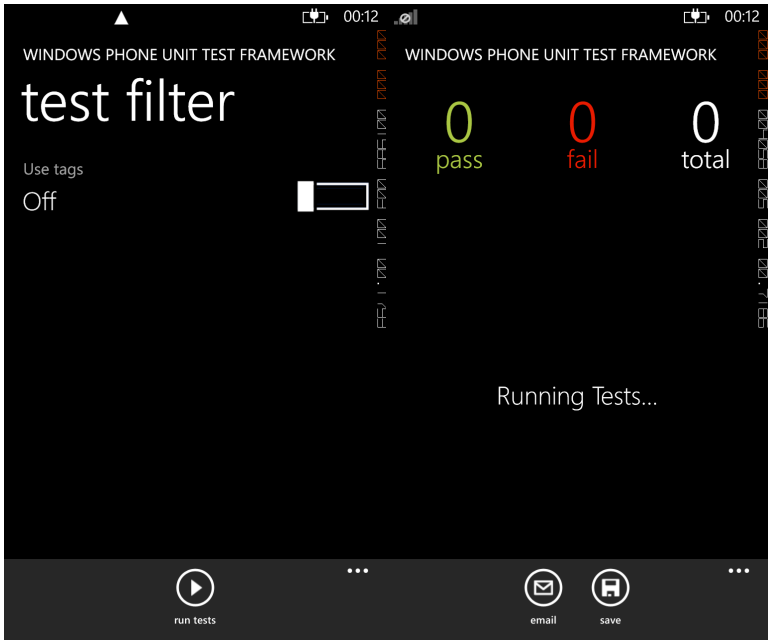
The very first thing to do is to create the page within our application that will contain all test results. For testing purposes, we can edit the `MainPage` (the default application page) code-behind file's ad force page to display information about unit tests.

```
public MainPage()
{
    InitializeComponent();
    this.Content = UnitTestSystem.CreateTestPage();
}
```

This simple modification of the `MainPage` constructor comes into effect after the application launches, and we will see the test settings page where we can play with different settings for the test. After clicking on the **Play** button at the bottom of the page, the application navigates to test-run the screen. The test result can be in one of two states:

- **Pass**: This indicates the tested piece of code returns the same value as expected

- **Fail**: This indicates the tested piece of code throws an exception or returns an incorrect value



Still we have no tests in our project. We can put the class with tests anywhere we want to, but as we are thinking of keeping things well organized, we are going to create a folder called `Unit tests` and will keep our test classes there. We are going to test the functionality of one of our `ViewModels`. To do this, we will create a test class whose name contains the class under the test name as prefix and `Test` as a suffix. We already created the `Model` class `Message` that contains the string property `message`. Our `ViewModel` `MessagesViewModel` has a list of "Message" objects and the `FindMessagesThatContain` method with the argument string. This method takes the text argument and returns all messages that contain the passed text. Most of the test methods will contain the following parts:

- **Data preparation**: This part is where the test data are prepared
- **Test object creation**: This part is where the object under test is created and initialized
- **Assertion**: This part is where the results of operations are checked

Attributes on the test classes and test methods cause the application/environment to know how to treat such test methods in a special way.

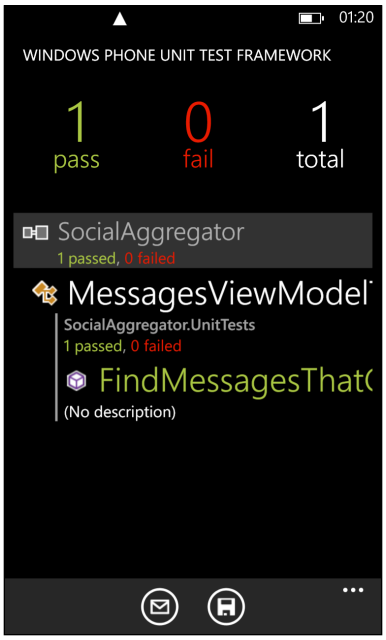
- `[TestClass]`: This attribute should be put on the class
- `[TestMethod]`: This attribute for the method within the test class

The following code shows a simple test class with one test method:

```
[TestClass]
public class MessagesViewModelTest
{
    [TestMethod]
    public void FindMessagesThatContain_3Items_Returns_2()
    {
        var messages = new List<Message>()
        {
            new Message() {message = "first message!"},
            new Message() {message = "second message!"},
            new Message() {message = "third sentence!"}
        };

        var viewModel = new MessagesViewModel();
        viewModel.Messages = new ObservableCollection<Message>(me
ssages);

        var result = viewModel.FindMessagesThatContain("message");

        Assert.IsNotNull(result);
        Assert.IsTrue(result.Count==2);
    }
}
```

We can see that our test method has a strange name; this is done on purpose. It is a good practice to call methods in a way that allows a developer to become acquainted with the details of method execution while looking at the name. This example executes the method and prepares a three-item list; that is why the name contains `"3Items"`, and the last part of the name is equal to the correct value that should be returned from the tested method. In this situation, there are two items. Looking at the three parts of the unit test built schema, we can find such places in our code when data are prepared (list of messages), a test object is created and initialized (that is, `ViewModel` creation), and results are checked by comparing the count of the return values.

After launching, Windows Phone Unit Test Framework shows the following results:



Test results contain a tree with the name of our project as a child node and names of classes under `test`; under test classes node is a list of test methods with information about passes or fails. Now we can see why naming is so important—in most cases when tests fail, after a quick look at the name of the failed test method, we are able to notice where and why we went wrong.

Another, and I think more common, way of placing test classes is a separate test project in our application. The naming convention of such a project is similar to the rest—new test projects should have such a name that the prefix is the name of the tested project and the suffix `Test`.

After the creation of the test project , we will reference the project that we want to test. The difference is that we can launch tests in Visual Studio outside an application. It is the fastest way and is used on a daily basis in most projects

Creating the application "SociAgg"

The examples included in this chapter are the basis of working for the SociAgg application. The application is intended to aggregate information from social portals, such as Facebook or Twitter, and will be under development in the next chapters. Extensive code was necessary in this chapter, along with very important information and "must have" things. I think we learned a lot about best practices in MVVM development and application development in general. The understanding of the MVVM pattern is just an implementation; there are other implementations and toolkits but I think the mechanisms and patterns shown here will be a very good solution in most cases and problems.

Summary

This chapter shows us what the MVVM pattern depends on and how to implement it in our application. For sure, there are many advantages of using MVVM and each of them were listed here. A huge part of XAML—data templates, bindings, and MVVM commands were described with many examples of real-life usage. There are some third-party libraries that can be used to improve performance and accelerate work, one of them being the MVVM toolkit described in this chapter. In the end, we had a brief description of what unit testing is and how to implement it in our application.

4
Integrating with Windows Phone

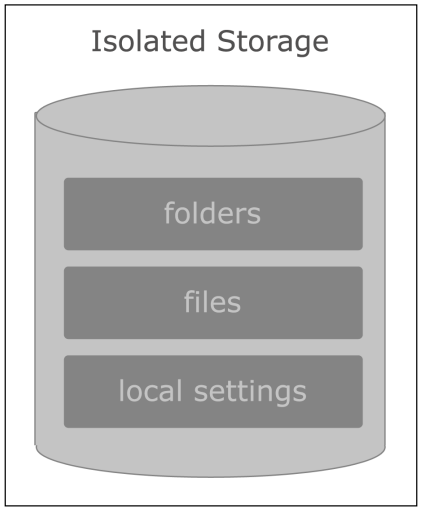
In this chapter, we focus our attention on the features delivered by Windows Phone SDK that help in integration with devices. We have described mechanisms that existed in previous versions of Windows Phone, such as the file API or some launchers and choosers, and also those released in the newest version of Windows Phone, such as the background agent or live tiles:

- Isolated storage
- Settings API
- File API
- Live tiles and notifications
- Background agent
- Toast notifications
- Launchers
- Choosers

Isolated storage

Isolated storage is a space designed for storing data that is used in an application. Why isolated? Because this piece of memory is available only within an application; other applications don't have access to the items stored in the isolated storage. It means that Windows Phone protects application data from unauthorized access; more importantly, such protection is done automatically. Things that can be written into isolated storage depend only on implementation and the author's ideas and can include:

- Temporary files, often used as a kind of helper or for saving the state of an application. It worth remembering that such files should be cleaned when they are no longer in use.
- Images or other media files.
- Files that are used as storage for data in applications such as serialized XML files or text files.
- Application local settings.



The preceding image shows how isolated storage resembles a single application; each application has its own storage like this.

The Settings API

Windows Phone SDK provides the ability to save some content in application storage. Application settings could be understood as a **key-value pair** that is saved within isolated storage. Settings API provides a class called `IsolatedStorageSettings`, and this mechanism is the best option to store some small pieces of data such as settings or application state. `IsolatedStorageSettings` is a kind of dictionary (a list of `key` and `value` properties) that contains unique keys and its values. The stored value is an object type, so we can store whatever we want there; just ensure you cast the proper class when getting it from the storage settings.

There are three methods that are used most often while dealing with settings:

- `Contains`: This returns true or false according to the settings store contain key passed in the method parameter
- `Add`: This adds a key-value pair to the settings dictionary; it is important that we do not try to add two items with the same key because the key is a unique value
- `Remove`: This removes settings pair of the specified key

Additionally, `IsolatedStorageSettings` has a `Save` method; other dictionaries don't implement it.

We are going to start with creating methods that handle application settings.

```
public class StorageSettingsProvider:IStorageSettingsProvider
{
  (…)
}
```

The storage provider class that we wrote implements an interface that will include methods used to manage storage. Why do we need that? It is not required to work properly, but our application will have unit testing enabled and using the interface, we can simply mock the provider object in unit test. That way we will be able to test methods that would use our provider. We don't want to create dependencies with the system—such as storage IO—that is why we need to create and pass an interface to the final implementation of ViewModel.

Adding the `settings` method to the dictionary item is given as follows:

```
public void SaveSetting(string key, string value)
{
  IsolatedStorageSettings settings =
    IsolatedStorageSettings.ApplicationSettings;
```

```
    if (!settings.Contains(key))
    {
      settings.Add(key, value);
    }
    else
    {
      settings[key] = value;
    }
    settings.Save();
  }
```

This example shows how to add or update the `settings` item. We wrote a method that wraps `IsolatedStorageSettings` and will use it in our application. We can see that, after adding or updating the existing `settings` item, we should call the `Save` method on the `ApplicationSettings` instance. We are passing `string key` and `string value` to the `Save` method, but the value parameter could be of any type that we want. We can put our own custom class object as a value and save it.

Getting the `settings` value from the dictionary item is given as follows:

```
  public string ReadSetting(string key)
  {
    IsolatedStorageSettings settings =
      IsolatedStorageSettings.ApplicationSettings;
    if (settings.Contains(key) && (settings[key] as string)!=null)
    {
      return settings[key] as string;
    }
    return string.Empty;
  }
```

Reading the `settings` mechanism uses the same object as writing. This example shows that getting the `settings` key value works the same as with a traditional dictionary. While reading the value from `settings`, we check whether any element with the specified key exists in a collection and whether it is a string.

The File API

As we said before, isolated storage could contain files and folders. The new Windows Phone 8 SDK provides asynchronous methods to manage files in our storage.

- `RenameAsync`: This renames the selected folder
- `DeleteAsync`: This deletes the folder in which the method was called

- `CreateFolderAsync`: This creates a folder with a specified name and default behavior in case of conflicts
- `CreateFileAsync`: This creates a file in the folder with a name and default behavior in case of conflict
- `OpenStreamForReadAsync`: This opens the file stream to read
- `OpenStreamForWriteAsync`: This opens the file stream to write

The following examples will show how to read/write the simplest message—a string type variable.

Reading the file

The following example will present `ReadTextFile()` asynchronously:

```
public async Task<string> ReadTextFile()
{
  string result = string.Empty;
  StorageFolder storage = ApplicationData.Current.LocalFolder;

  if (storage != null)
  {
    var dataFolder = await storage.GetFolderAsync("FolderName");
    var file =
      await dataFolder.OpenStreamForReadAsync("SampleFile.txt");
    using (StreamReader streamReader = new StreamReader(file))
    {
      result = streamReader.ReadToEnd();
    }
  }
  return result;
}
```

The preceding example shows how to handle the reading of data from the file in a given folder (`FolderName`) that exists within isolated storage. The first thing here is that the storage object should be available for reading purposes; using `OpenStreamforReadAsync`, we get a stream that can be read. As we can see in this method, we use asynchronous context; it allows the application to still be responsive to user input even if it has something to do in the background. The `async` keyword makes the method work asynchronously; in the asynchronous context, we can use the `await` keyword to call another method synchronously.

Creating a folder and writing files

The following example is more sophisticated than one with file reading:

```
public async void WriteTextToFile(string content, string fileName)
{

  byte[] bytes = System.Text.Encoding.UTF8.GetBytes(content);
  StorageFolder storage = ApplicationData.Current.LocalFolder;
  var dataFolder = await storage.CreateFolderAsync
    ("FolderName", CreationCollisionOption.OpenIfExists);

  var file = await dataFolder.CreateFileAsync
    ("SampleFile.txt", CreationCollisionOption.ReplaceExisting);

  using (var stream = await file.OpenStreamForWriteAsync())
  {
    stream.Write(bytes, 0, bytes.Length);
  }
}
```

It shows the creation of the folder within our storage with the name and default behavior when a conflict occurs. For example, we can expect a folder with the same name to already exist; then, instead of writing, we would open that folder. In such a situation, using `CreationCollisionOption.OpenIfExists` is reasonable. The next thing here is creating a file within a folder instance and opening a stream for writing.

Reading and writing serializable classes

Previous examples show how to write simple values but we often face problems where we need to save complex objects. We need preparation of the model that we want to serialize and save in the file. Preparation means we need to:

- Mark the class with the `DataContract` attribute
- Mark the properties with the `DataMember` attribute
- Provide a getter and setter for each data member
- Provide a non-parameterized class constructor

After applying such modifications to the model, we are ready to write methods that will handle reading and writing of complex objects. To make it extremely useful, we will use generic asynchronous classes:

```
Task<T> ReadFile<T>() where T : class;
void WriteToFile<T>(T content, string fileName);
```

Here, `T` means that it could be our model complex type. Now we can modify the previous examples slightly and introduce the `DataContractSerializer` class, which will handle serialization and deserialization. Then, using streams we serialize our object with:

```
DataContractSerializer dcs =
  new DataContractSerializer(typeof(T));
dcs.WriteObject(stream, content);
```

We deserialize it with:

```
DataContractSerializer dcs =
  new DataContractSerializer(typeof (T));
result = dcs.ReadObject(file) as T;
```

As we can see, we are casting our model, Type (`T`), which indicates that we have to set a constraint in the generic method where `T` is the class.

Implementing tile notification

Live tiles were designed to improve user response to our application; users feel that our application thinks about them all the time because it updates its tile and shows some information. The clue is to make users feel integrated with the application even if it is not running.

Updating the application tile from code

As we know, we have several tile sizes and templates. Each tile size could present some information to the user. Depending on the tile template that we want to use, we can update the data. If our application is pinned to the Start screen, we can get the active tile for the application using the `ShellTile` API. Using the following line we get the default tile for the application:

```
var activeTile = ShellTile.ActiveTiles.FirstOrDefault();
```

This value is not null when the application is pinned to the Start screen and we have to check that we do not face the exception. The `activeTile` object has the `Update` method, which gets `TileData`. `TileData` will update our tile and can be any of the following types:

- `CycleTileData`: This describes a tile that background cycles between a collection of 1 to 9 images
- `FlipTileData`: This describes a tile that has two sides and flips between those two

- `IconicTileData`: This describes a tile that contains a static icon
- `StandardTileData`: This describes a two-sided standard tile

Each tile data object contains different information according to tile template requirements.

```
public static void UpdateFlipTile()
{
  var activeTile = ShellTile.ActiveTiles.FirstOrDefault();
  if (activeTile != null)
  {
    activeTile.Update(new FlipTileData()
    {
      Title = "New title text",
      BackTitle = "New back title",
      Count = 2
      //possible other fields setting
    });
  }
}
```

Updating a flip tile could look like the preceding example. `FlipTileData` initialization sets content that will be presented to the user such as `Title`, `Count`, and background images. After calling the `UpdateFlipTile` method, our tile will update, and if it's pinned, its content will change.

Background agents

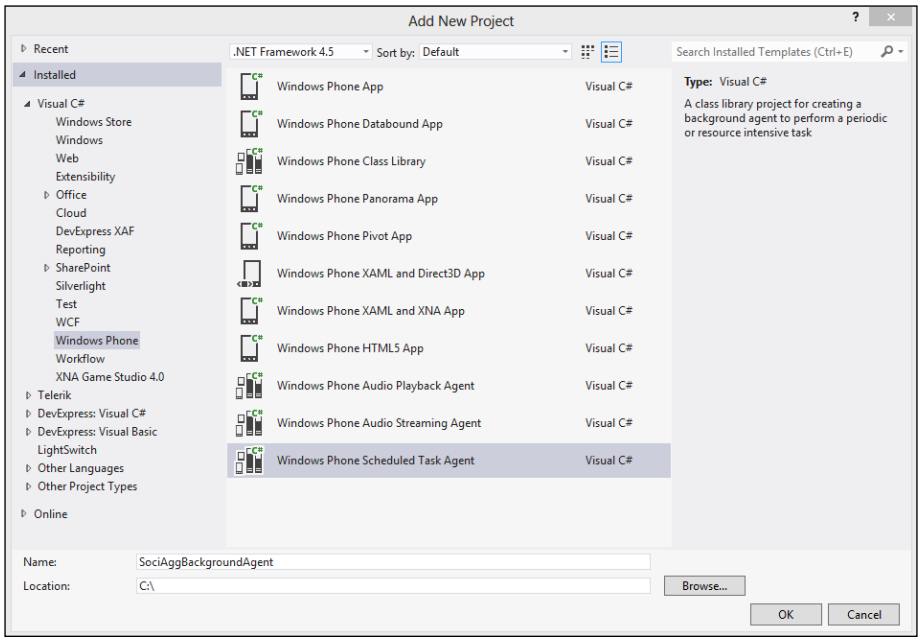
Modern applications are very interactive since they collect data and load content with no user attention. Windows Phone applications that use background agent are able to execute code even if they are not running in the foreground.

An application can implement only one background agent, which could be one of two types:

- `PeriodicTask`: This is used to specify tasks that run typically every 30 minutes and could be 25 seconds long. There is a periodic task limitation in the system for a maximum of six periodic agents per device. If our phone reaches this limit, adding a new task to the schedule causes `InvalidOperationException` to be thrown. It is usually used to get some updates, achieve synchronization, or calculate some data.

- **Resource-intensive agent**: Tasks will run only if the power source is plugged into the device and typically run for 10 minutes. There is also a limitation in the internet connection type; that is, it can only use a Wi-Fi or PC connection. It fires when the device is not actively used by the user and is mainly used to synchronize big amounts of data, upload pictures, and so on.

To create background agent, the first thing we have to do is add a new project to our project. Why do we need a separate assembly? Because the Windows Phone system will execute only this agent assembly.



The new project exists in the project template as a scheduled task agent. Creation of this project automatically generates the `ScheduledAgent` class and includes some methods to handle the functionality of background work. The agent contains the overridden `OnInvoke` method, which should be modified in the way we want our agent to work. Using the tiles notification example, we are going to call the method that sets tiles in our application.

```
protected override void OnInvoke(ScheduledTask task)
{
  LiveTiles.UpdateFlipTile();
  NotifyComplete();
}
```

When the work of an agent is completed, we should call `NotifyComplete` to make the device know that we have finished with our background task. An `OnInvoke` method is executed when the background agent launches `PeriodicTask`.

Our background agent is ready but we still have to make it work. To do that, we have to add a reference background agent assembly to our main project. Then we need to activate and schedule our task behavior, set when it should start, or provide a description.

```
public class BackgroundTaskManager
{
  private string periodicTaskName = "sociaggUpdate";
  public void RunPeriodicTask()
  {
    if (ScheduledActionService.Find(periodicTaskName) == null)
    {
      var backgroundAgentTask = new PeriodicTask(periodicTaskName)
      {
        Description = "Description shown in device settings"
      };
      ScheduledActionService.Add(backgroundAgentTask);
    }
  }
}
```

An instance of the `BackgroundTaskManager` class will be created when the application starts and the `RunPeriodicTask` method is executed. As we can see, there are some options that help us to schedule the background agent using `ScheduleActionService`. What is important is that the task name has to be unique and the `Description` property is the text that we will see in the settings area while displaying the background tasks.

For testing purposes, we can force our application to invoke the background agent more often by adding the following line during initialization:

```
ScheduledActionService.LaunchForTest
  (periodicTaskName,new TimeSpan(0,0,0,5));
```

We can see that the agent will be launched for test and will execute the `RunPeriodicTask` method every 5 seconds but, as we can see, it is used only for testing purposes while the debugger is attached. Tiles are not the only things that are often updated in background agents; another feature is toast notification. Important execution of agent tasks could be prevented by low battery level. The Windows Phone system decides when to execute background tasks and stops execution to save battery power.

Toast notifications

Toast notifications are often used to inform users about things that apply to our application. It could be information about messages, invitations, and so on. We are going to create a simple method that shows a sample message, and after clicking, it goes directly to one of the pages in the application.

```
public static void ShowSettingsToast()
{
  var toastMessage = new ShellToast();
  toastMessage.Title = "Hello";
  toastMessage.Content = "Your app settings needs your attention";
  toastMessage.NavigationUri =
    new Uri("/Views/SettingsPage.xaml", UriKind.Relative);
  toastMessage.Show();
}
```

Implementation is very simple; it's just an initialization of the `ShellToast` object that contains `Title`, `Content`, and `NavigationUri` and that points to our application page. Calling this method in background agent causes the notification to be seen when the application is not running.

Launchers

Launchers are a Windows Phone API feature that launch some native applications such as maps, e-mail, and phone call. Open applications (tasks) can be cancelled; then as they finish, the calling application is reactivated. There are many launchers implemented in SDK and we will describe a few of them:

- **E-mail composing**: Using this launcher enables the user to send e-mails when using our application. It opens the email application, creates a new message, and focuses on message entry. While creating this launcher, we are able to specify the message text, topic, and recipients.
- **SMS composing**: This launcher opens the messaging application and optionally sets the text and recipients, but SMS messages will be sent only if the user clicks on send.
- **Phone call**: This task opens the phone call application and sets number. The call will be made only if the user presses the call button.
- **Maps task**: This task launches with the maps application. We can specify whether it should center the map to the user's current location or different specified coordinates. If we want to search for a particular place, we can set search text during initialization of the launcher.

- **Review in marketplace**: It launches the Windows Phone Marketplace application focused on the rating page; it could be a good way to convince the user to rate our application.
- **Sharing link**: This launcher causes sharing of a hyperlink in the selected social media portal.
- **Web browser task**: This launcher opens the Internet link that was specified in its initialization properties.

These are only some of the launchers; we can find information about the rest of them in the SDK documentation. Every launcher can be opened in the same way; there are three steps:

1. Create the launcher instance.
2. Set the parameters.
3. Call the `Show()` method.

For instance:

```
MapsTask mapsTask = new MapsTask();
mapsTask.Center = new GeoCoordinate(50.341, 19.561);
mapsTask.SearchTerm = "cinema";
mapsTask.ZoomLevel = 3;
mapsTask.Show();
```

The created map launcher enables us to specify to which coordinates the map should be centered (if we omit setting this property, it will center on the user's current location), the search text, and the map zoom level. In the last step, the map launcher is started by calling the `Show()` method.

Choosers

Choosers' tasks work in a manner similar to launchers, but the main difference is that, when a new task is finished, it usually populates our application with some data. For example, camera or gallery image choosers. In this situation, we also have many choosers provided by the SDK:

- **Address chooser**: It launches the contact book application, which allows us to select the physical address of our contact.
- **E-mail address chooser**: It is similar to the address chooser. It opens users' contacts and allows selecting a contact with their e-mail address.

- **Camera chooser**: It launches the camera application, which enables the user to take photos that will be captured in our application.
- **Gallery photo chooser**: It gives users the ability to select existing photos from their photo gallery and send them back to our application.

Using a chooser is a bit more sophisticated because we need to handle the callback method that populates data from the chooser task to our application. The following are the steps to remember while creating a chooser:

1. Create the chooser instance.
2. Initialize the completed event handler.
3. Create a method that handles the task completed event.
4. Call the `Show()` method.

A sample chooser code of the camera chooser should look like this:

```
CameraCaptureTask cameraChooserTask;
public void InitializeCameraChooser()
{
  cameraChooserTask = new CameraCaptureTask();
  cameraChooserTask.Completed += new
    EventHandler<PhotoResult>(cameraChooserTask_Completed);
  cameraChooserTask.Show();
}

private void cameraChooserTask_Completed(object sender,
  PhotoResult e)
{
  if (e.TaskResult == TaskResult.OK)
  {
    var photoStream = e.ChosenPhoto;
  }
}
```

Examples show how to create and handle a chooser instance. If the task is completed, the completed event will be raised; it causes `cameraChooserTask_Completed` to be called. If the result of the task was OK, the `ChosenPhoto` property will be available for use. Why do we need to check status? Because a user can change his mind and cancel the task, and then the completed method will be raised with the cancel status. Many other choosers work in a similar way, and all of them are available in the SDK documentation.

Summary

In this chapter we described how to use Windows Phone 8 features that make our application more integrated with the user's device. We learned how to use launchers and choosers, how to read and write settings, and how to manage files in the device storage.

The next chapter shows us how to integrate our application with social media portals.

5
Integrating with Twitter and Facebook

This chapter covers how to enable social portals. We will see how to register a Facebook application and use Graph API in our application. Also, Twitter integration will be described, which requires creating a special application within the portal. Majorly, we will cover the following:

- Facebook C# SDK
- TweetSharp

Facebook for developers

Social media gives us the power to share information about ourselves and be in touch with our friends and community. There are some built-in mechanisms such as "share" in WP8, but more complex information is available only by using a dedicated application.

If we want to create our application online with Facebook, we should first sign in to our Facebook account and create a developer account.

Go to `https://developers.facebook.com/`. Using this site, we are able to create a Facebook application. This site will be our agent in Facebook that will allow us to get data from Facebook. One of the fields to fill in will be the type of application we would like to create. There are multiple solutions to connect to Facebook, but the one that we will use is Graph API. When we finish creating our application, we will get our Facebook Application ID and the secret key—both will be needed when creating the Windows Phone application.

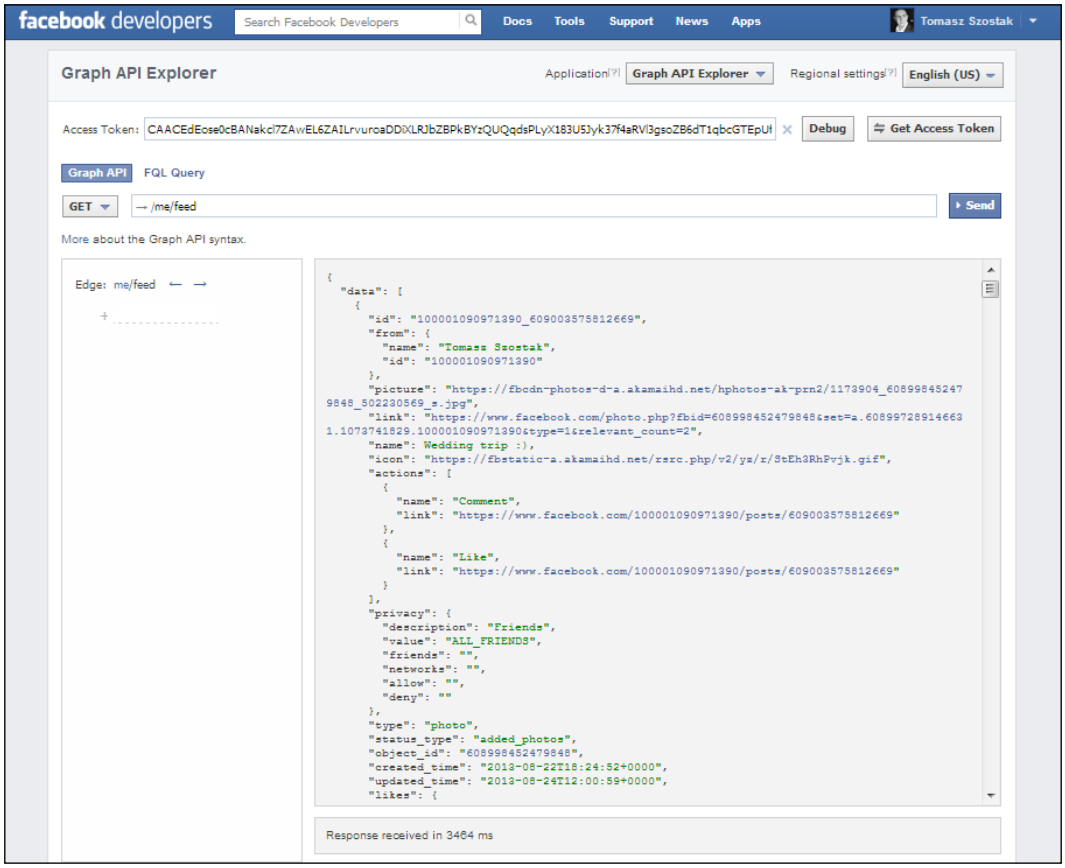
Later, we will talk about Facebook C# SDK, but now we can spend some time playing with the Facebook graph to find out how queries to Facebook should be built and how data is returned.

Using Graph API

The following site contains the **Graph API Explorer**, which gives us the ability to test queries and view the results our application returns:

`https://developers.facebook.com/tools/explorer`

Using the `me/feed` command, we tell our application to give us feeds from our Facebook wall ("me" means the currently logged-in user).



As we can see, returned data is in JSON format that should be parsed to objects or a list of objects. To receive feeds from our friend's wall, we should replace `"me"` with our friend's ID.

For more commands, check the following page, where all available commands and result documentation are available:

`https://developers.facebook.com/docs/reference/api`

Facebook SDK integration

Once we have an application ID and a secret key, we can start with developing the client application. The first thing to do is to attach the Facebook SDK to our project. We can do this by executing the following command in Nuget manager console, which is accessible by going to **Tools | Library Package Manager | Package Manager Console**.

```
PM> Install-Package Facebook
```

And the next command is to have the `FacebookClient` class with the methods that we will need:

```
PM> Install-Package Facebook.Client
```

These two commands will install and create references to the `Facebook` and `Facebook.Client` libraries, which will be used to send and receive data from Facebook. Having these in hand, we can start with developing our first connection mechanisms and extend our `MainViewModel` class with advanced functionality.

```
private async Task Authenticate()
{
  string message = String.Empty;
  try
  {
    session = await App.FacebookSessionClient.LoginAsync(
      "user_about_me,read_stream,read_mailbox");
    App.AccessToken = session.AccessToken;
    App.FacebookId = session.FacebookId;

    SaveSessionObject(session,new StorageSettingsProvider());

    Messenger.Default.Send<AuthenticationMessage>(new
AuthenticationMessage());
  }
  catch (InvalidOperationException e)
  {
    message = "Login failed! Exception details: " + e.Message;
    MessageBox.Show(message);
  }
}
```

The preceding code shows how to log in to Facebook and set the access token and Facebook session ID into our static storage. The first thing called in the code is the `LoginAsync` method that gets keywords. These keywords specify the information that our application will use; users have to confirm whether they want to use our application. If the authentication returns the session object, we can save it to our settings storage, without logging in, while it updates.

Now, let's have a look at our `MainViewModel` field.

```
private BitmapImage _myFBPhoto { get; set; }
```

This will store our Facebook profile image, and we place it next to our username (that is why we need the rights to read profile information). When we finish the authentication part, we should get information about the currently logged-in user.

```
FacebookClient fb = new FacebookClient(App.AccessToken);
fb.GetCompleted += OnFbOnGetCompleted;
fb.GetTaskAsync("me");
```

As you can see in the preceding code, we can get information about the currently logged-in user by creating the `FacebookClient` object, setting the `GetCompleted` event handler, and specifying what we want to get in the `GetTaskAsync` method. Executing this method with the `"me"` argument will result in getting information about the currently logged-in user. When the tasks are complete, the `OnFbOnGetCompleted` method is called.

```
private void OnFbOnGetCompleted(object o, FacebookApiEventArgs e)
        {
            if (e.Error != null)
            {
                return;
            }

            var result = (IDictionary<string, object>)
e.GetResultData();

            DispatcherHelper.UIDispatcher.BeginInvoke(() =>
                {
                    string name = String.Format("{0} {1}",
result["first_name"], result["last_name"]);
                    Uri myPictureUri =
                        new Uri(string.Format("https://graph.facebook.
com/{0}/picture?type={1}&access_token={2}",
                                                App.FacebookId,
"square", App.AccessToken));
```

```
                this.myFBPhoto = new BitmapImage(myPictureUri);
                this.UserName = name;
            });
        }
```

The preceding code handles the upcoming data by setting the `myFBPhoto` and `UserName` fields in our `ViewModel` class, which updates `View` via `NotifyPropertyChange`. One thing that might look strange to you is `DispatcherHelper`, the static class containing a static dispatcher. Why do we need that? Because we are working in a multithreaded environment, we need to think about simultaneous access to resources and one of the resources is our UI (view). A very helpful mechanism goes with MVVM Light—`DispatcherHelper`. Remember, `ViewModel` is not a control/page/container or any other UI element, so we cannot use the traditional dispatcher from Silverlight or WPF. That's why such a helper is a great idea. Let's look at another example—but more importantly let's look at how to parse the data received from the Facebook service. We will implement a Toast notification and a Tile update for our application that will display how many unread messages we have in our account. In the `SociAggBacgroundAgent` sample project, from the previous chapter, we will create the `SocialBL` class that will get information about messages in our account.

```
public async Task<int> GetUnreadFBMessagesCount()
{
  var fbSession = getSession();
  if (fbSession != null && fbSession.Expires>DateTime.Now)
  {
    FacebookClient fb = new FacebookClient(fbSession.AccessToken);

    dynamic msgs =
      await fb.GetTaskAsync("me/inbox?fields=unread");
    var model =
      JsonConvert.DeserializeObject<Models.Message>(
      msgs.ToString()) as Models.Message;
    if (model != null)
    {
      return model.summary.unread_count;
    }
  }
  if (fbSession == null)
  {
    return 0;
  }
  return 0;
}
```

This mechanism will work only if a user has logged in at least once to his/her Facebook account using our application. Then the session object will be stored into our settings storage, so that we can read it anywhere in our application. When we call `FacebookClient` with the `"me/inbox?fields=unread"` parameter, it means we want to get the unread messages from our inbox. Facebook returns to us with a very complex JSON object that has to be handled in our code. There are many powerful tools that handle serialization and deserialization, such as JSONConvert from the NewtonSoft JSON library. As you can see in the preceding code, we have successfully converted a complex JSON object into our `Model` class, in one line, which looks as follows:

```
public class Message
{
  private Summary _summary;
  public dynamic data { get; set; }
  public dynamic paging { get; set; }
  public Summary summary { get; set; }
}
```

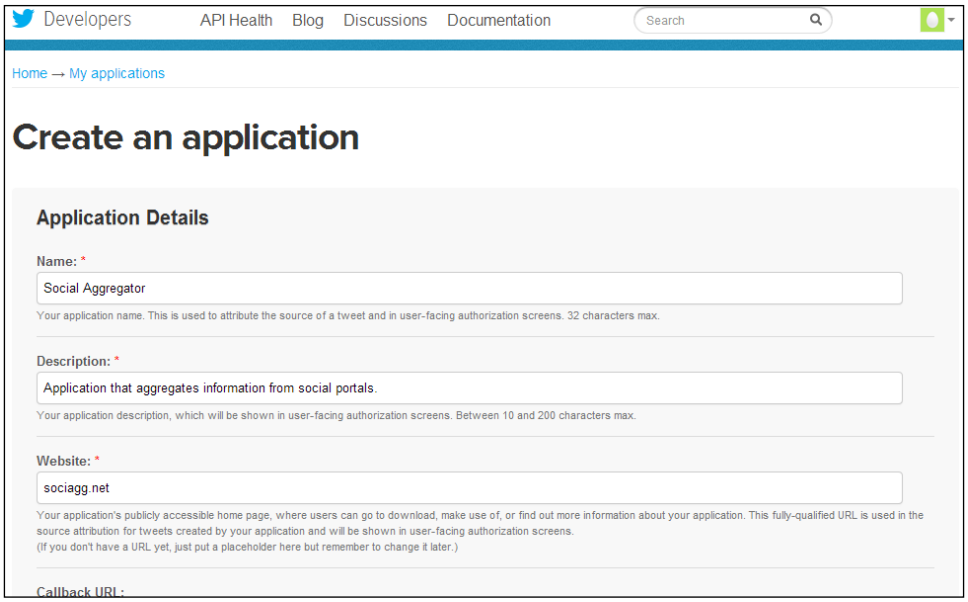
Of course, variable names should be similar to the names of the objects returned from Facebook. The object most interesting for us is the `Summary` object.

```
public class Summary
{
  public int unseen_count { get; set; }
  public int unread_count { get; set; }
  public DateTime updated_time { get; set; }
}
```

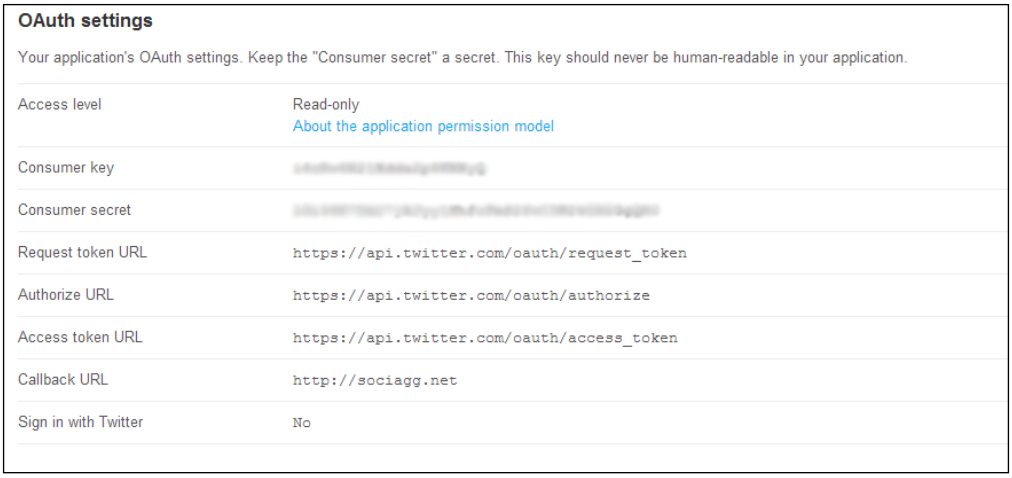
With the `unread_count` field, it will be set to the proper value during deserialization from JSON. And that is how we manage to get information about one aspect of our account. Now we will see how simple it is to get information from Facebook—there is no problem getting the **Notifications** count or **Friends Invitations** information.

Twitter integration

Just as we saw with Facebook integration, we should start with registering our application in Twitter for developers. If you have a Twitter account, just go to `https://dev.twitter.com/apps/new` and provide the necessary information about the new Twitter application.



When you are done with configuring the Twitter application, get the consumer key and consumer secret key that you should put in your application to get the access token from Twitter.



Once we have the consumer key and consumer secret key, we can create static variables that will keep these values in the application—we are going to use that in our Twitter communication. The Twitter API uses the OAUTH standard (`http://oauth.net/`). The next thing to do is to download and reference `TweetSharp` in our project; for this, we have to execute the following Nuget console line:

```
PM> Install-Package TweetSharp
```

Once we reference the `TweetSharp` project, we can proceed with creating functionality for logging in and getting profile information . Of course, there are many more features available in `TweetSharp`, which are listed as follows:

- Getting direct messages
- Getting followers
- Getting tweets
- Sending tweets

The implementation of these features is demonstrated next. Before we start with creating the new application page to log in to Twitter, we should create additional fields in our `ViewModel` class, such as `TwitterUserName` and `myTwitterPhoto` in `MainViewModel`. These variables will keep information about our account. Other fields that should be added are static fields that will be used to manage calls to the Twitter service. This is demonstrated in the following code snippet:

```
public static readonly string TwitterConsumerKey =
  "YourConsumerSecret";
public static readonly string TwitterSecretKey =
  "YourSecretKey";
public static readonly string TwitterCallbackUrl =
  "YourCallbackUrl";
public static readonly string TwiterSessionObjectKey =
  "TwitterSession";
```

The Twitter session static key will be used to read and save the Twitter authentication token to log in to the application once and use this session. First of all, we need to add a new application page in the **Views** folder; we need to handle Twitter logging in in the `WebBrowser` control (this allows us to display HTML and browse the Internet) and handle events in the code behind; so it will make our clean code a bit ugly. Our new page should inherit from `ApplicationPageBase` to support sending and receiving messages. In page constructor, we will call the `LoginToTwitter` method:

```
private void LoginToTwitter()
{
```

```
    twitterService = new TwitterService(
      Statics.TwitterConsumerKey, Statics.TwitterSecretKey);
    twitterService.GetRequestToken(
      Statics.TwitterCallbackUrl, CallBackToken);
}
```

This is why we need static fields that keep Twitter settings, so that we can check it in the Twitter application. The `GetRequestToken` method requests authentication using the `CallBackToken` method, which navigates the `WebBrowser` control to the authentication URI.

```
private void CallBackToken(OAuthRequestToken rt,
  TwitterResponse response)
{
  if (response.StatusCode == HttpStatusCode.Unauthorized)
  {
    return;
  }
  Uri uri = twitterService.GetAuthorizationUri(rt);
  requestToken = rt;
  webBrowser.Dispatcher.BeginInvoke(() =>
    webBrowser.Navigate(uri));
}
```

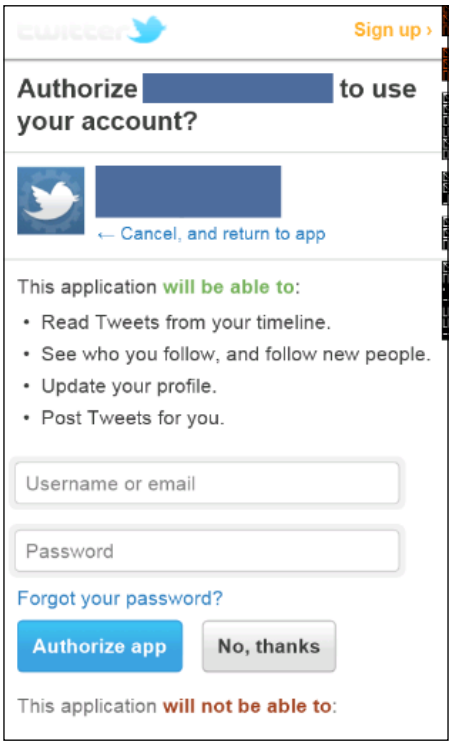
Execution of the preceding code causes it to display the login page in the `WebBrowser` control. The login page contains a list of rights that will be required to use applications—these rights should be set in the **Twitter for the developers'** part while defining or editing applications. Usually, Twitter authentication works in the following way:

1. Enter username and password.
2. If the authentication succeeds, the page with the verification key will be displayed.
3. The user copies the verification key and goes back to application to authenticate.

We want to create as simple a mechanism as possible, without forcing the user to enter any code. We can do it in the following way:

1. Enter the username and password.
2. If the authentication succeeds, a callback URL with the verification code is captured and parsed.

3. We get the access token.



To implement the preceding approach, we should handle the `OnNavigating` `WebBrowser` event as shown in the following code snippet:

```
private void WebBrowser_OnNavigating(
  object sender, NavigatingEventArgs e)
{
  if (e.Uri.ToString().Contains(
    Statics.TwitterCallbackUrl + "?oauth_token"))
  {
  var cb = new Action<OAuthAccessToken,
    TwitterResponse>(CallBackVerifiedResponse);
  var values = ParseQueryString(e.Uri.AbsoluteUri);
  string verifier = values["oauth_verifier"];
  twitterService.GetAccessToken(
    requestToken, verifier, CallBackVerifiedResponse);

  }
}
```

The preceding code captures all URLs that are passed through the `WebBrowser` control and looks for the authentication code. If it finds the verification code, it passes it to the `GetAccessToken` method. When the method execution is completed, it fires the `CallBackVerifiedResponse` method, which takes the authentication token and the response object.

```
private void CallBackVerifiedResponse(
  OAuthAccessToken at, TwitterResponse response)
{
  if (at != null)
  {
    AccessToken = at.Token;
    if (string.IsNullOrEmpty(AccessToken))
    {
      //failed
      return;
    }
  AccessTokenSecret = at.TokenSecret;
  DispatcherHelper.UIDispatcher.BeginInvoke(() =>
  {
    SaveTwitterSessionObject(at,new StorageSettingsProvider());
    var msg = new TwitterAuthenticationMessage();
    msg.IsAuthenticated = true;
    Messenger.Default.Send<TwitterAuthenticationMessage>(msg);
    Messenger.Default.Send(new Uri("/MainPage.xaml",
      UriKind.Relative), "NavigationRequest");

    });
  }
}
```

The `CallBackVerifiedResponse` method saves the authentication token to settings storage for later use. The next message informs `MainViewModel` that the authentication succeeded and the page is redirected to `MainPage`. The `MainViewModel` class receives the message that the authentication is done, and gets information about our profile from Twitter. As you can see in the following code, having the authentication token work with `TweetSharp` becomes very easy:

```
private void updateTwitterUserInfo (OAuthAccessToken TwitterToken)
{
  var service = new TwitterService(
    Statics.TwitterConsumerKey, Statics.TwitterSecretKey);
```

```
    service.AuthenticateWith(TwitterToken.Token,
      TwitterToken.TokenSecret);
    service.GetUserProfile(new GetUserProfileOptions()
    { IncludeEntities = false, SkipStatus = false },
    (user, reponse) => DispatcherHelper.UIDispatcher.BeginInvoke(
      ()=>
      {
        TwitterUserName = user.Name;
        myTwitterPhoto = new BitmapImage(
          new Uri(user.ProfileImageUrl));
          }));
}
```

The main part of the preceding code is the `GetUserProfile` method that enables us to get our Twitter profile information, and then the return data that can be set to appropriate fields. The preceding example shows us how to use the lambda expression (`http://msdn.microsoft.com/en-us/library/vstudio/bb397687.aspx`) instead of creating a new method that handles the callback.

Getting other information from Twitter and sending tweets can be implemented similarly. The code available for this chapter creates foundations for the application that can be submitted to the app hub.

Summary

In this chapter, we have taken our first steps towards enabling social media in our application. We learned how to create a Facebook application and use its Graph API, how to implement `TweetSharp` in our application, and how to register a Twitter application. The code samples available for this chapter will explore the basic tasks in social portal usage.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Thank you for buying
Windows Phone 8 Application Development Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING

Developing Windows Store Apps with HTML5 and JavaScript

ISBN: 978-1-849687-10-2 Paperback: 184 pages

Learn the key concepts of developing Windows Store apps using HTML5 and JavaScript

1. Learn about the powerful new features in HTML5 and CSS3
2. Quick start a JavaScript app from scratch
3. Get your app into the store and learn how to add authentication

---

Microsoft Windows PowerShell 3.0 First Look

ISBN: 978-1-849686-44-0 Paperback: 200 pages

A quick, succinct guide to the new and exciting features in PowerShell 3.0

1. Explore and experience the new features found in PowerShell 3.0
2. Understand the changes to the language and the reasons why they were implemented
3. Discover new cmdlets and modules available in Windows 8 and Server 8
4. Quickly get up to date with the latest version of Powershell with concise descriptions and simple examples

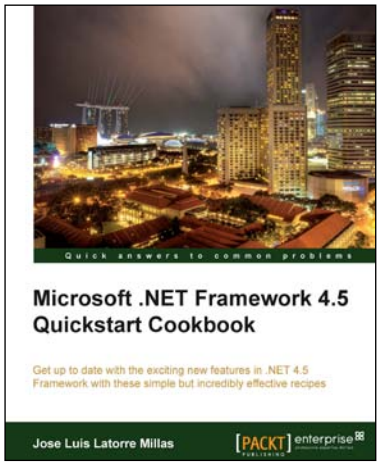
[PACKT] PUBLISHING

Mastering Windows 8 C++ App Development

ISBN: 978-1-849695-02-2 Paperback: 304 pages

A practical guide to developing Windows Store apps with C++ and XAML

1. Details the most important features of C++, XAML, and WinRT for building fantastic Windows Store apps
2. Full of detailed and engaging code samples that can be used as a basis for your own projects
3. Provides a clear overview of Windows Runtime and C++/CX



Microsoft .NET Framework 4.5 Quickstart Cookbook

ISBN: 978-1-849686-98-3 Paperback: 226 pages

Get up to date with the exciting new features in NET 4.5 Framework with these simple but incredibly effective recipes

1. Designed for the fastest jump into .NET 4.5, with a clearly designed roadmap of progressive chapters and detailed examples.
2. A great and efficient way to get into .NET 4.5 and not only understand its features but clearly know how to use them, when, how, and why.
3. Covers Windows 8 XAML development, .NET Core (with Async/Await and reflection improvements), EF Code First and Migrations, ASP.NET, WF, and WPF