Microsoft

Build Windows 8 Apps with Microsoft Visual C++

Luca Regnicoli
Paolo Pialorsi
Roberto Brunetti

ebook+exercises

Step by Step

Microsoft

Build Windows 8 Apps with Microsoft Visual C++ Step by Step

Luca Regnicoli
Paolo Pialorsi
Roberto Brunetti

ISBN: 978-0-7356-6723-5

1 2 3 4 5 6 7 8 9 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Christopher Hearse

Editorial Production: Zyg Group, LLC

Technical Reviewer: John Mueller

Copyeditor: Zyg Group, LLC

Indexer: Zyg Group, LLC

Cover Design: Twist Creative • Seattle

Cover Composition: Zyg Group, LLC

Illustrator: Rebecca Demarest

This book is dedicated to Barbara.

—ROBERTO BRUNETTI

This book is dedicated to my parents. Thanks!

—PAOLO PIALORSI

This book is dedicated to my mother, Vanna, the strongest woman I have ever known.

—LUCA REGNICOLI

Contents at a Glance

Contents

Introduction

Windows 8 is Microsoft's newest operating system, intended to let developers fluent in various programming languages—such as C++, C#, or JavaScript—leverage its powerful infrastructure to build applications using a brand-new library called the Windows Runtime API.

This book provides an organized walk-through of the features, APIs, and user experience in Windows 8. The content is *introductory*—it discusses each component from a theoretical viewpoint interspersed with basic but effective code samples, which you can follow to get a jump-start in developing for the Windows 8 platform.

The book provides coverage of almost all the main Windows 8 aspects and features, and it offers essential guidance in learning them using the classic Step by Step approach.

In addition to its coverage of core Windows 8 features using C++, the book discusses some related aspects, such as Windows Communication Foundation (WCF) Data Services, Open Data Protocol (OData), ADO.NET Entity Framework, and applications architecture. Beyond the explanatory content, each chapter includes a rich set of step-by-step examples, as well as downloadable sample projects that you can explore for yourself.

Who should read this book

This goal of this book is to provide experienced C++ developers with the information they need to begin working with the main components of the Windows 8 operating system and the Windows Runtime. Starting with the Windows Runtime APIs, the book moves readers through a comprehensive discussion of the new user experience, including how to design interfaces that work for the keyboard, the mouse, and touch screens. This book does not teach C++; readers need a solid knowledge of the C++ language to fully understand the code presented in the book and to follow along by performing the exercises using Microsoft Visual Studio 2012. This book is also useful for software architects conversant with C++ who need an overview of the components they would plan to include in the overall architecture of a real-world Windows 8 solution.

Who should not read this book

If you have worked with Windows 8 already, this book is probably not for you. It is an introductory guide to developing applications that leverage the platform using C++.

Assumptions

To get the most out of this book, you should have at least a minimal understanding of C++ development and object-oriented programming concepts. Although you can also develop for Windows 8 using any .NET language or JavaScript, this book includes examples in C++ only.

In addition to the C++ language, the examples in Chapter 10, "Architecting a Windows 8 app," assume you have a basic understanding of ASP.NET and WCF, although the code presented for those examples doesn't use any advanced features of either technology.

Organization of this book

This book is divided into 10 chapters, each of which focuses on a different aspect or technology within the Windows 8 operating system and Windows Runtime APIs.

Finding your best starting point in this book

We suggest that you start reading the book from the beginning. By following this path, you will discover all of the aspects of the new look and feel, the new user experience, and the new user interface for touch-based devices required for building successful Windows 8 applications. Chapter 2, "Windows 8 user interface style," is particularly important, because you need to understand the design concepts underlying the Windows 8 UI style. Chapter 3, "My first Windows 8 app," is the fundamental starting point for building your first Windows 8 application. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to Windows 8 development	Start with Chapter 1.
New to Windows 8 UI style	Start with Chapter 2.
Not new to Windows 8 development using the provided templates	Start with Chapter 4.

A XAML developer	Start with Chapter 3 and then skip to Chapter 9 to gain a solid understanding of the controls specific to Windows 8 apps and how to design flexible layouts.

Most of the book's chapters include hands-on procedures and examples that let you try out the concepts discussed in each chapter. No matter which sections you choose to focus on, be sure to download the companion code from the publisher's site (see the "Code samples" section of this introduction), and install them on your system.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 8 installed
- Visual Studio 2012, any edition tailored for Windows 8 (the Express edition for Windows 8 is free)
- Computer with a 1.6 GHz or faster processor

- 1 GB of RAM (1.5 GB if running on a virtual machine)
- 10 GB (NTFS) of available hard disk space
- 5400 RPM (or faster) hard disk drive
- DirectX 9-capable video card running at 1024 × 768 or higher display resolution

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2012.

Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects are available for download from the book's page:

http://aka.ms/BuildW8AppsVCSbS/files

Note In addition to the code samples, your system must have Microsoft Visual Studio 2012 installed.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Windows8cplusplusStepByStep.zip file.

Acknowledgments

We'd like to thank all the people who supported us in writing this book.

Marco Russo has been involved with all of us in the most important phases of writing this book and its twin, *Build Windows 8 Apps with Microsoft Visual C# and Visual Basic Step by Step* (Microsoft Press, 2013).

Vanni Boncinelli tested all the code we wrote.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/BuildW8AppsVCSbS/errata

If you find an error that is not already listed, you can report it to us through the same page. If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

Introduction to Windows Store apps

After completing this chapter, you will be able to

- Understand the main features of a Windows Store App.
- Evaluate the key benefits of creating a Windows Store app for Windows 8.
- Recognize the main capabilities and features of the new Windows 8 operating system.

This chapter provides an overall introduction to Microsoft Windows 8 and to the new world of Microsoft Windows Store apps, from a developer's perspective.

The Windows 8 experience

Windows 8 is one of the most innovative and revolutionary investments made by Microsoft in the last decade in the operating systems area. Prior to Windows 8, the operating systems market consisted of three main families: server operating systems, client/desktop operating systems, and mobile/tablet-oriented operating systems.

Windows 8, together with its sibling Microsoft Windows Server 2012, introduces a new paradigm wherein the client/desktop and mobile/tablet-oriented operating systems can be combined, sharing features, capabilities, user interfaces (UIs), and behaviors. In the last few years, the market for tablet devices has exploded, with an increasing number of people working at home and in their offices on a small tablet device. Nevertheless, until the release of Windows 8, it wasn't a simple matter to reconcile the needs of end users using tablet devices with the infrastructural constraints of corporate networks. For example, many tablet end users would like to install software from a trusted and secure online marketplace, regardless the corporate policies of the individual end user's company. Moreover, a common end-user need is to check corporate email accounts as well as any private email accounts using a unique device and unique email client software. Furthermore, the increase in social-media use leads to the sharing of private contacts, agendas, tasks, pictures, and instant messages with business contacts, meetings, and corporate network instant communication and videoconferencing.

However, technology without governance could become a nightmare both for end users and IT professionals. With Windows 8, end users can leverage a corporate-provided tablet device and install software from a safe and secure marketplace (either public or corporate constrained), check multiple email accounts while complying with company security policies, and socialize with friends, colleagues, and business contacts, all within a safe and sandboxed environment.

Moreover, for the sake of backward compatibility, all the software created targeting Windows 7 desktops will still continue to work on Windows 8, using the old-style desktop-oriented approach.

So, let's see the new Windows 8 user interface and the key features of this new operating system. Figure 1-1 shows the new Start screen, one of the revolutionary features introduced with Windows 8.

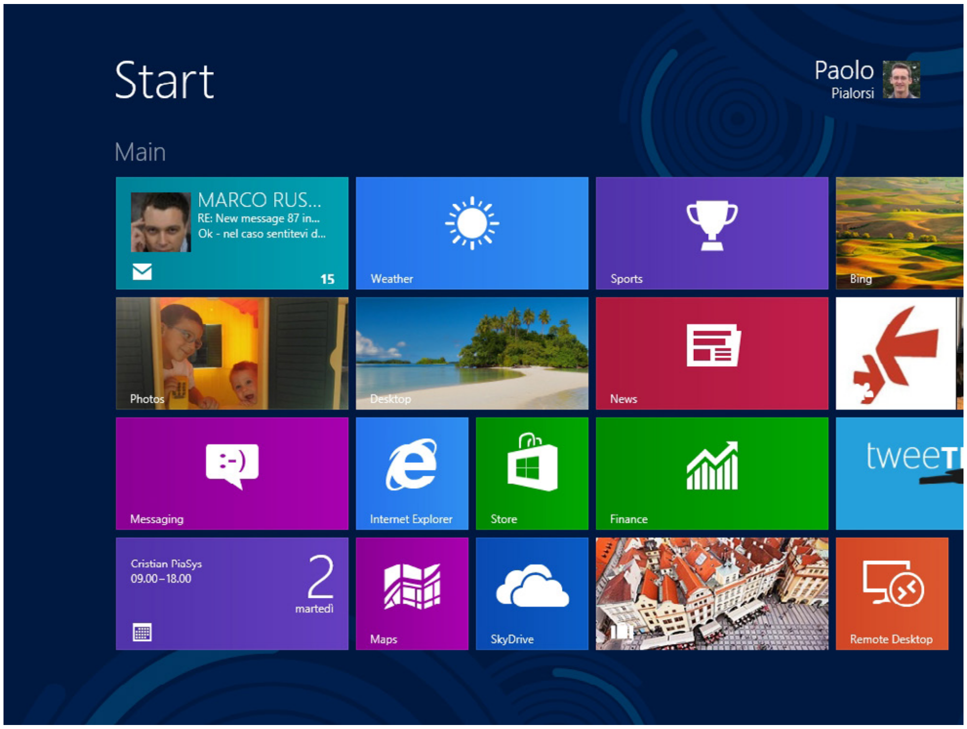


FIGURE 1-1 The Windows 8 Start screen.

The new Start screen is made up of a set of squares and rectangles called *tiles*, each of which represents a link to a software application. Each tile can also provide an animated feedback to the end user. Tiles can be small (square) or wide (rectangle). Many apps provide both sizes, letting end users choose between them in the main screen according to personal preference. For example, in the top-left corner of Figure 1-1, just under the Main title, you can see a wide tile for the Mail app, which indicates that there are 15 email messages to read in the inbox. The tile also provides a short preview of the messages.

To reduce the size of the tile, you can right-click it or swipe your finger downward on the tile, which both selects it and activates a command bar, called the app bar, which will be discussed later. Figure 1-2 shows the Mail app tile selected.

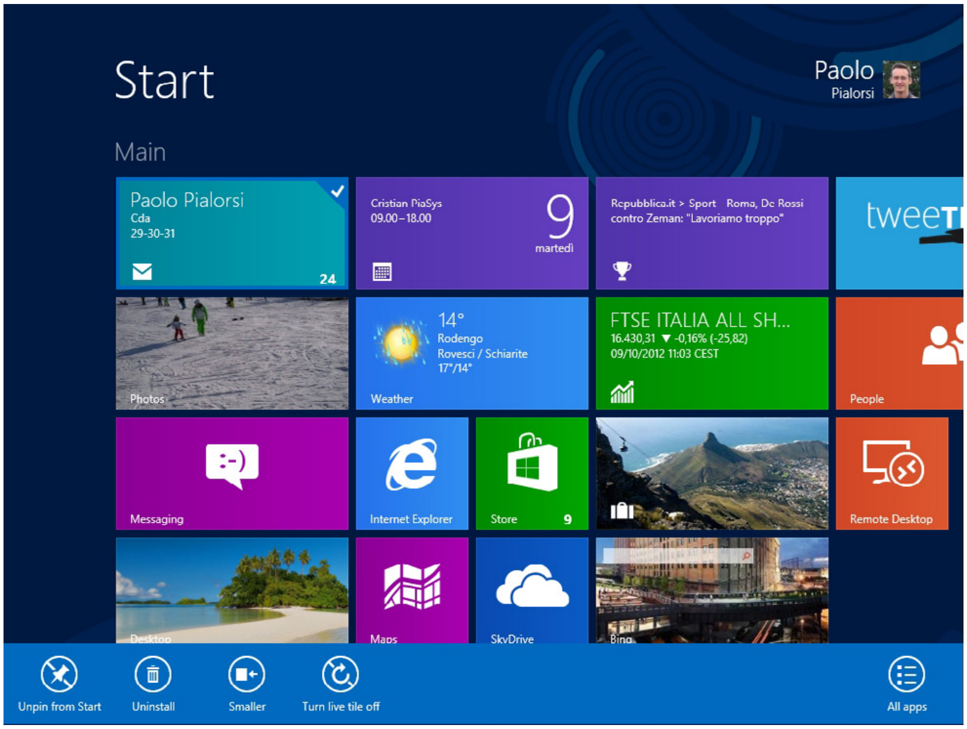


FIGURE 1-2 The app bar of the Start screen.

The app bar may contain many active commands, which vary according to context. For example, with a tile selected, you can select the Smaller command to change the tile from wide to square, if the tile you selected is wide. You can also turn off dynamic updating of the tile by selecting Turn Live Tile Off. Or you can select Uninstall to remove the app from your device completely. If you select the Smaller command, the tile will become square and the preview of the unread email will disappear. You can see the result in Figure 1-3.

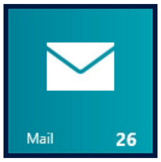


FIGURE 1-3 The small tile of the Mail app.

A user with a tablet device can tap (i.e., touch using a single finger) one of these tiles to start an application instance or to resume an already running instance. Similarly, a user with a desktop PC and a mouse can click the tile and get the same result. The Start screen is based on the idea of the "panoramic view" that has been available in Windows Phone since version 7. In a panoramic view, you can scroll horizontally, using either touch gestures on a tablet/touch screen or the mouse wheel, touchpad, or keyboard if you are using a laptop or desktop. You can also use the traditional scrollbar that appears at the bottom of the screen.

As soon as you tap an app tile, the foreground application becomes the app you selected. When you are starting that app for the first time in a given Windows 8 session, Windows creates the instance and loads it into memory. Otherwise, if the app is already running, Windows promotes it to the foreground application. In both cases, whatever application was previously in the foreground is sent into the background, where it may be *suspended* by the operating system. Suspension means freezing: the app gets no CPU threads and no I/O capability, leaving all the computer resources free to support the main (foreground) application. If the user later returns to a suspended application, the operating system resumes it in its previous state. Later in Chapter 4, "Application life-cycle management," you will learn more about the application life cycle for Windows Store apps. Figure 1-4 shows the Windows Store app Bing Weather running in the foreground.

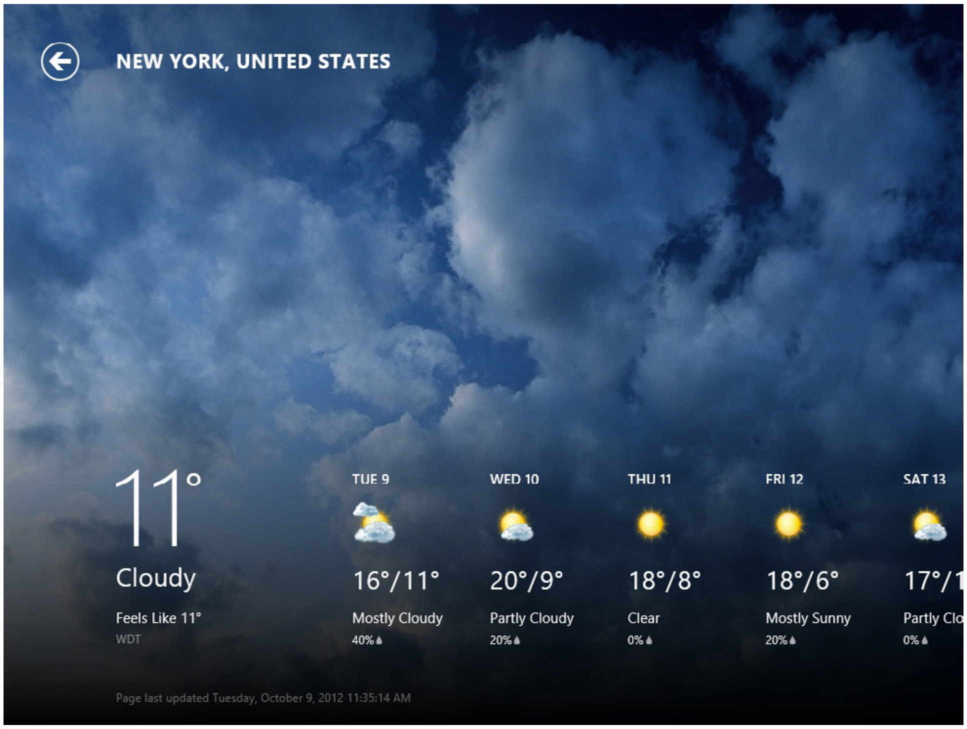


FIGURE 1-4 The Bing Weather app running in the foreground.

In Figure 1-4, the app takes up the entire screen, which satisfies one of the main ideas of the user experience design for Windows Store apps: “content, not chrome.” Chapter 2, “Windows 8 user interface style,” will help you more fully understand the meaning of that phrase.

There are some exceptions; not all applications are Windows Store apps. For example, if you launch an old-style desktop application, Windows switches to the classic Windows Desktop, just as if you were in a previous version of Windows. Figure 1-5 shows an older desktop-style application, Microsoft SQL Server Management Studio. Note the absence of the classic Start button.

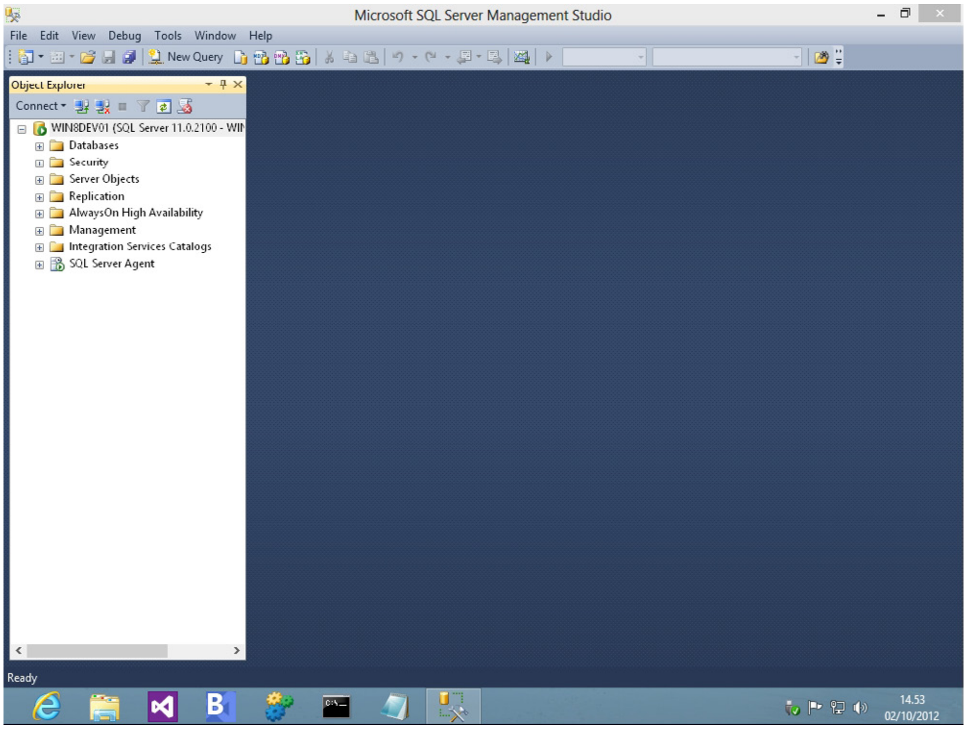
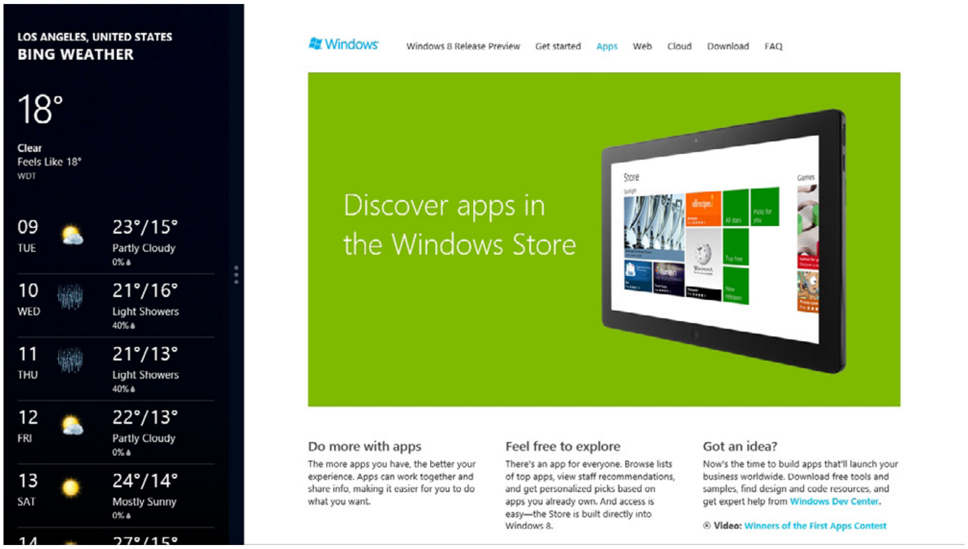

Microsoft SQL Server Management Studio
File Edit View Debug Tools Window Help
New Query
Object Explorer
Connect
WIN8DEV01 (SQL Server 11.0.2100 - WIN
Databases
Security
Server Objects
Replication
AlwaysOn High Availability
Management
Integration Services Catalogs
SQL Server Agent
Ready
14.53
02/10/2012

FIGURE 1-5 A standard desktop application in Windows 8.

You aren’t limited to one application at a time open. If you have a device with a wide (16:9 or 16:10) screen aspect ratio, you can snap two applications onto the screen at the same time. For example, Figure 1-6 shows the Bing Weather app snapped on the left, with the new Internet Explorer 10 for Windows 8 on the right.


LOS ANGELES, UNITED STATES
BING WEATHER
18°
Clear
Feels Like 18°
WDT
09 TUE 23°/15° Partly Cloudy 0%
10 WED 21°/16° Light Showers 40%
11 THU 21°/13° Light Showers 40%
12 FRI 22°/13° Partly Cloudy 0%
13 SAT 24°/14° Mostly Sunny 0%
Windows
Windows 8 Release Preview
Get started
Apps
Web
Cloud
Download
FAQ
Discover apps in the Windows Store

Do more with apps

The more apps you have, the better your experience. Apps can work together and share info, making it easier for you to do what you want.

Feel free to explore

There's an app for everyone. Browse lists of top apps, view staff recommendations, and get personalized picks based on apps you already own. And access is easy—the Store is built directly into Windows 8.

Got an idea?

Now's the time to build apps that'll launch your business worldwide. Download free tools and samples, find design and code resources, and get expert help from Windows Dev Center.

Video: Winners of the First Apps Contest

FIGURE 1-6 A couple of apps running in the new Windows 8 snapped view mode.

Of course, you can also switch the sizes of the two snapped apps, as shown in Figure 1-7.

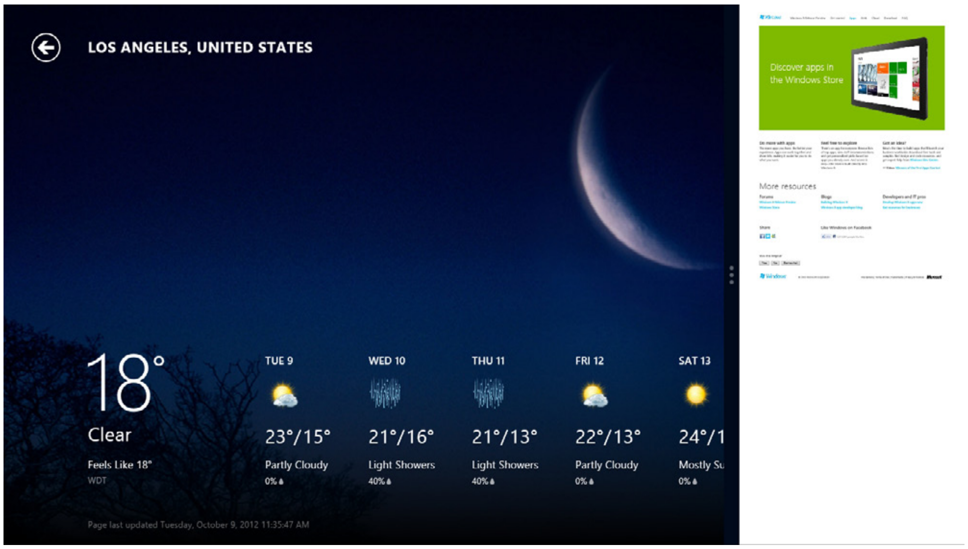

LOS ANGELES, UNITED STATES
18°
Clear
Feels Like 18°
WDT
TUE 9 23°/15° Partly Cloudy 0%
WED 10 21°/16° Light Showers 40%
THU 11 21°/13° Light Showers 40%
FRI 12 22°/13° Partly Cloudy 0%
SAT 13 24°/1 Mostly Su 0%
Page last updated Tuesday, October 9, 2012 11:35:47 AM
Discover apps in the Windows Store
More resources

FIGURE 1-7 Another configuration of the new snapped view mode of Windows 8.

From a developer's perspective, the most important thing to understand at this point is that every Windows Store app must support snapping to be certified by the Windows Store. Bing Weather, as you saw in previous figures, supports the snapped view by adapting the layout of the page to present the information in a smaller horizontal portion of the screen. If you create an app that cannot present information in this manner, you must fill the snapped view with a clear message for the user—you would never use the full-screen view for a snapped view because the user would not be able to interact properly with the application.

In fact, whenever you want to publish a Windows Store app, you have to submit it to the Windows Store (or eventually to a corporate Enterprise Store) for approval. From the public, official Windows Store point of view, an app must adhere to a clear set of requirements before it will be certified. Any application that does not adhere to these requirements will be rejected. You can find complete details about the requirements on the official page available on the Windows 8 developer section of Microsoft Developer Network (MSDN): *http://msdn.microsoft.com/library/windows/apps/hh694083.aspx*. For example, one rule states that if your app connects to the Internet for any purpose, you must provide a privacy information page. Thus, if your app invokes a remote web service, which is a common situation, you must provide a privacy page that explains how you manage users' data. Chapter 4 discusses the process of submitting an app to the Windows Store in more detail.

Going back to the Start screen, another useful bit of information is that you can arrange tiles in groups, which helps organize them on the Start screen. To move a tile from one group to another, you simply need to drag and drop it, using either touch gestures or the mouse. To create a new group, you move a tile into the middle region between two existing groups. When you do that, a gray bar appears, representing the frame of the new group. Dropping the tile onto this gray bar creates a new group.

If you zoom out the Start screen, by using a "pinch" gesture (explained in Chapter 2) or by scrolling the mouse wheel backward while pressing Ctrl, the Start screen changes. You can assign a name to a group by clicking it or swiping your finger down on the group to select it and then clicking the Name Group button in the bottom app bar. Figure 1-8 shows the Start screen while zoomed out, with a group of tiles selected and the bottom app bar showing the available commands.

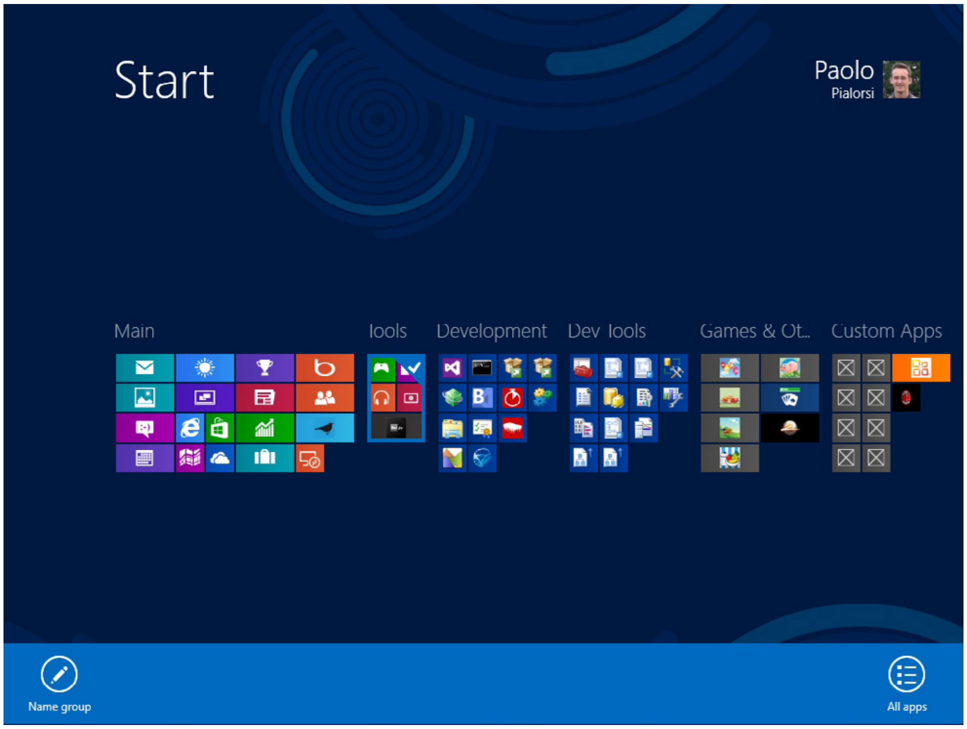
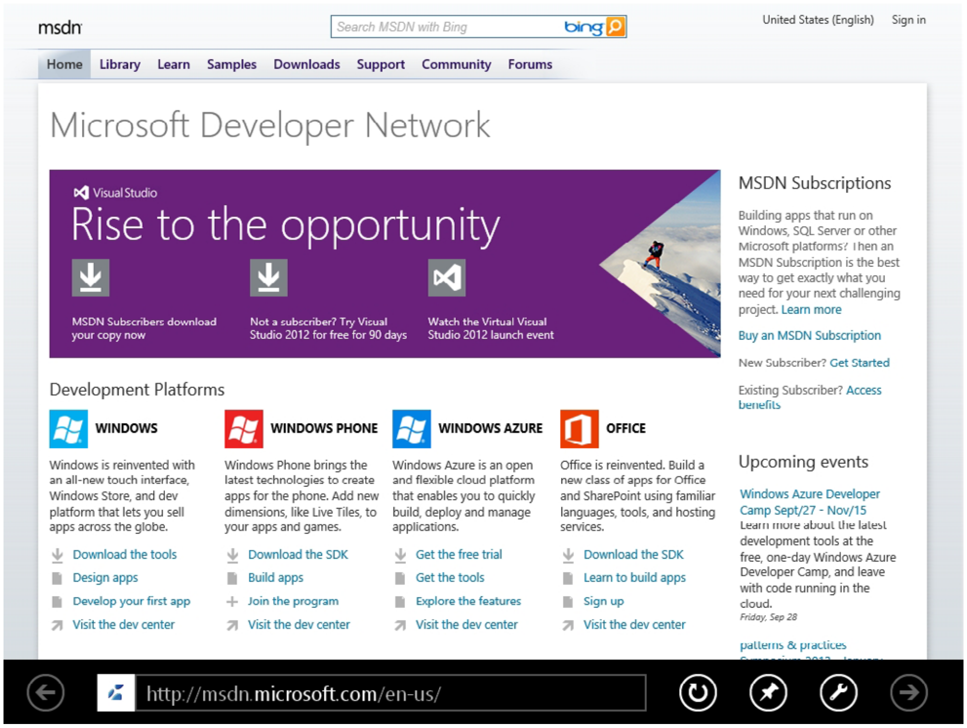


FIGURE 1-8 The Windows 8 Start screen when it is zoomed out.

Charms bar and app bars

Other new and key features of Windows 8 are the app bars and the charms bar. Chapter 2 discusses these in more detail as well as the philosophy behind them. For now, simply consider that these changes arose from the need to support new devices such as tablets and smartphones, where users interact primarily with their hands, through touch. This new touch-oriented perspective necessarily introduced new tools and solutions. Using the bottom app bar, you can manage tasks and actions related to the current context or item. You can see an example in Figure 1-9, where Internet Explorer 10 for Windows 8 displays the bottom app bar so the user can edit the current URL, refresh the page, pin the page on the new Start screen, or change the browser settings.


msdn | Search MSDN with Bing | United States (English) Sign in

Home Library Learn Samples Downloads Support Community Forums

Microsoft Developer Network

Visual Studio

Rise to the opportunity

MSDN Subscribers download your copy now

Not a subscriber? Try Visual Studio 2012 for free for 90 days

Watch the Virtual Visual Studio 2012 launch event

MSDN Subscriptions

Building apps that run on Windows, SQL Server or other Microsoft platforms? Then an MSDN Subscription is the best way to get exactly what you need for your next challenging project. Learn more

Buy an MSDN Subscription

New Subscriber? Get Started

Existing Subscriber? Access benefits

Development Platforms

WINDOWS

Windows is reinvented with an all-new touch interface, Windows Store, and dev platform that lets you sell apps across the globe.

- Download the tools
- Design apps
- Develop your first app
- Visit the dev center

WINDOWS PHONE

Windows Phone brings the latest technologies to create apps for the phone. Add new dimensions, like Live Tiles, to your apps and games.

- Download the SDK
- Build apps
- Join the program
- Visit the dev center

WINDOWS AZURE

Windows Azure is an open and flexible cloud platform that enables you to quickly build, deploy and manage applications.

- Get the free trial
- Get the tools
- Explore the features
- Visit the dev center

OFFICE

Office is reinvented. Build a new class of apps for Office and SharePoint using familiar languages, tools, and hosting services.

- Download the SDK
- Learn to build apps
- Sign up
- Visit the dev center

Upcoming events

Windows Azure Developer Camp Sept/27 - Nov/15

Learn more about the latest development tools at the free, one-day Windows Azure Developer Camp, and leave with code running in the cloud.

Friday, Sep 28

patterns & practices

http://msdn.microsoft.com/en-us/

FIGURE 1-9 The new Internet Explorer 10 UI.

The top app bar provides navigation assistance to end users. For example, you might use it to show a top-level menu or a list of main sections available in the current app. Figure 1-10 shows the top app bar of the Windows Store app, which is the app you can use to search, download, buy, and install other apps.

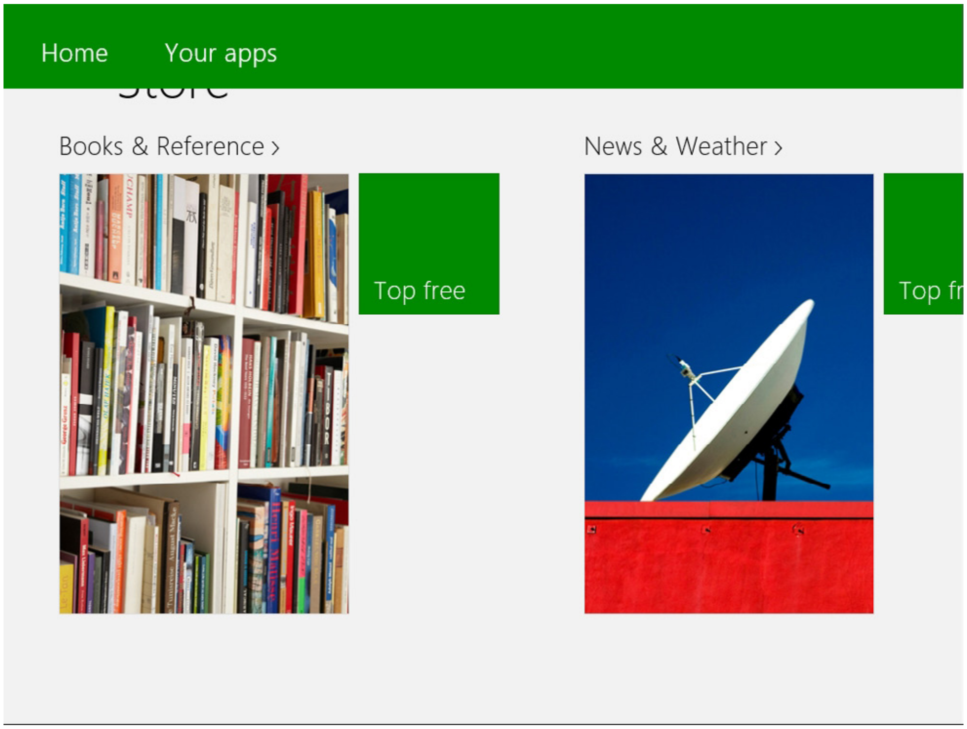


FIGURE 1-10 The top app bar of the Windows Store App.

To show the top and bottom app bars, swipe your finger from the top or bottom border of the screen toward the center of the screen. Alternatively, you can press Windows+Z or right-click the mouse.

Finally, the charms bar allows you to access useful features and actions provided by the operating system, regardless of where you are. For example, you can use the charms bar to access system settings, the local search engine, Sharing features, and so on. Figure 1-11 shows the charms bar in action.

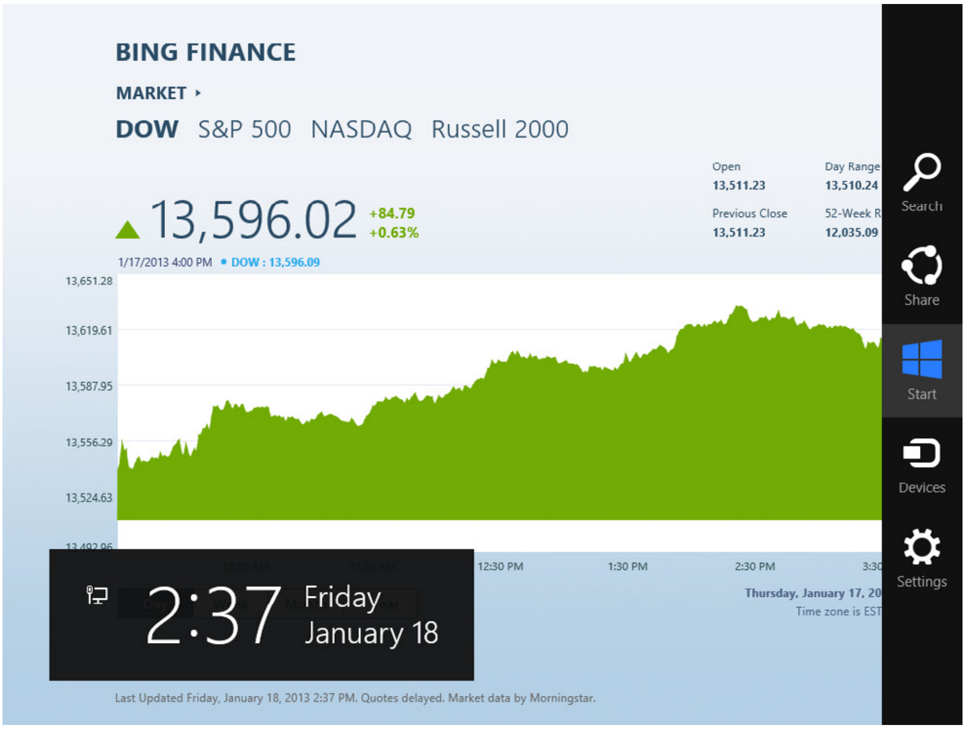


FIGURE 1-11 The Windows 8 charms bar.

To show the charms bar, swipe your finger from the right border of the screen toward the center of the screen. Alternatively, you can press Windows+C on the keyboard. You can also move the mouse pointer to the lower- or upper-right corner of the screen. Finally, you can activate specific charms bar commands directly using keyboard shortcuts. For example, pressing Windows+Q activates a search for installed applications (*Q* = query), while pressing Windows+F (*F* = find files) activates a search for files. To activate the sharing feature, press Windows+H.

Through the charms bar, you can activate specific panels such as the Settings panel, which you can also activate by pressing Windows+I. Figure 1-12 shows the Settings panel in action.

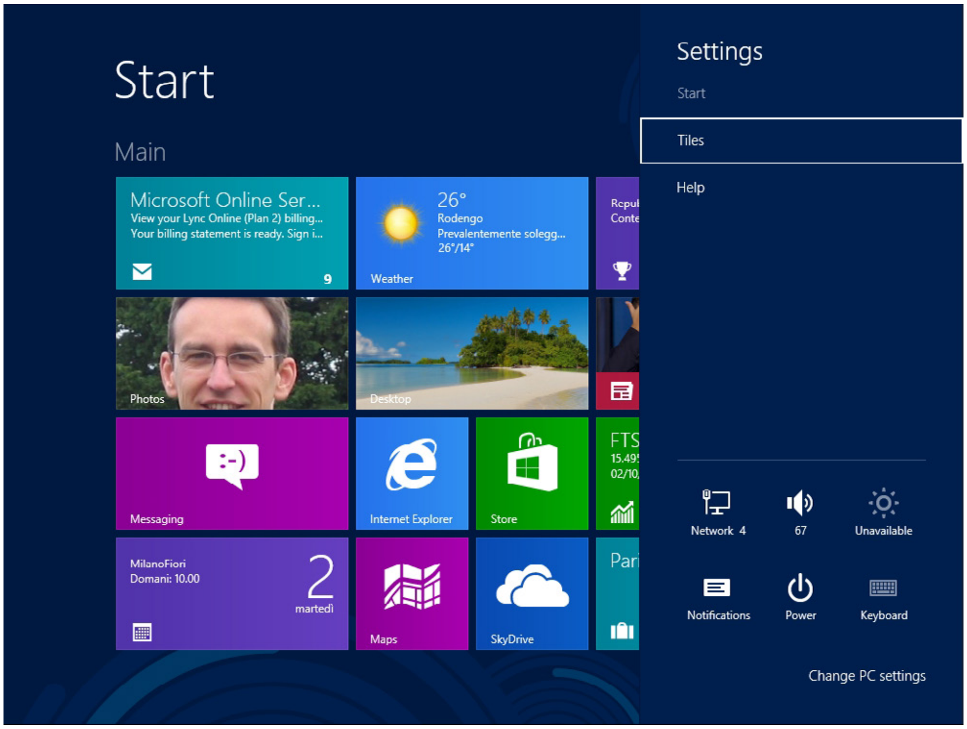
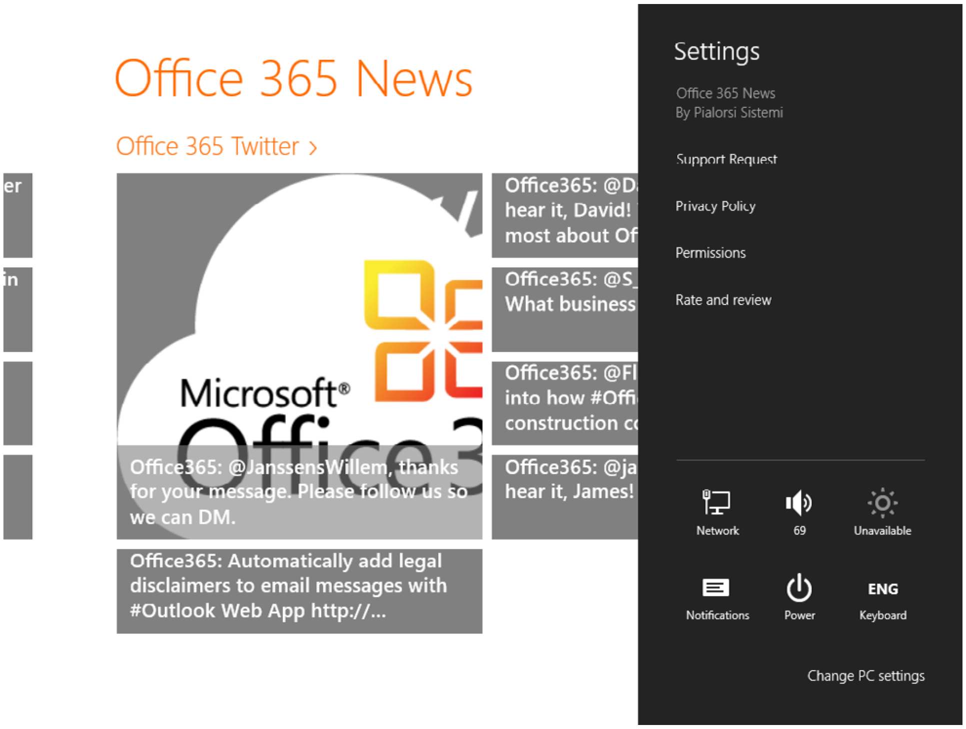


FIGURE 1-12 A flyout configuration panel for managing the settings from the charms bar.

One key feature of the charms bar is that you can also host custom commands and custom panels in it. For example, if you are developing a Windows Store app and you want to provide some custom settings for end users, you can add a command to the charms bar. By pressing the custom command while your app is in foreground, you can activate a flyout panel, which is a custom control that renders within the charms bar (see Figure 1-13).



Office365: @JanssensWillem, thanks for your message. Please follow us so we can DM.

Office365: Automatically add legal disclaimers to email messages with #Outlook Web App http://...

FIGURE 1-13 Another example of a flyout panel for configuring settings of an app.

The charms bar illustrated in Figure 1-13 provides Support Request and Privacy Policy commands, which are custom commands specific to the app currently in foreground. The Privacy Policy command navigates to the privacy page required for any app that consumes a remote service over the Internet, as you learned earlier in this chapter.

Windows Runtime

A Windows Store app is a software solution that adheres to the UI and technical specifications of the Windows Store. You can create a Windows Store app using any language that supports the new Windows Runtime (WinRT). WinRT is a rich set of application programming interfaces (APIs) built upon the Windows 8 operating system that provides direct and easy access to all the main primitives, devices, and capabilities from any language with which you can develop Windows 8 apps. WinRT is available only to Windows 8 apps. Its main purpose is to unify the development experience of building a Windows 8 app, regardless of the programming language you use to program that app.

For now, saying that you can use "any language supporting the Windows Runtime" means that you can choose to use C++, .NET (C# or VB), or JavaScript. Nevertheless, there are no technical limitations restricting use of WinRT from any other language, as long as it adheres to WinRT specifications. Chapter 5, "Introduction to the Windows Runtime," explains more about this topic as well as the architecture of WinRT.

At this point, you can think of WinRT as an infrastructural framework of libraries that simplify developing Windows Store apps by hiding the inner details of the operating system from the common and everyday developer perspective. For example, you can ask WinRT to open the webcam standard user interface to capture photos or videos without having to know anything about the underlying driver or Win32 API.

Here's a more complete example. The following code excerpt shows how easy and simple it is to capture a picture from your PC's camera using the C++ language.

```
void CaptureWin8::MainPage::TakePhoto_Click(Platform::Object^ sender,
          Windows::UI::Xaml::RoutedEventArgs^ e) {

    CameraCaptureUI^ dialog = ref new CameraCaptureUI();
    concurrency::task<StorageFile^> (
        dialog->CaptureFileAsync(CameraCaptureUIMode::Photo)).then([this] (
           StorageFile^ file) {
        if (nullptr != file) {
            concurrency::task<Streams::IRandomAccessStream^> (
            file->OpenAsync(FileAccessMode::Read)).then([this] (
                Streams::IRandomAccessStream^ stream) {
                    BitmapImage^ bitmapImage = ref new BitmapImage();
                    bitmapImage->SetSource(stream);
                    image->Source = bitmapImage;
            });
        }
    });
}
```

You could define the same action using JavaScript, as shown in the following code excerpt:

```
var dialog = new Windows.Media.Capture.CameraCaptureUI();
dialog.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo).done(
    function (file) {
    if (file) {
        var photoBlobUrl = URL.createObjectURL(file, { oneTimeOnly: true });
        document.getElementById("capturedPhoto").src = photoBlobUrl;
    }
};
```

Moreover, even using C# you can achieve the same result, as you can see in the following code excerpt:

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

    var camera = new CameraCaptureUI();
    var img = await camera.CaptureFileAsync(CameraCaptureUIMode.Photo);
    if (img != null) {
        var stream = await img.OpenAsync(FileAccessMode.Read);
        var bitmap = new BitmapImage();
        bitmap.SetSource(stream);
        image.Source = bitmap;
    }
}
```

Badges, live tiles, toasts, and lock screen

Another group of new features in Windows Store apps are badges, live tiles, toasts, and the lock screen. Badges and live tiles support showing dynamic information to end users, even when those users may not be directly using your app but are browsing through the Start screen. You can use a badge and/or a live tile to provide information about news, new items to check, new tasks to execute, or whatever else is meaningful and appropriate so users get a better experience with your app from the Start screen without opening the application. For example, the out-of-the-box Mail app uses the badge to show the number of unread mails in the inbox and a live tile to show a rotating list containing excerpts from all the unread emails. Moreover, the Windows Store app uses a badge to notify users about the number of available updates for their installed apps. Figure 1-14 shows some badges and live tiles in action.

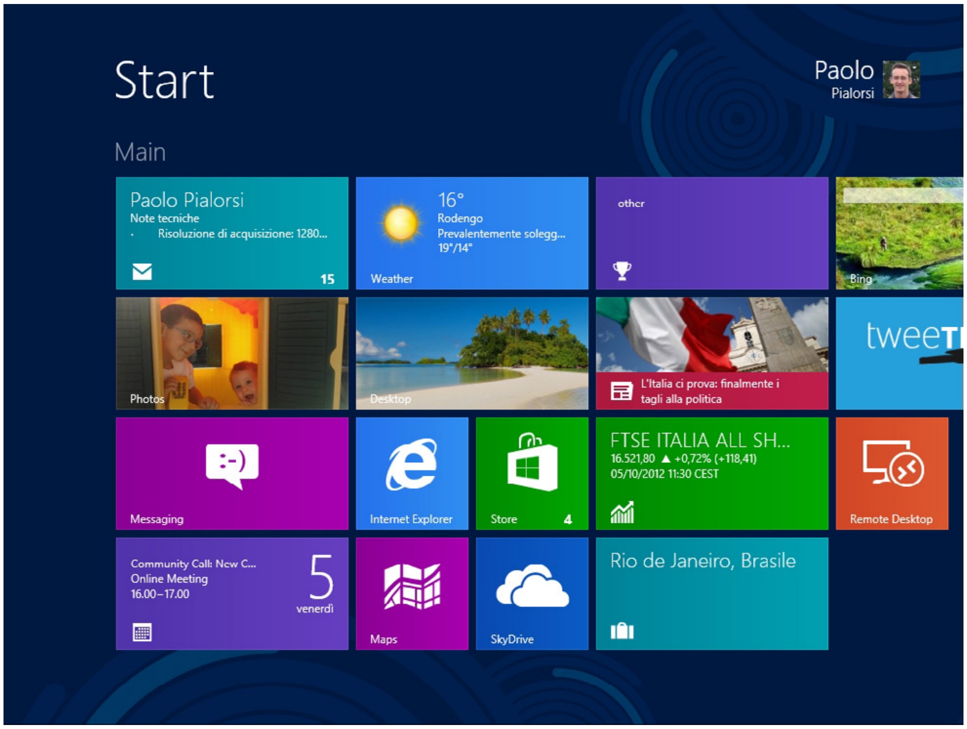


FIGURE 1-14 The Start screen with tiles showing badges and live tiles.

Notice the number 4 in the bottom-right corner of the Windows Store app—this is a badge indicating that there are four pending updates. You can also see the badge with number 15 in the bottom-right corner of the Mail app, notifying the user that there are 15 new emails in the inbox. Furthermore, the Mail app uses a live tile to show an excerpt of the most recent unread mails.

A live tile has even more functionality. For example, a live tile can completely change its content to remain dynamic and fresh, and pique the curiosity of the end user. Figure 1-15 shows four different states that can be assumed by the tile of a single app (the Bing Travel app, in this case).

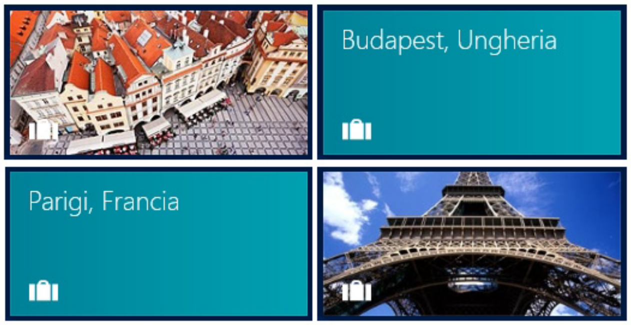


FIGURE 1-15 Some sample layouts for a live tile.

Official guidelines for Windows Store apps (see *http://msdn.microsoft.com/library/windows/apps/hh465403.aspx*) suggest using a wide tile only when you have live tiles to display. For apps that do not require a live tile, you should use the smaller square tiles. If you need to display only relatively static content for your tiles, you can simply use a badge to provide small and lightweight notifications. Chapter 9, "Rethinking the UI for Windows 8 apps," covers how to create a live tile.

Toasts are another technique for providing asynchronous alerts to an end user. For example, an alert/alarm application can ask the operating system to send to the user a toast at a predefined wake-up time. WinRT will send the toast even if the application that requested the toast is not active at that time.

Toasts are also important for notification purposes because when the user is working with an app in the foreground, background apps cannot interact with the user except through toasts. In fact, as you will learn in Chapter 4, due to the Windows 8 architecture and the application life-cycle management of Windows Store apps, only the foreground app has the focus and is running; all the other background apps can be suspended (or even terminated) by WinRT. A suspended app cannot execute or consume any CPU cycles. However, you can define a background task that will work in the background (more on this topic later in this chapter)—even in a separate process from the owner app—and you can define background actions. When these actions need to alert the user about their outcomes, they can use a toast.

A toast can be a simple text, or an image, or many combinations of the two. Figure 1-16 shows a toast in the upper-right corner of the screen provided by the Windows Store app, informing the user that an app installation task has completed in the background. Chapter 9 shows you how to create a toast for your own Windows 8 apps.

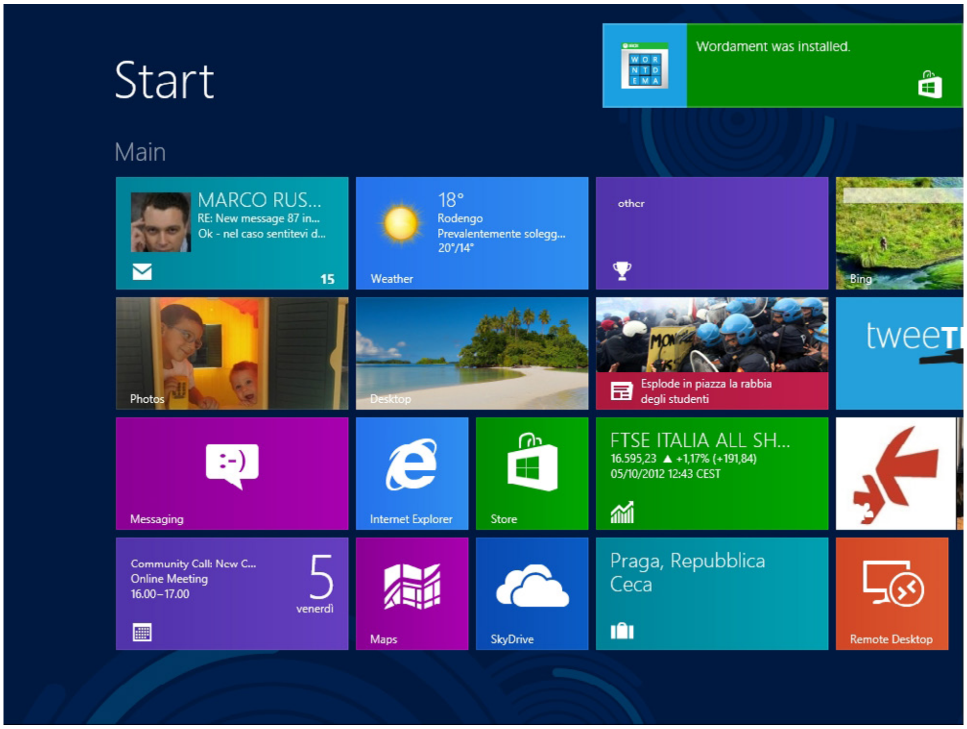


FIGURE 1-16 A toast rendered within a user session in Windows 8.

One last opportunity you have, while developing a Windows Store App, is to provide lightweight information to the end user through the lock screen. The lock screen is the screen that displays when a Windows 8 user session is locked out, which can occur after a period of inactivity or is displayed when the end user presses Windows+L to lock the session.

For example, in Figure 1-17, the lock screen provides some information about the current date and time, the next appointment in the user's agenda, and a set of small icons in the lower part of the screen. Those icons provide information about network connection status, battery status (for a device running on battery power), unread email in the inbox, and some other lightweight information.

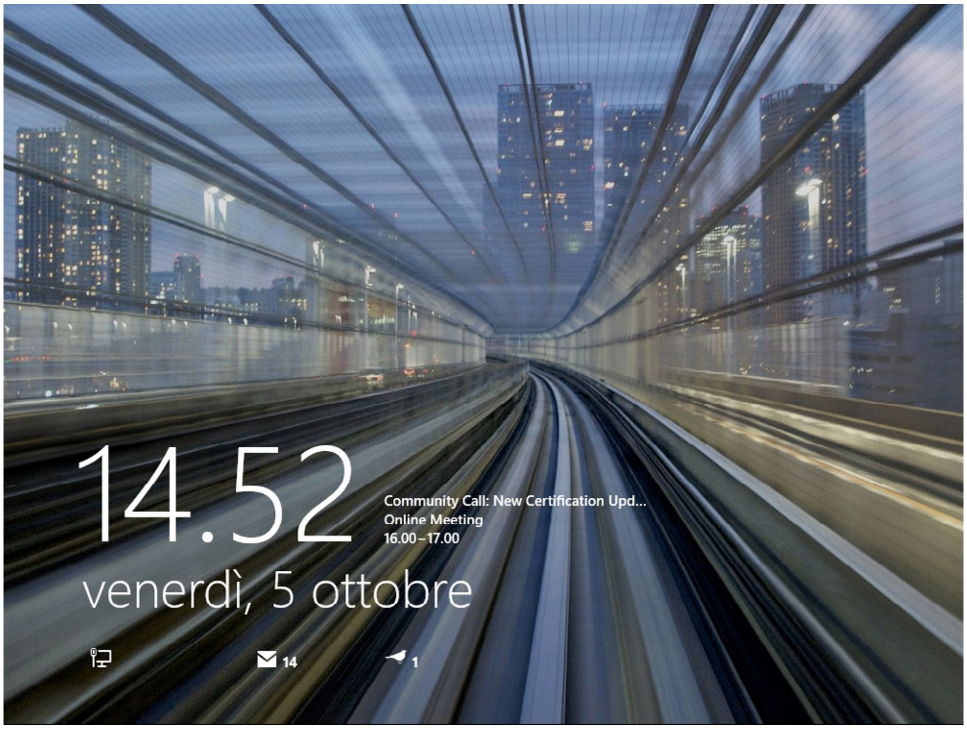
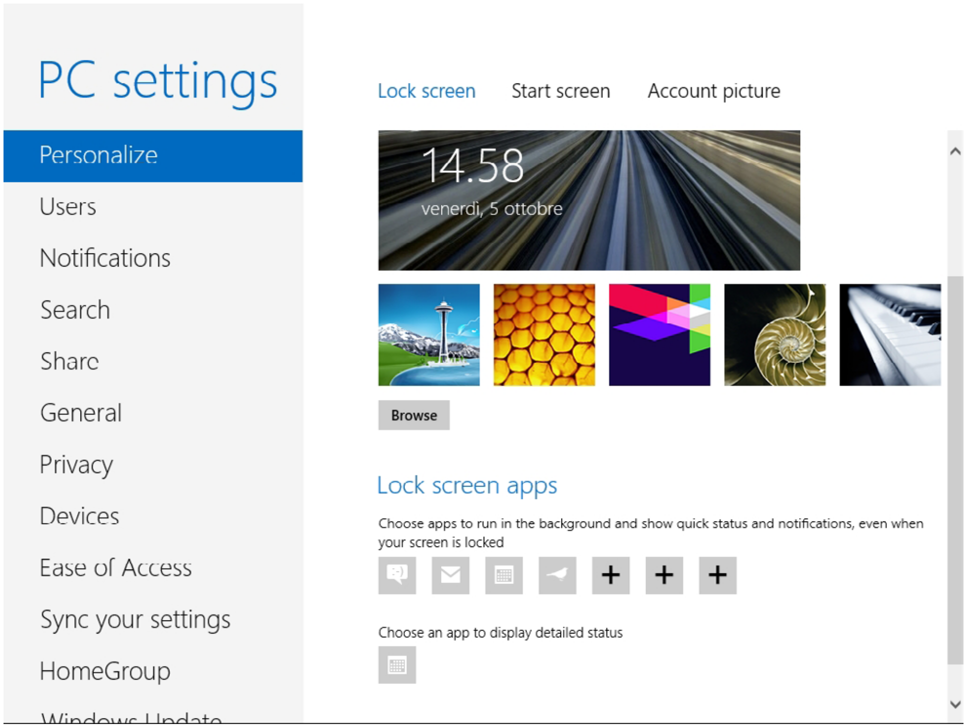


FIGURE 1-17 The Windows 8 lock screen.

An end user can choose what to see on the lock screen by using the proper panel in the system configuration. However, users may not display more than seven lock screen items at once that provide such detailed information. All seven apps will be able to show badges and toasts on the Start screen, but only one of those apps will be allowed to show the text of its latest tile notification on the lock screen.

Figure 1-18 shows the configuration panel for the lock screen. To reach it, you need to show the charms bar by, for example, pressing Windows+C and then selecting the Settings command. Finally, click the Change PC Settings command. Under the Personalize section on the Lock Screen tab, you will find the lock screen configuration settings.



Choose apps to run in the background and show quick status and notifications, even when your screen is locked

Choose an app to display detailed status

FIGURE 1-18 The PC settings for a Windows 8 system.

As you can see, the lock screen settings page enables users to choose the background image to display on the lock screen and to select which seven apps will execute in the background to provide information through the lock screen icons. Last but not least, users can select which app can display detailed text status. The last app, by default, is configured to be the Calendar app. To be available to function as a lock screen app, your software must declare that capability within an *app manifest file*, which will be explained later starting in Chapter 3, "My first Windows 8 app."

The information shown by a lock screen–enabled app is the same as the information that app shows on the Start screen. In fact, the text shown beside the small icon on the lock screen comes from the app's badge, while the detailed text status is taken from the app's tile text.

Background tasks

As stated earlier in this chapter—and as you will explore further in Chapter 4—a Windows Store app executes code only when it is in the foreground. However, there are situations where you want to be able to execute some code, even if your app is not the one currently in foreground, which necessarily means your app is in the background. To do that, you need to create a *background task*. A background task can execute code even when the corresponding app is suspended, but it runs in an environment that is both restricted and resource managed. Moreover, background tasks receive only a limited amount of system resources. Therefore, you should use background tasks only to execute small pieces of code that don't require any user interaction. For example, you should not use a background task to execute complex business logic or calculations, because the amount of system resources available to the background task is very tight and limited. In addition, complex background workloads consume battery power and CPU cycles, reducing the efficiency and responsiveness of the system.

To create a background task, you must define a class and register it with the operating system. A background task is just a class that implements a specific interface (*IBackgroundTask* in C#, for example) defined by WinRT. You register the task using a *BackgroundTaskBuilder* class instance. Many types of background tasks are available that respond to different kinds of triggers:

- ***ControlChannelTrigger*** Raised when there are incoming messages on the control channel
- ***MaintenanceTrigger*** Happens when it is time to execute system maintenance tasks
- ***PushNotificationTrigger*** Raised when a notification arrives on the Windows Notifications Service channel
- ***SystemEventTrigger*** Happens when a specific system event occurs
- ***TimeTrigger*** Triggered when a time event occurs

In particular, a *SystemTrigger* can occur in response to any of the following system events:

- ***InternetAvailable*** The Internet becomes available.
- ***LockScreenApplicationAdded*** An app tile is added to the lock screen.
- ***LockScreenApplicationRemoved*** An app tile is removed from the lock screen.
- ***ControlChannelReset*** A network channel is reset.
- ***NetworkStateChange*** A network change such as a change in cost or connectivity occurs.
- ***OnlineIdConnectedStateChange*** The online ID associated with the account changes.
- ***ServicingComplete*** The system has finished updating an application.
- ***SessionConnected*** The session is connected.
- ***SessionDisconnected*** The session is disconnected.
- ***SmsReceived*** A new SMS message is received by an installed mobile broadband device.
- ***TimeZoneChange*** The time zone changes on the device (for example, when the system adjusts the clock for daylight saving time).
- ***UserAway*** The user becomes absent.
- ***UserPresent*** The user becomes present.

Whenever such an event occurs, you can check a set of conditions to determine whether or not your background task should execute. The conditions you can check are as follows:

- ***InternetAvailable*** The Internet must be available.
- ***InternetNotAvailable*** The Internet must be unavailable.
- ***SessionConnected*** The session must be connected.
- ***SessionDisconnected*** The session must be disconnected.
- ***UserNotPresent*** The user must be away.
- ***UserPresent*** The user must be present.

To optimize resource consumption, some triggers fire only to apps in the lock screen. For example, a *TimeTrigger* can be leveraged only by an app in the lock screen. The same requirement is valid for *PushNotificationTrigger* and *ControlChannelTrigger*. Even some of the *SystemTrigger* events are reserved for apps in the lock screen, such as *SessionConnected*, *UserPresent*, *UserAway*, and *ControlChannelReset*. Because your app should register for these events and triggers only when it's in the lock screen, the *SystemTrigger* events *LockScreenApplicationAdded* and *LockScreenApplicationRemoved* are provided, which let an app register and unregister such triggers appropriately.

Generally speaking, common language runtime (CLR) and C++ apps can execute a background task in the app itself or in a system-provided host (BackgroundTaskHost.exe). Moreover, tasks for triggers of type *PushNotificationTrigger* or *ControlChannelTrigger* can also execute in the app process.

To properly introduce the background tasks, one last topic to cover is resources management. Every background task must execute its code using a constrained amount of CPU and network bandwidth. For example, each app on the lock screen receives 2 seconds of CPU time every 15 minutes, plus 2 more seconds for executing background tasks, just after the previous 2 seconds. In comparison, an app that is not on the lock screen receives 1 second of CPU time every 2 hours.

From a network bandwidth perspective, these constraints are a function of the amount of energy consumed by the network interface. For example, with a throughput of 10 megabits, an app on the lock screen can consume about 450 MB per day, while an app that is not on the lock screen can consume about 75 MB per day.

So the purpose of these constraints is to reduce battery and resource consumption. These rules do not apply for apps that rely on critical background tasks such as *ControlChannelTrigger* and *PushNotificationTrigger*—those kinds of tasks receive guaranteed resources. Finally, there is a global pool of resources (CPU and network) that is shared across apps, and that can be used to provide extra resources to those apps that need them. Of course, an app should not rely on the availability of such resources, because they are shared between all background tasks of any app, so another app could already have consumed them all. The global pool is refilled every 15 minutes, using a refill quota related to whether the device is running on an AC adapter or on battery power.

Contracts and extensions

A powerful set of features available for developing Windows Store apps are called WinRT contracts. WinRT and Windows Store apps can share data, information, features, and behaviors through shared communication contracts. A *contract* is an agreement between an app and the Windows 8 operating system by which an app that follows specific rules can communicate and exchange data with any other app—without directly knowing about the other app—using the operating system and WinRT as a proxy.

For example, start the Bing Travel app from the Start screen and navigate to a target location for a journey, such as Rome in Italy. Then show the charms bar (press Windows+C) and select the Share command. You will be prompted by a panel within the charms bar asking you to decide whether you want to share that location by email, with friends using the People app, or via any other Windows Store app configured as a sharing target for the type of content you want to share. You can see the result in Figure 1-19.

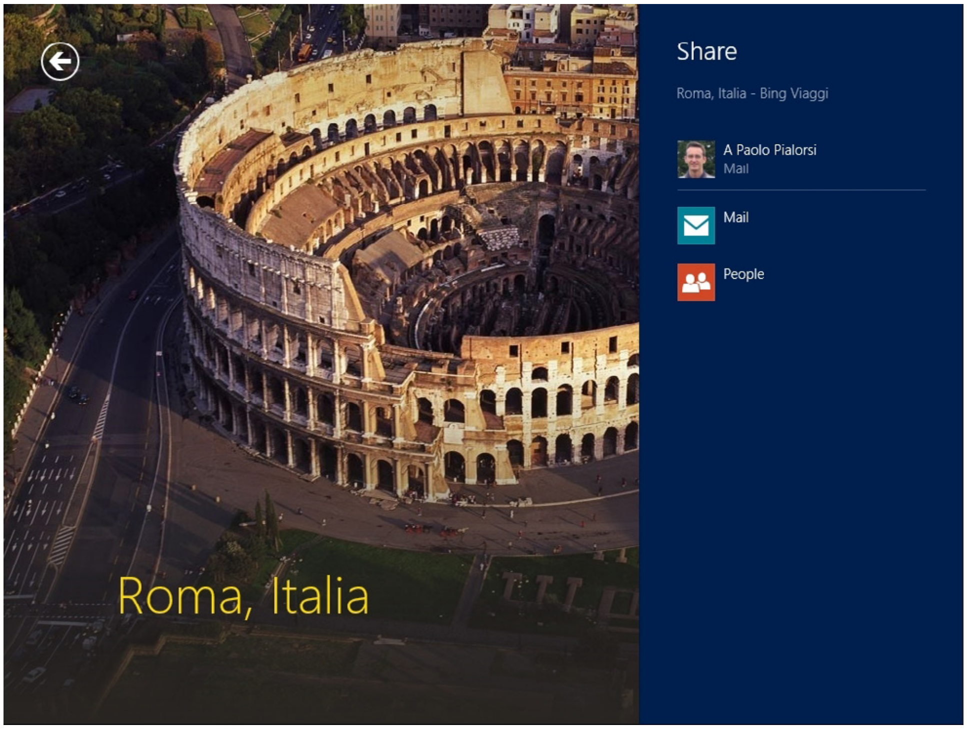
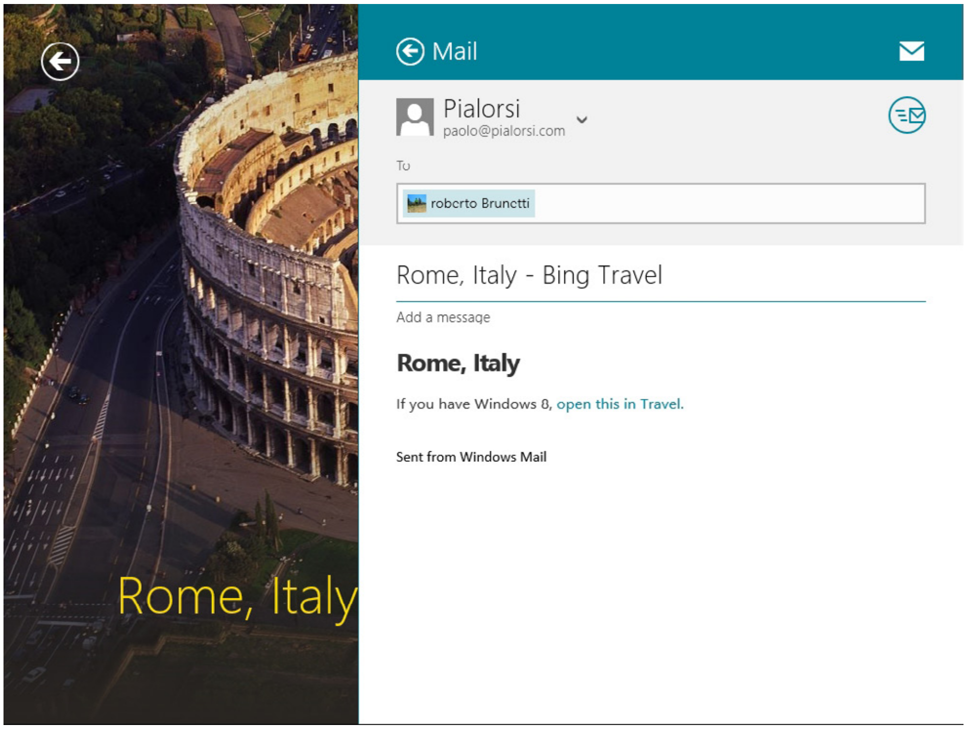


FIGURE 1-19 The share content flyout panel within the Bing Travel app.

As soon as you have made a choice, for example by selecting Mail, Windows launches the sharing target app behind the scenes, so it can handle the shared content. In Windows Mail for example, you can send the information about Rome to someone else via email (see Figure 1-20).



If you have Windows 8, open this in Travel.

Sent from Windows Mail

FIGURE 1-20 The UI exposed by the Mail app while sharing some content with it.

In reality, neither app (Bing Travel nor Windows Mail) is aware of the other. WinRT, sitting in the middle, joins them through a contract called a *Share* contract.

These features are shared by all apps, not just custom apps. For example, when you are using the Windows Store app, and you activate the search feature (Windows+Q), the operating system uses a Search contract to query the Windows Store app for apps that satisfy the search criteria provided.

WinRT includes a rich set of contracts, as shown in the following list:

- **Cached File Updater** You can leverage this contract to keep track of files changes and cache them. For example, the SkyDrive app uses this contract to monitor file changes.
- **File Picker** This contract enables you to register your app as a target for the file picker UI.
- **Play To** This contract allows your app to be listed in the list of apps available in the Play To section of the Connect command in the charms bar.
- **Search** This contract provides search capabilities to your app.
- **Settings** This contract supports providing a panel where users can enter custom settings for your app.
- **Share** This contract supports sharing content between apps.

In addition to contracts, there are also *extensions*, which allow an app to adhere to an agreement with the operating system rather than a third-party app. You can use an extension to extend Windows standard features. For the sake of simplicity, consider what happens when you connect a device or insert a disc into the CD/DVD reader. An operating system message informs the end user that he or she can execute/play the new device or media, providing a list of available actions and players. You can register your app as supporting the AutoPlay extension, and your app will then appear in the list of available AutoPlay targets.

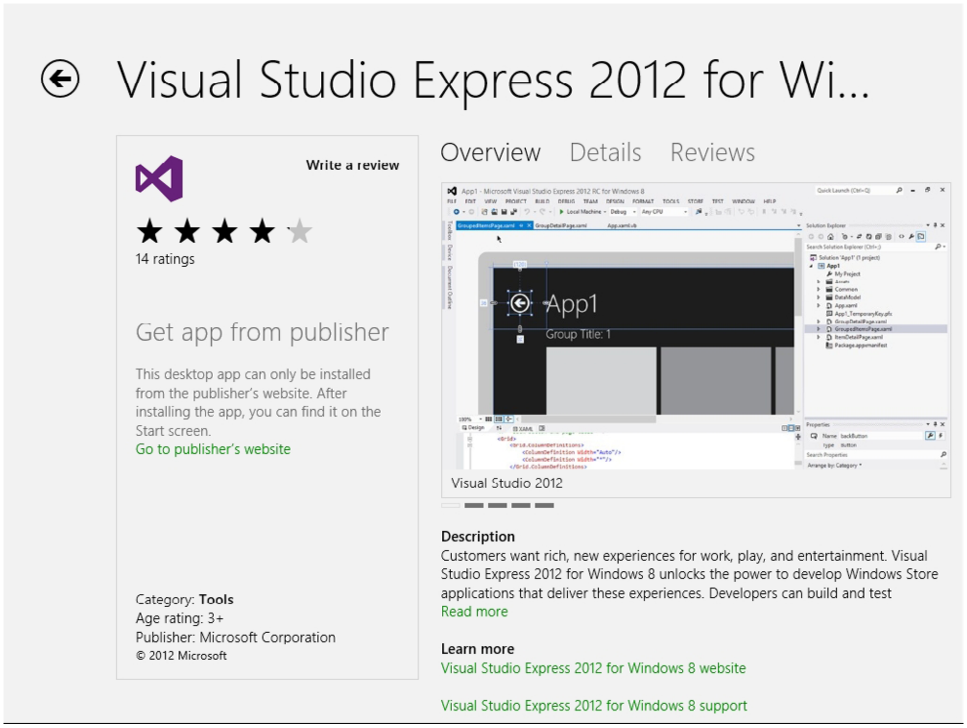
The following is a list of available extensions:

- **Account picture provider** When an end user changes his or her own account picture, you can register your app as an account picture provider.
- **AutoPlay** This extension enables your app to be included in the list of AutoPlay targets.
- **Background tasks** The app can run background tasks.
- **Camera settings** You can provide custom UI for camera settings.
- **Contact picker** You can register your app as contact picker provider.
- **File activation** This extension enables you to register an app to execute a specific file type based on the file extension.
- **Game Explorer** You can register you app as a game, providing a Game Definition File (GDF), and your app will be available as a game only if compliant with the target family safety rules.
- **Print task settings** You can declare that your app can have a custom printer UI and can print by communicating directly with a printer device.
- **Protocol activation** This extension allows you to register a protocol moniker for your app. For example, Windows Mail can be activated with a mailto: protocol moniker. Internet Explorer 10 can be activated with an http: protocol moniker. You can register your own moniker and use it to activate your app.
- **SSL/certificates** This extension enables your app to install a digital certificate onto the target device.

As you will learn in Chapter 3, it is simple to register or consume a contract through WinRT.

Visual Studio 2012 and Windows 8 Simulator

To develop a Windows Store app, you will first need to install a development environment such as Microsoft Visual Studio 2012. To accomplish this task, you can buy and install a regular license for Visual Studio 2012 either directly from Microsoft or from an authorized reseller. However, for evaluation purposes, you can get started with a free edition of Visual Studio 2012, called Visual Studio Express 2012. In particular, one edition of the Visual Studio Express family of products is called Visual Studio Express 2012 for Windows 8. Using this development tool, you can create Windows Store apps either from scratch or by starting with a set of prebuilt application templates and models. You can download Visual Studio Express 2012 for Windows 8 from the Microsoft website at *http://www.microsoft.com/visualstudio/*. Alternatively, you can acquire it through the Windows Store, where it's listed under the Tools category of apps. Figure 1-21 shows the page dedicated to Visual Studio Express 2012 for Windows 8 in the Windows Store.



Visual Studio Express 2012 for Wi...

Write a review

14 ratings

Get app from publisher

This desktop app can only be installed from the publisher's website. After installing the app, you can find it on the Start screen.
Go to publisher's website

Category: **Tools**
Age rating: 3+
Publisher: Microsoft Corporation
© 2012 Microsoft

Overview Details Reviews

Visual Studio 2012

Description
Customers want rich, new experiences for work, play, and entertainment. Visual Studio Express 2012 for Windows 8 unlocks the power to develop Windows Store applications that deliver these experiences. Developers can build and test
Read more

Learn more
Visual Studio Express 2012 for Windows 8 website

Visual Studio Express 2012 for Windows 8 support

FIGURE 1-21 The Visual Studio Express 2012 desktop application within the Windows Store.

Perhaps even better, you can download a 90-day evaluation copy of Visual Studio 2012 from *http://www.microsoft.com/visualstudio/eng/downloads*. This 90-day trial version should provide sufficient time for you to complete this book and experience all the exercises and demos using a complete version of the product.

After installing Visual Studio, you will be able to create custom apps and publish them in the Windows Store. Chapter 3 and Chapter 4 discuss how to accomplish these tasks in more detail.

Another possible development track to consider is that you can download and install a retail version of Visual Studio 2012 (Professional, Premium, or Ultimate), even on previous editions of Windows. For example, perhaps you still don't have a Windows 8 PC; instead, you're using a Windows 7 desktop machine. You can still install Visual Studio 2012 and develop software solutions. However, you will not be able to develop Windows Store apps. Moreover, you cannot download and install Visual Studio Express 2012 for Windows 8 on a computer without Windows 8, because that edition requires you to have Windows 8 or later.

A final option for testing and executing your apps is to use the Windows 8 Simulator, which is part of the Windows 8 SDK included with Visual Studio 2012. Figure 1-22 shows the Windows 8 Simulator in action.

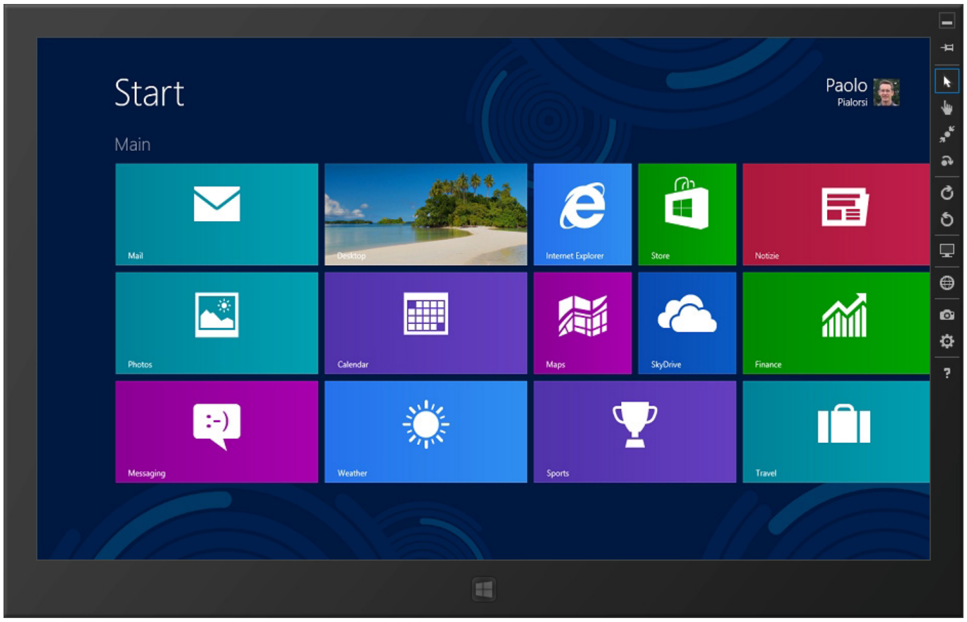


FIGURE 1-22 The Windows 8 Simulator in action.

As you can see from Figure 1-22, the Simulator looks like a small tablet PC with Windows 8 on board. On the right side, the Simulator includes a set of commands through which you can simulate all the various scenarios for Windows 8. These commands are as follows, from top to bottom:

- **Always on top** This command puts the Simulator always on top.
- **Mouse mode** When you move and click your mouse, the Simulator will react to mouse interactions as well.
- **Basic touch mode** Your mouse pointer will become like a finger, and when you click the Simulator, it will be handled as a finger touch.
- **Pinch/zoom touch mode** This command is similar to the previous option, but you use it to simulate zoom-in and zoom-out via touch gestures.
- **Rotation touch mode** This command is similar to the previous option, but you use it to simulate touch rotation gestures.
- **Rotate clockwise (90 degrees)** This command rotates the device clockwise 90 degrees.
- **Rotate counterclockwise (90 degrees)** This command rotates the device counterclock-wise 90 degrees.

- **Change resolution** This command changes the screen resolution of the simulator device. The available resolutions are as follows:

  10.6", 1024 × 768

  10.6", 1366 × 768

  10.6", 1920 × 1080

  10.6", 2560 × 1440

  12", 1280 × 800

  23", 1920 × 1080

  27", 2560 × 1440
- **Set location** This command allows you to simulate a GPS location, for testing location-based apps.
- **Copy screenshot** Use this command to create a screenshot of the Simulator screen, which is useful for creating promotional pictures of your apps and is required to publish a real app on the Windows Store.
- **Screenshot settings** This command configures the copy screenshot behavior, such as the destination directory of the image files.
- **Help** This command provides a link to the Simulator's help.

Using the Simulator, you can fully test your apps, even without a physical tablet device or touch screen, and without a Windows 8 environment.

One of the most important features of the Simulator is the ability to change the resolution, orientation, and form factor of the screen so you can test your application's behavior in many different "devices" without the need to buy real ones.

Last but not least, remember that you cannot develop a Windows Store app using Microsoft Visual Studio 2010 or any other earlier edition of the product. The only edition of Visual Studio suitable for developing Windows Store apps is Visual Studio 2012 or later.

Summary

This chapter presented an overview of Microsoft Windows 8 and Windows Store apps. You learned about the key new features of Windows 8 as well as the main goals behind the development of a Windows Store app. You learned about apps, the Windows Store, badges, live tiles, toasts, background tasks, the new lock screen, the new Start screen, and more. The chapter also provided information about which development environments you can use to develop Windows Store apps.

Quick reference

To	Do this
Notify a user of an action that happened in the background	Use a toast, a badge, or a live tile. You can also use the lock screen, in case it is suitable for your context.
Execute some code while your app is suspended	Use a background task.
Make the contents managed by your app searchable by the end user	Support the Search contract.
Develop a Windows Store app	Install Microsoft Visual Studio Express 2012 for Windows 8 or Microsoft Visual Studio 2012 on a Windows 8 device.
Simulate the execution of a Windows 8 app in different resolutions, orientations, and form factors	Run the Windows 8 Simulator available within Visual Studio 2012.

CHAPTER 4

Application life-cycle management

After completing this chapter, you will be able to

- Understand the application manifest settings.
- Use the application manifest to modify application capability and appearance.
- Deploy and test an application.
- Understand the way Windows 8 manages the different running states of an application.
- Respond to launching, activating, suspending, and resuming events.
- Use the application data store to save data locally.

In preceding chapters, you saw how Windows 8 provides a new user interface, offers a completely new user experience, and exposes a new set of APIs called Windows Runtime (WinRT) APIs through which you can interact with the operating system. You also developed a simple application in Chapter 3, "My first Windows 8 app."

This chapter introduces the complete application life cycle in Microsoft Windows 8, from deployment, to launch, to uninstallation. You will start by analyzing the various settings in the application manifest. The *application manifest* is a file that defines an application's appearance on the Start screen and informs Windows 8 about which WinRT features the application will use. You will also explore how WinRT manages an application's life cycle at run time, by launching, suspending, resuming, and terminating the application as needed.

First, a Windows 8 application cannot include an app.config file. This means that—just as in a Microsoft Windows Phone or Windows Presentation Foundation (WPF) Web Browser Application (WBA)—you cannot use the classic .NET configuration mechanism to provide application and system settings. There is no *System.Configuration* namespace or any equivalent classes in the WinRT APIs. The Windows 8 runtime system executes Windows Store applications in a sandboxed process, similar to a Silverlight or WPF WBA. Users cannot navigate to the file system where the application is installed and change files, because Windows 8 apps are mainly downloaded and installed from the Windows Store.

Application manifest

As in a Windows Phone 7.*x* project, deployment information and many configuration settings are stored in a manifest file that WinRT calls Package.appxmanifest, which is an XML-formatted file that describes various aspects of the project, as you can see in the following listing, taken from a real Windows Store application.

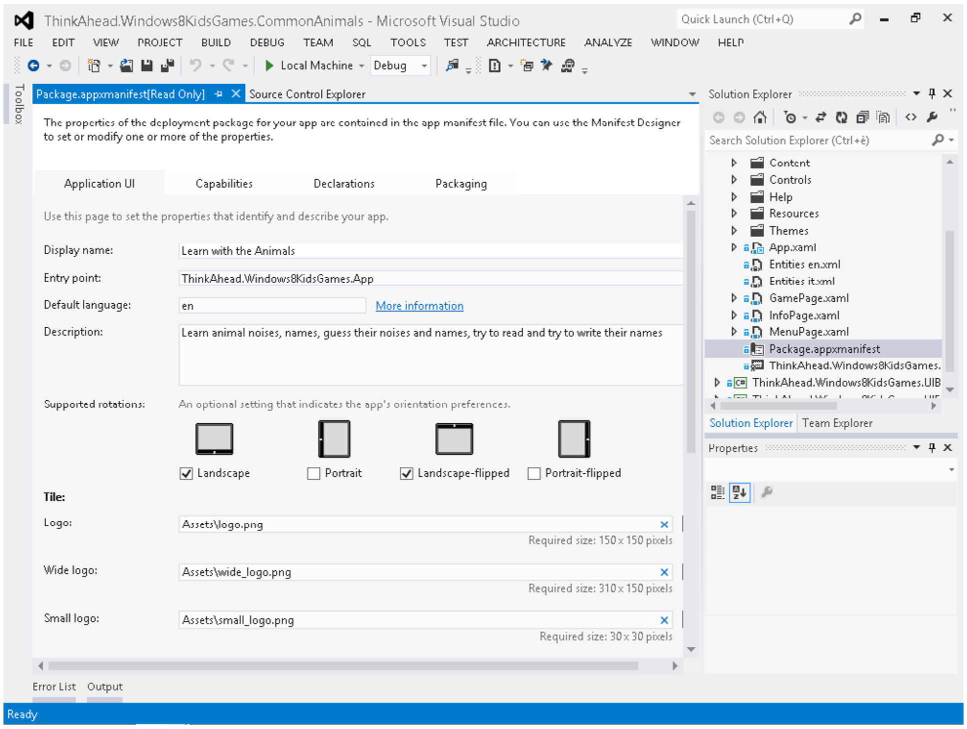
```
<?xml version="1.0" encoding="utf-8"?>

<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">

 <Identity Name="ea15f786-9bb0-4d64-98b0-d251fa375633" Publisher="CN=Devleap"
    Version="1.0.0.1" />

 <Properties>
    <DisplayName>Learn with the Animals</DisplayName>
    <PublisherDisplayName>ThinkAhead</PublisherDisplayName>
    <Logo>Assets\Store_Logo.png</Logo>
 </Properties>
 <Prerequisites>
    <OSMinVersion>6.2.1</OSMinVersion>
    <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
  </Prerequisites>
  <Resources>
    <Resource Language="x-generate" />
  </Resources>
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe"
       EntryPoint="ThinkAhead.Windows8KidsGames.App">
      <VisualElements DisplayName="Learn with the Animals" Logo="Assets\logo.png"
         SmallLogo="Assets\small_logo.png" Description="Learn animal noises, names,
         guess their noises and names, try to read and try to write their names"
         ForegroundText="dark"
         BackgroundColor="#464646">
        <DefaultTile ShowName="noLogos" WideLogo="Assets\wide_logo.png"
           ShortName="Learn with the Animals" />
        <SplashScreen Image="Assets\splash_screen.png" BackgroundColor="#b4dfba" />
        <InitialRotationPreference>
          <Rotation Preference="landscape" />
          <Rotation Preference="landscapeFlipped" />
        </InitialRotationPreference>
      </VisualElements>
    </Application>
  </Applications>
</Package>
```

The first section, called *Properties*, contains information for the Windows Store, such as the title for the application, the name of the publisher, the official logo, and a brief description.

The last section, called *Capabilities*, contains a list of all the operating system features the application needs to access on the user's PC or tablet. When application code requests one of these features, the user receives a direct request to give the application specific permission to use the feature. The user can revoke this permission at any time; your code has to fail gracefully if the user denies the permission to use a capability.

This way of working is similar to a Windows Phone 7.*x* project, where WMAppManifest.xml tells the operating system the capabilities the application requires to run. You can find more information on application capabilities in Chapter 6, "Windows Runtime APIs."

Figure 4-1 shows the Manifest Designer that Microsoft Visual Studio 2012 provides to simplify the application definition. To open the designer, simply double-click the Package.appxmanifest file in Solution Explorer. The figure shows the real manifest for one of the authors' applications, called Learn with the Animals. The Application UI tab lets you choose the display name of the application (the name used for the Start screen), a description of the application, three logos for the application, and so on.



The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or modify one or more of the properties.

Use this page to set the properties that identify and describe your app.

Display name: Learn with the Animals

Entry point: ThinkAhead.Windows8KidsGames.App

Default language: en More information

Description: Learn animal noises, names, guess their noises and names, try to read and try to write their names

Supported rotations: An optional setting that indicates the app's orientation preferences.

✓ Landscape ☐ Portrait ✓ Landscape-flipped ☐ Portrait-flipped

Tile:

Logo: Assets\logo.png
Required size: 150 x 150 pixels

Wide logo: Assets\wide_logo.png
Required size: 310 x 150 pixels

Small logo: Assets\small_logo.png
Required size: 30 x 30 pixels

FIGURE 4-1 The Application UI tab.

The first tab of the Visual Studio Manifest Designer produces the following section in the application manifest:

```
<VisualElements DisplayName="Save The Planet" Logo="Assets\Logo.png"
     SmallLogo="Assets\SmallLogo.png" Description="ThinkAhead.SaveThePlanet.Win8.UI"
     ForegroundText="light" BackgroundColor="#222222" ToastCapable="true">
        <LockScreen Notification="badgeAndTileText" BadgeLogo="Assets\BadgeLogo.png" />
        <DefaultTile ShowName="allLogos"  />
        <SplashScreen Image="Assets\SplashScreen.png" BackgroundColor="#000000" />
</VisualElements>
```

VisualElements, as the name implies, defines the display name for the Windows 8 Start screen, the various logos for the tile (*Logo*), the small tile (*SmallLogo*), and the wide tile (*WideLogo*), as well as the supported rotation and the default one, the badge default logo, and the image for the splash screen.

All the required images referenced by the application package manifest are provided as placeholders by the Visual Studio templates for Windows Store applications and are placed in the Assets folder of the project. The default template uses an image for the application logo that displays on the default application tile (Logo.png), an image for the initial splash screen (SplashScreen.png), a small logo image that is shown in the tile if the application switches its tile from code (SmallLogo.png) and, last but not least, an image used by the Windows Store to represent the application (StoreLogo.png). As you can see in Figure 4-1, you can also provide a wide logo that displays if the user chooses a wide tile for the application from the Start screen.

Figure 4-2 shows the application tile in the Windows 8 Start screen. The tile presents the image described in the application manifest as the *WideLogo* property and, as you will learn in Chapter 9, "Rethinking the UI for Windows 8 apps," the application can also modify the tile from code or create a secondary tile.

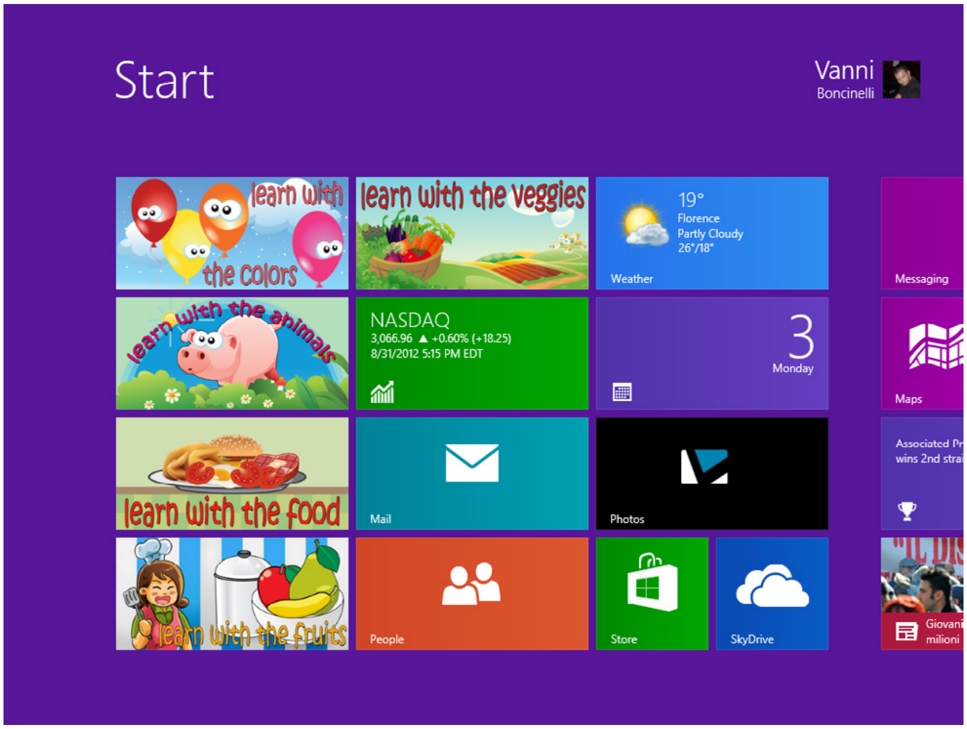


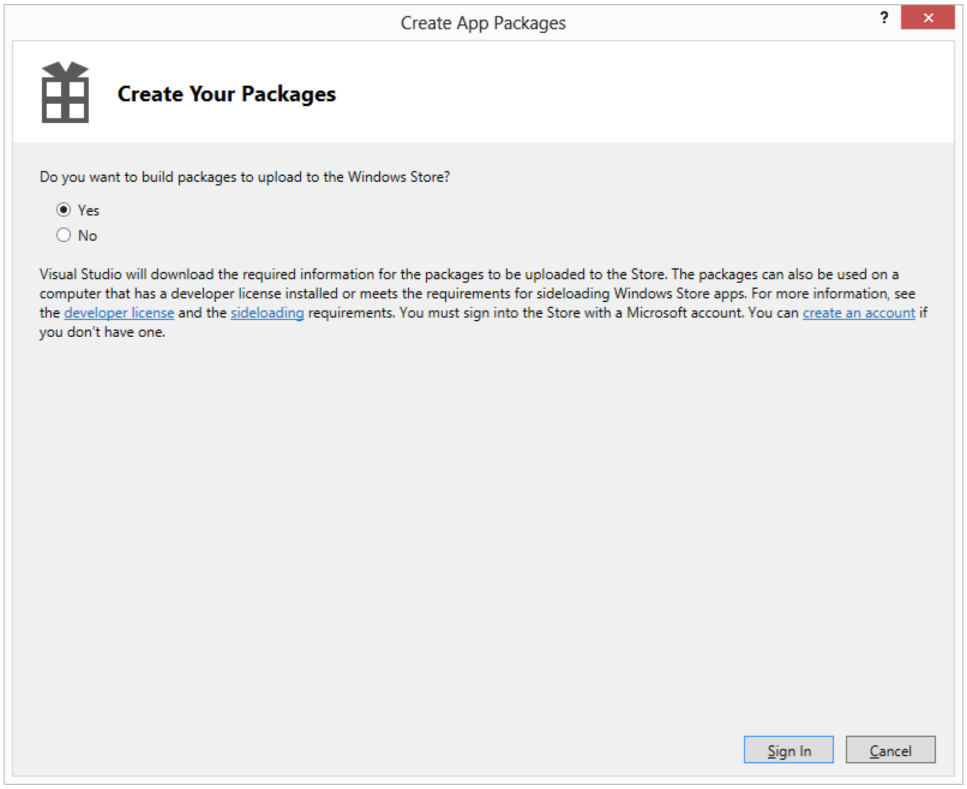
FIGURE 4-2 The Learn with the Animals wide tile on the Start screen.

Application package

The application manifest contains all the application information the system needs to deploy it on a target machine. That could be the local machine or the Windows 8 Simulator, which you can use for testing and debugging purposes. The manifest also contains all the information needed to package the application for the Windows Store.

When you run an application using the F5 button, Visual Studio 2012 compiles the application, builds the application package, and asks the operating system to install the package on the developer machine or the Windows 8 Simulator.

Visual Studio lets you package and deploy the application on the Windows Store by using the Store | Create App Package feature. This menu item launches a Create App Packages wizard that guides you through the process to package the application and upload it to the store, or to simply build the package to use it on a developer machine, as you can see from the two options and associated descriptions in Figure 4-3.



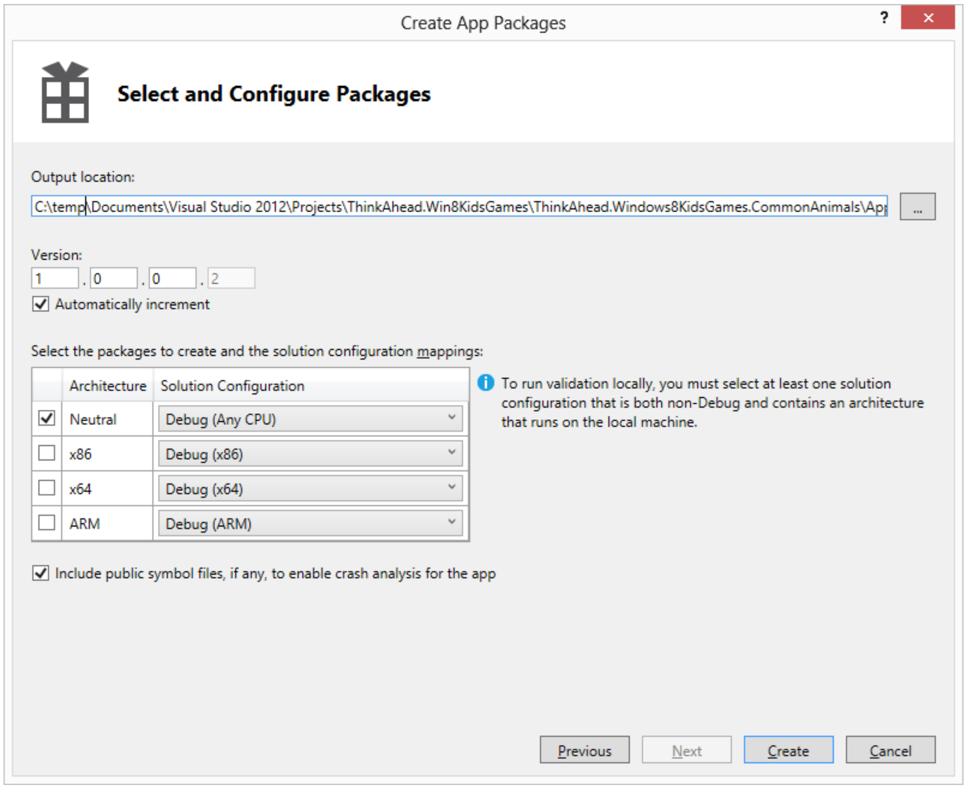
Do you want to build packages to upload to the Windows Store?

[✓ Yes] No

Visual Studio will download the required information for the packages to be uploaded to the Store. The packages can also be used on a computer that has a developer license installed or meets the requirements for sideloading Windows Store apps. For more information, see the developer license and the sideloading requirements. You must sign into the Store with a Microsoft account. You can create an account if you don't have one.

FIGURE 4-3 The Create App Packages wizard.

If you choose the first option, you will be asked for the Windows Live ID associated with your Windows Store account to publish the application. In both cases, the last step of the wizard lets you choose the processor architecture for which to build the application, and then it creates the package (see Figure 4-4).


Create App Packages

Select and Configure Packages

Output location:

C:\temp\Documents\Visual Studio 2012\Projects\ThinkAhead.Win8KidsGames\ThinkAhead.Windows8KidsGames.CommonAnimals\Ap

Version: 1 . 0 . 0 . 2

☑ Automatically increment

Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
☑	Neutral	Debug (Any CPU)
☐	x86	Debug (x86)
☐	x64	Debug (x64)
☐	ARM	Debug (ARM)

To run validation locally, you must select at least one solution configuration that is both non-Debug and contains an architecture that runs on the local machine.

☑ Include public symbol files, if any, to enable crash analysis for the app

FIGURE 4-4 The Create App Packages wizard lets you choose the output location, version, and configuration.

The package contains one binary file that represents the application and a folder with four different files:

- ***<App Name_Version_Compilation>*.appxupload** This is the real "package," and it contains the compiled application to be installed. For example, the application Learn with the Animals, version 1.0.0.6 for Any CPU is packaged in a file called LearnwiththeAnimals_1.0.0.6_AnyCPU.appxupload. This is the file for the Windows Store.
- ***<App Name_Version_Compilation>*.cer** This is a certificate used to sign the application in the local development environment. The private key is contained in the .pfx file of the Visual Studio 2012 project. During the installation process, this certificate is added to the Trusted Root Certification Authorities of the local machine.

- ***<App Name_Version_Compilation>*.appxsym** This file contains the debugging symbols.
- **Add-AppxDevPakage.bat** This file contains the script to install the application, the signing certificate in the Trusted Root Certification Authorities, and all the dependencies the application needs to run.

Note The directory name is also dependent on and formed by the compilation type (Debug, Release, Platform) and the current user name, and it contains all the deployed application files.

When the application is installed on the system, Windows 8 creates a directory in X:\Users\<*username*>\AppData\Local\Packages\ using the globally unique identifier (GUID) associated with the application. This GUID is automatically generated when the Create App Packages wizard creates a new Windows Store application and is stored in the application manifest in the *Identity* tag, as shown in the following excerpt:

```
<Identity Name="380ac04e-991e-4e5f-8758-5f56e68b0e94" Publisher="CN=DevLeap"
    Version="1.0.0.2" />
```

You can uninstall an application at any time by selecting its tile and choosing Uninstall from the Windows 8 Start screen. As shown in Figure 4-5, you can also unpin the application by choosing the Unpin from Start item in the app bar. Doing so simply removes the tile from the Start screen and does *not* uninstall the application from the system. You can always reach the application again by pressing Windows+Q and searching within the installed apps.

Note You can use the batch file to install the application on a developer machine manually.

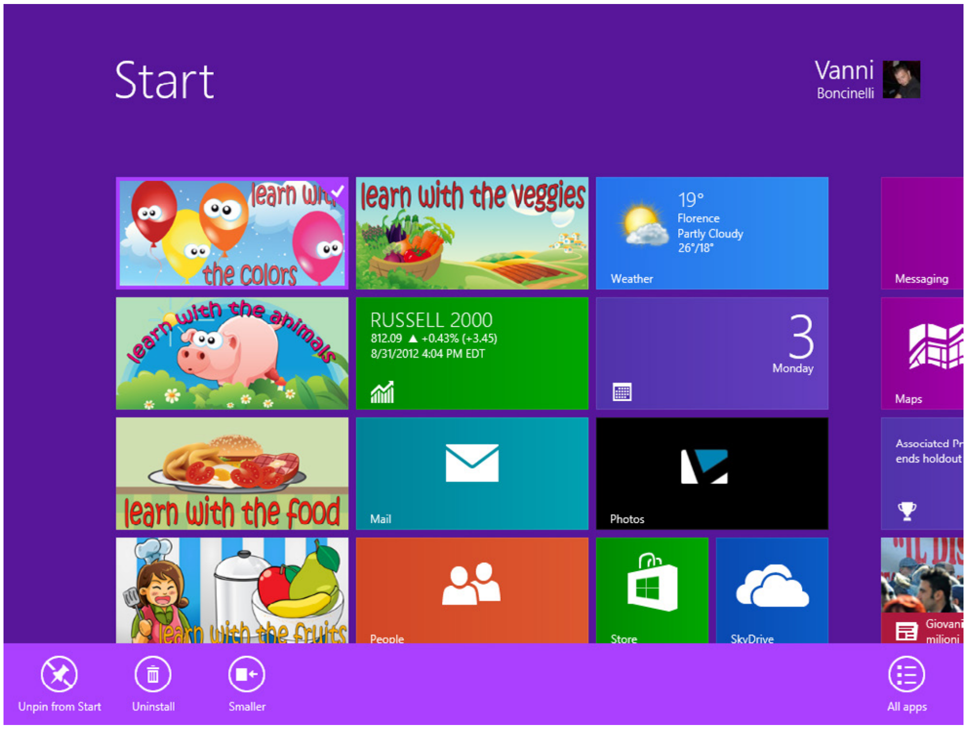


FIGURE 4-5 The Windows 8 Start screen app bar.

Windows Store

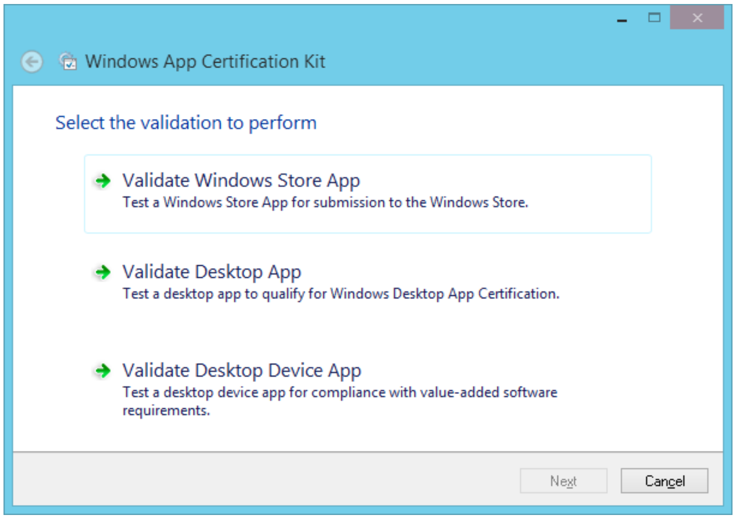
The Windows Store enables users to search for, download, install, and review applications. As a developer, you can upload your applications to the Windows Store, making them available for download or purchase on every tablet and PC running Windows 8 around the globe.

To upload an application, you first need to create a Windows Store account and bind it to a Windows Live ID. This procedure is quite straightforward. It also lets you define the application's publisher name, the name that appears in many Windows Store screens near the application name. Users can search for apps in the Windows Store by name, by keyword, and by publisher.

After you have created an account, you can upload an application immediately. Alternatively, you can reserve a name for an application you plan to develop within a year. If you plan to sell the application, you must also fill out the fiscal profile for the person or company specified as the publisher. You will also need to complete the IRS module related to your fiscal position. For example, if you are the publisher and you live outside the United States, you will need to fill out the W-8BEN form. Fortunately, a wizard will guide you during the process of choosing the right module and filling it out online.

You can upload and sell an application before you have filled in all the fiscal data, but you will receive no money until you have completed the financial profile.

Aside from these administrative tasks, the process of publishing an application is straightforward. First, you should verify that the application conforms to Windows Store requirements locally. This step is not required, but it's very useful to validate your application quickly before performing any upload. You can validate your app with the application verifier (Windows App Certification Kit, or ACK) which, as shown in Figure 4-6, validates an application for technical compliance with Windows Store rules.


Windows App Certification Kit

Select the validation to perform

Validate Windows Store App
Test a Windows Store App for submission to the Windows Store.

Validate Desktop App
Test a desktop app to qualify for Windows Desktop App Certification.

Validate Desktop Device App
Test a desktop device app for compliance with value-added software requirements.

FIGURE 4-6 The Windows App Certification Kit verifies that an application conforms to Windows Store rules.

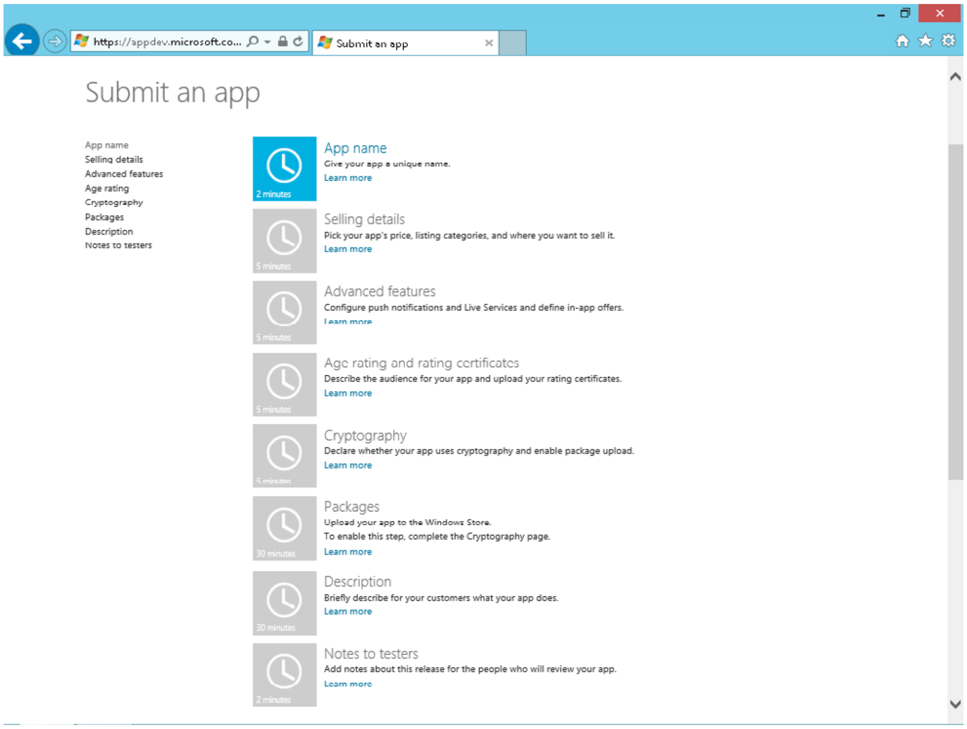
The tool can also validate a desktop application and a desktop device application for Windows 8 Desktop App Certification. The Windows App Certification Kit is installed on your system together with Visual Studio Express for Windows 8; you can launch it from the Start screen.

The next step lets you choose the application to validate (remember to deploy it to the local system compiling the project in release mode). The validation begins by launching the application. It is very important that you *do not interact* with the application (and the system) during the test, because the test also verifies how the application is suspended and how it resumes, as well as whether it closes and terminates correctly.

If your application does not pass all the tests performed by the Windows App Certification Kit, there's no point in trying to upload the application package to the Windows Store. The Windows Store service performs exactly the same verification process, so there is no way an app that fails local certification can pass the store certification. At the end of the process, you will receive detailed information on any application problems, presented as errors or warnings. As stated earlier, the local verification step is not required, but it is very useful and recommended.

When you have completed local certification, if you haven't reserved a name for your app, you must choose an application name before uploading the package.

For each application you upload to the Windows Store, you need to provide required information, including the application name and sales details (price, country availability, trial version availability, and so on). You can enrich this information by providing details such as an age rating (especially if your app is a game), the cryptography mechanism your app uses (if any), and notes to testers, as shown in Figure 4-7.


https://appdev.microsoft.co...

Submit an app

Submit an app

- App name
- Selling details
- Advanced features
- Age rating
- Cryptography
- Packages
- Description
- Notes to testers

2 minutes — App name
Give your app a unique name.
Learn more

5 minutes — Selling details
Pick your app's price, listing categories, and where you want to sell it.
Learn more

5 minutes — Advanced features
Configure push notifications and Live Services and define in-app offers.
Learn more

5 minutes — Age rating and rating certificates
Describe the audience for your app and upload your rating certificates.
Learn more

5 minutes — Cryptography
Declare whether your app uses cryptography and enable package upload.
Learn more

30 minutes — Packages
Upload your app to the Windows Store.
To enable this step, complete the Cryptography page.
Learn more

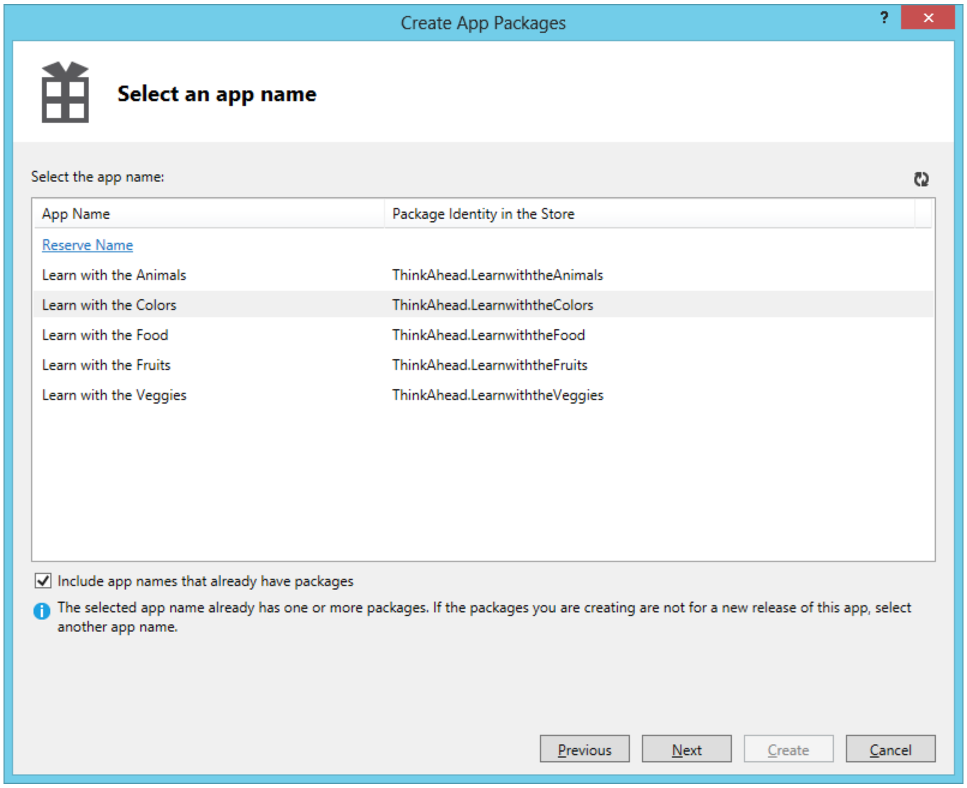
30 minutes — Description
Briefly describe for your customers what your app does.
Learn more

2 minutes — Notes to testers
Add notes about this release for the people who will review your app.
Learn more

FIGURE 4-7 The Submit an App page allows you to fill in information about your application when submitting it to the Windows Store.

After completing the information on the Submit an App page, you need to upload the package. You can build the package directly from Visual Studio 2012 as you saw in the previous section of this chapter, choosing the option to associate the package with the application in the store. From a practical viewpoint, after you create an application using the Windows Store dashboard, you can associate it with the Visual Studio project using the Store menu. This association modifies the application manifest using the publisher name and publisher ID taken from the store services.

To do that, from the Store menu, select the Associate App with the Store menu item to bind the project to an application and import the publisher name and certificate to the project. Alternatively, you can perform the binding operation when you build the application package, as shown in Figure 4-8.



Select the app name:

App Name	Package Identity in the Store
Reserve Name	
Learn with the Animals	ThinkAhead.LearnwiththeAnimals
Learn with the Colors	ThinkAhead.LearnwiththeColors
Learn with the Food	ThinkAhead.LearnwiththeFood
Learn with the Fruits	ThinkAhead.LearnwiththeFruits
Learn with the Veggies	ThinkAhead.LearnwiththeVeggies

✓ Include app names that already have packages

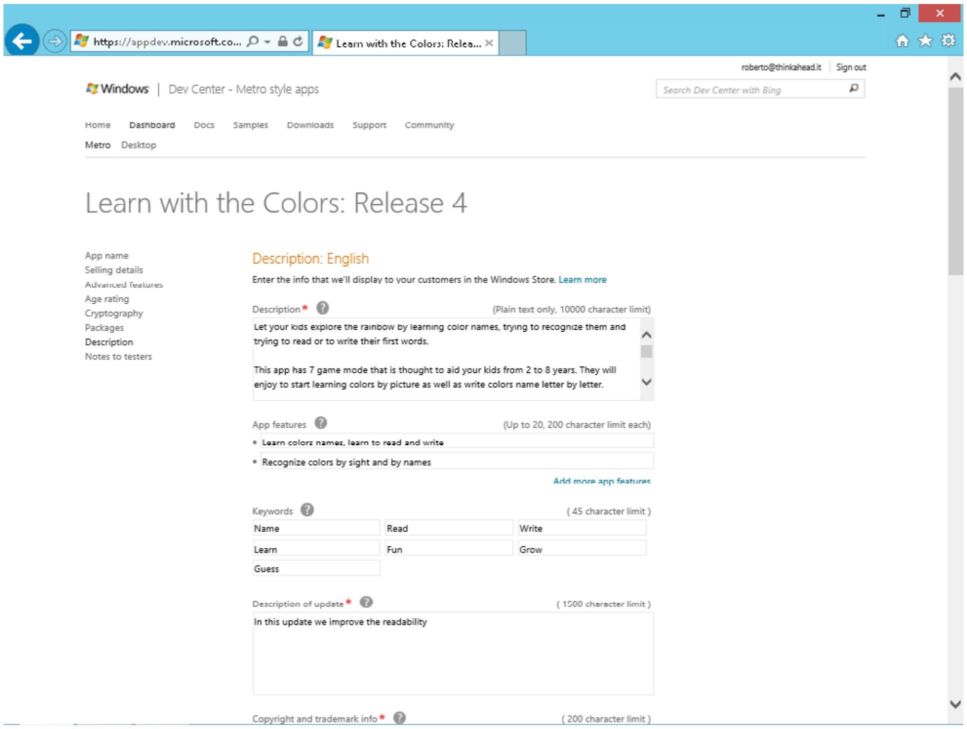
The selected app name already has one or more packages. If the packages you are creating are not for a new release of this app, select another app name.

FIGURE 4-8 Associate an application Visual Studio project with the real Windows Store application.

After the upload operation completes, you need to fill in an important form linked to the Description button that allows you to define all of the marketing information for your app:

- Description of the application (free text)
- Two lines describing major application features
- Seven keywords (may be more in future releases)
- Optional copyright information and license terms
- Eight optional screen shots, each one with a required description
- Promotional images that will be used by the system if your app is selected to appear on the New Apps or Top Apps page of the store
- Application minimum hardware requirements
- An email address that users can write to if they have support requests
- A privacy policy

For example, Figure 4-9 shows the attributes used for the Learn with the Colors application, which is a Windows 8 app the authors published to the Windows Store.


https://appdev.microsoft.co...

Learn with the Colors: Relea...

roberto@thinkahead.it | Sign out

Windows | Dev Center - Metro style apps

Search Dev Center with Bing

Home Dashboard Docs Samples Downloads Support Community

Metro Desktop

Learn with the Colors: Release 4

App name
Selling details
Advanced features
Age rating
Cryptography
Packages
Description
Notes to testers

Description: English

Enter the info that we'll display to your customers in the Windows Store. Learn more

Description* (Plain text only, 10000 character limit)

Let your kids explore the rainbow by learning color names, trying to recognize them and trying to read or to write their first words.

This app has 7 game mode that is thought to aid your kids from 2 to 8 years. They will enjoy to start learning colors by picture as well as write colors name letter by letter.

App features (Up to 20, 200 character limit each)

- Learn colors names, learn to read and write
- Recognize colors by sight and by names

Add more app features

Keywords (45 character limit)

Name	Read	Write
Learn	Fun	Grow
Guess		

Description of update* (1500 character limit)

In this update we improve the readability

Copyright and trademark info* (200 character limit)

FIGURE 4-9 Attributes for the authors' Learn with the Colors app.

For each application you distribute through the Windows Store, the store service provides statistics such as store trends, a financial summary, the number of downloads, and reviews and rating information.

Launching

When you create a new Windows Store application using the Visual Studio template, you will end up with a solution containing one project with a default page called MainPage.xaml and a class that represents the application defined in the App.xaml.cpp and App.xaml.h files. WinRT invokes the method called *OnLaunched* immediately after creating the application instance. You can override this method in your application to perform some activities.

Understand the *OnLaunched* event

In this procedure, you will start coding the event handlers for application events.

1. Create a new application project. To do so, open Visual Studio 2012 and select New Project from the File menu (the sequence can be File | New | Project for full-featured versions of Visual Studio). Choose Visual C++ in the Templates tree and then Windows Store from the list of installed templates. Then choose Blank App (XAML) from the list of available projects.

2. Name the new project **ALMEvents**, and then choose a location on your file system and accept the default solution name. When you've finished, click OK.

   As you learned in Chapter 3, the Windows Store Application template provides a default page (MainPage.xaml), an application entry point in the *App* class (App.xaml.cpp and App.xaml.h files), and a default application description in Package.appxmanifest.

3. Open the App.xaml.cpp file and scroll down until you can see the *OnLaunched* method.

   This method is called by WinRT when the user launches the application. An application is launched when the user clicks the application tile. The default code inside the method simply instantiates a new *Frame* class, sets it as the current content, and then navigates to the main page, calling the *Navigate* method on the frame and passing the *MainPage* class. The last line activates the current content that is the Main Page. The code also contains a test to check for the presence of an existing frame (meaning the application is already running), which will be explained later in this chapter.

   The following snippet shows the *OnLaunched* method:

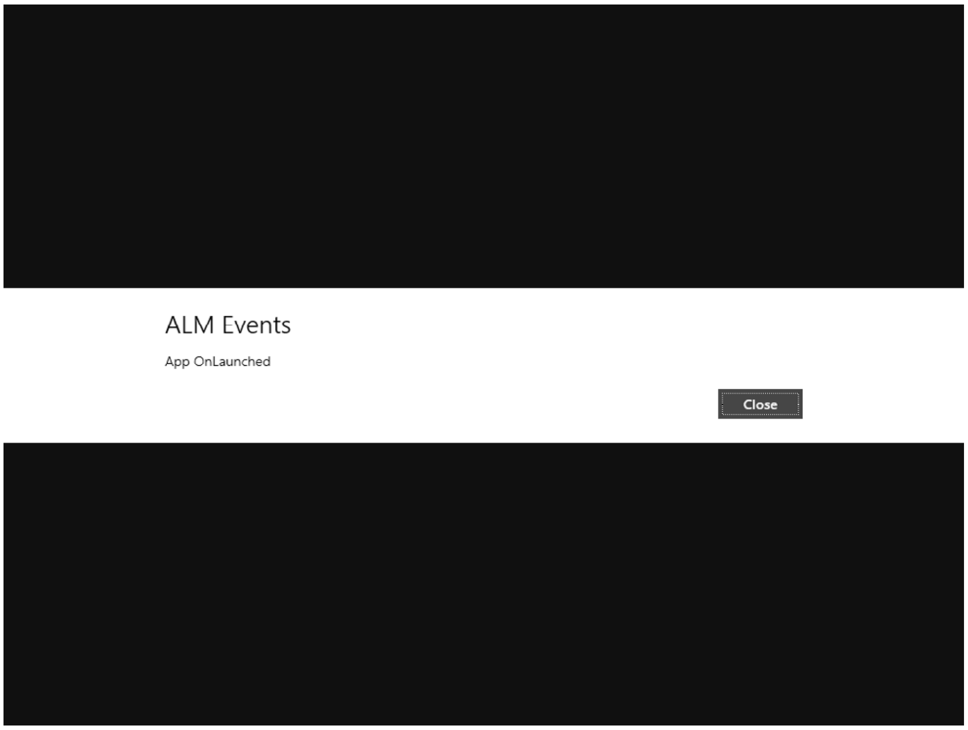
```
void App::OnLaunched(Windows::ApplicationModel::Activation::LaunchActivatedEventArgs^
    args)
{
        auto rootFrame = dynamic_cast<Frame^>(Window::Current->Content);

        // Do not repeat app initialization when the Window already has content,
        // just ensure that the window is active
        if (rootFrame == nullptr)
        {
                // Create a Frame to act as the navigation context and associate it with
                // a SuspensionManager key
                rootFrame = ref new Frame();

                if (args->PreviousExecutionState ==
                    ApplicationExecutionState::Terminated)
                {
                        // TODO: Restore the saved session state only when appropriate,
                        // scheduling the final launch steps after the restore is
                        // complete

                }
                if (rootFrame->Content == nullptr)
                {
                        // When the navigation stack isn't restored, navigate to the
                        // first
                        // page, configuring the new page by passing required
                        // information as a navigation parameter
                        if (!rootFrame->Navigate(TypeName(MainPage::typeid),
                                args->Arguments))
                        {
                                throw ref new FailureException(
                                   "Failed to create initial page");
                        }
                }
                // Place the frame in the current Window
                Window::Current->Content = rootFrame;
                // Ensure the current window is active
                Window::Current->Activate();
        }
        else
        {
                if (rootFrame->Content == nullptr)
                {
                        // When the navigation stack isn't restored, navigate to the
                        // first
                        // page, configuring the new page by passing required information
                        // as a navigation parameter
                        if (!rootFrame->Navigate(TypeName(MainPage::typeid),
                               args->Arguments))
                        {
                                throw ref new FailureException(
                                   "Failed to create initial page");
                        }
                }
                // Ensure the current window is active
                Window::Current->Activate();
        }
}
```

4. Add the following two lines just at the beginning of the method, before the rest of the code, as presented in the previous step:

```
auto dia = ref new Windows::UI::Popups::MessageDialog(
            "App OnLaunched",
            "ALM Events");
dia->ShowAsync();
.....
```

The first line instantiates the *MessageDialog* class, passing to it the content and the title as string parameters. This class represents what in the past was called a *message box*. The second line of code shows the message dialog box in the default location and begins an asynchronous operation for processing the dialog box, and then calls the *Start* method to start the operation.

5. Press F5 to start the application or deploy the application as you learned in Chapter 3, and tap or click the application tile. The following image shows the message dialog box.


ALM Events

App OnLaunched

As you can see, the dialog box is shown full screen, and it displays the title and the content passed as parameters in the class constructor.

6. Click or tap the Close button to close the dialog box. You will see a completely black page—this is because the default page presents nothing.

7. Press the Windows button to open the Start screen (you can also move the mouse in the lower-left corner of the screen and choose Start from the Start menu).

8. Scroll right until you find the ALMEvents application, and tap or click the application tile to launch it again. The application is already running, and you will not see the dialog box; in fact, WinRT does not call the *OnLaunched* method on the application when the application instance is already loaded.

   This behavior is significantly different than in previous versions of Windows, where the system started a new instance of the application each time the user launched it. In Windows 8, there can be only one instance running at the same time. When the user launches an already running application, WinRT just brings the application to the foreground.

9. Close the application by pressing Alt+F4 and repeat steps 5–7 to verify the application flow again.

The parameter received by the *OnLaunched* method is of type *LaunchActivatedEventArgs*, a class that implements the *IActivatedEventArgs* interface you saw in Chapter 3. This interface is implemented by different classes that serve as event arguments for different activation events. The first property of the interface is *Kind*, and it can assume one of the values defined in the *ActivationKind* enumeration. This property lets the developer ask for the kind of launch. For instance, if the application is launched by the user, this property will be *ActivationKind.Launch*; if the application is launched by the system when the user selects it as search target, the property will be *ActivationKind.Search*; if the application is activated to receive something from other applications using a Share contract, the property will be *ActivationKind.ShareTarget*. There are two different methods in the base class to react to this activation. You will see these differences in this chapter.

Show the launch kind

In this procedure, you will change the code of the previous procedure to show the activation kind.

1. Replace the code you inserted in the previous procedure for the *OnLaunched* method to create a message that contains the activation kind as follows. You will need to insert the lines in bold:

   ```
   Platform::String^ message = "App Launched: " + args->Kind.ToString();
   auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
   dia->ShowAsync

   ...
   ```

   The first line uses the *Kind* property of the event args to build the message text, and the second line presents it in a message dialog box.

2. Run the application, deploying it from the Build menu, and start the application by clicking the application tile on the Start screen. You will see the dialog box presenting the message "App Launched: Launch."

3. Click the Close button and do not close the application.

4. Go to the Start screen and click the application tile. You won't see any messages because the application is already running.

5. Close the application using Alt+F4.

Understand the previous state

In this procedure, you will modify the code for the *OnLaunched* method to test the execution state for the previous launch of the application. If the user closes the application normally, the previous execution state will be *ClosedByUser*, telling you that everything went well for the user. If the user has never launched the application, the previous execution state will be *NotRunning*.

1. Change again the first line of the *OnLaunched* event to build a more detailed message that shows the activation kind and the previous execution state by replacing the first line of the method with the one shown in the following code excerpt (bold line):

```
Platform::String^ message = "App Launched: " + args->Kind.ToString()
            + " - Previous State: " + args->PreviousExecutionState.ToString();
auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
dia->ShowAsync();
...
```

2. Deploy the application using the Deploy menu item on the Build menu.

3. Start the application by launching it from the Start screen.

4. Verify that the message "App Launched: Launch – Previous State: ClosedByUser" displays, meaning the application was closed by you previously (if, in fact, you closed it in the previous procedure). The message can be "App Launched: Launch – Previous State: NotRunning" if the application was closed immediately before. Try it closing and launching it from the Start screen quickly.

5. Close the application using Alt+F4.

6. Modify MainPage.xaml by adding two buttons and their corresponding click events in the *Grid* control as follows:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal" VerticalAlignment="Top">
        <Button Click="Crash_Click" Content="Crash" />
        <Button Click="Close_Click" Content="Close" />
    </StackPanel>
</Grid>
```

   The first one will be used to perform an invalid operation that causes a crash of the application. The second will be used to gracefully close the application from code.

7. Implement the event handlers for the *Crash* and the *Close* click events (highlighted in bold) using the following code in the MainPage.xaml.h (Listing 4-1) and MainPage.xaml.cpp (Listing 4-2) files, respectively.

LISTING 4-1 Code-behind file for the *App* class: MainPage.xaml.h

```
#pragma once

#include "MainPage.g.h"

namespace ALMEvents
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public ref class MainPage sealed
    {
    public:
        MainPage();

    protected:
        virtual void OnNavigatedTo(Windows::UI::Xaml::Navigation::NavigationEventArgs^ e)
            override;
    private:
        void Crash_Click(Platform::Object^ sender,
            Windows::UI::Xaml::RoutedEventArgs^ e);
        void Close_Click(Platform::Object^ sender,
            Windows::UI::Xaml::RoutedEventArgs^ e);
        };
}
```

LISTING 4-2 Code-behind file for the *App* class: MainPage.xaml.cpp

```
#include "pch.h"
#include "MainPage.xaml.h"

using namespace ALMEvents;

using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;

// The Blank Page item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=234238

MainPage::MainPage()
{
        InitializeComponent();
}
```

```
/// <summary>
/// Invoked when this page is about to be displayed in a Frame.
/// </summary>
/// <param name="e">Event data that describes how this page was reached. The Parameter
/// property is typically used to configure the page.</param>
void MainPage::OnNavigatedTo(NavigationEventArgs^ e)
{
        (void) e;        // Unused parameter
}


void ALMEvents::MainPage::Crash_Click(Platform::Object^ sender,
     Windows::UI::Xaml::RoutedEventArgs^ e)
{
        int a = 10;
        int b = 0;
        int c = a / b;
}


void ALMEvents::MainPage::Close_Click(Platform::Object^ sender,
     Windows::UI::Xaml::RoutedEventArgs^ e)
{
        Application::Current->Exit();
}
```

8. Deploy the application by right-clicking the project in the solution and choosing Deploy from the context menu.
9. Launch the application from the Start screen, click Close on the dialog box, and click the Crash button on the main page. The application should crash, returning to the Start screen in a few seconds. Be patient.
10. Launch the application again from the Start screen. The dialog box will show NotRunning as the previous state.
11. Close the dialog box and then click the Close button to gracefully close the application.
12. Launch the application again from the Start screen to verify that the dialog box shows NotRunning as the previous state.
13. Close the dialog box and then close the application by using Alt+F4 or swiping your mouse or finger from the upper-center of the screen to the lower-center to close the application in the canonical way.
14. Wait for at least 20 seconds and then launch the application again from the Start screen to verify that the dialog box shows ClosedByUser as the previous state.

To summarize, an application receives a call to the *OnLaunched* method from WinRT when the user launches the application and the application is not already running. This method receives the launch kind and the previous state. Also note that there can be only one instance of a Windows Store application in Windows 8.

Activating

If the user "launches" the application using the Search contract, the application receives a call to the *OnSearchActivated* method, as you learned in Chapter 3. In this case, WinRT calls this procedure *activation*, as the name of the method implies. Activation is a more correct term, since the application is not launched by the user. The parameter args received by the *OnLaunched* method, as you learned in the preceding procedure, has a property called *Kind* that can assume the value of *Search*, but don't be confused by this. When the user selects the target of a search, WinRT invokes the *OnSearchActivated* method on the *App* class and never invokes the *OnLaunched* events. Both event arguments, as well as other event args for other activation methods, implement a common interface; this explains why both of them have the same property. The following procedure clarifies these concepts.

Understand the *OnSearchActivated* method

In this procedure, you will modify the code for the App.xaml.cpp file to test the activation for search. You will use the Search Contract template from Visual Studio, which you learned about in Chapter 3.

1. Implement the Search contract by right-clicking the project in Solution Explorer and choosing Add New Item.
2. Scroll down in the Windows Store folder until you find the Search Contract item.
3. Click the Add button without changing the default name, and click Yes when the dialog box asks you to add the requested files.

   You will not implement a real search page in this procedure; you will just test the activation for searching.

   Adding the Search Contract item modifies the Package.appxmanifest file to declare the Search contract and adds the following line in the App.xaml.cpp file:

```
void ALMEvents::App::OnSearchActivated(
   Windows::ApplicationModel::Activation::SearchActivatedEventArgs^ args)
{

       // TODO: Register the Windows::ApplicationModel::Search::
       //   SearchPane::GetForCurrentView()->QuerySubmitted
       // event in OnWindowCreated to speed up searches once the
       // application is already running

        // If the app does not contain a top-level frame, it is possible that this
        // is the initial launch of the app. Typically this method and OnLaunched
        // in App.xaml.cpp can call a common method.
        auto previousContent = Window::Current->Content;
        auto rootFrame = dynamic_cast<Windows::UI::Xaml::Controls::Frame^>
            (previousContent);
        if (rootFrame == nullptr)
        {
                // Create a Frame to act as the navigation context and associate it with
                // a SuspensionManager key
                rootFrame = ref new Frame();
                Common::SuspensionManager::RegisterFrame(rootFrame, "AppFrame");

                auto prerequisite = Concurrency::task<void>([](){});
                if (args->PreviousExecutionState ==
                    ApplicationExecutionState::Terminated)
                {
                        // Restore the saved session state only when appropriate,
                        // scheduling the final launch steps after the restore is
                        // complete
                        prerequisite = Common::SuspensionManager::RestoreAsync();
                }
                prerequisite.then([=](Concurrency::task<void> prerequisite)
                {
                        try
                        {
                                prerequisite.get();
                        }
                        catch (Platform::Exception^)
                        {
                                // If restore fails, the app should proceed as though
                                // there
                                // was no restored state.
                        }

                        // TODO: Navigate to the initial landing page of the app as if
                        // it were launched. This allows the user to return to your app
                        // from the search results page by using the back button.

                        //Navigate to the search page
                        rootFrame->Navigate(TypeName(
                           SearchResultsPage::typeid), args->QueryText);
                        // Place the frame in the current Window
                        Window::Current->Content = rootFrame;
                        // Ensure the current window is active
                        Window::Current->Activate();

                }, Concurrency::task_continuation_context::use_current());
        }
        else
        {
                //Navigate to the search page
                rootFrame->Navigate(TypeName(SearchResultsPage::typeid),
                    args->QueryText);
                // Ensure the current window is active
                Window::Current->Activate();
        }
}
```

4. Add the following three bold lines just at the beginning of the *OnSearchActivated* method:

```
void ALMEvents::App::OnSearchActivated(
     Windows::ApplicationModel::Activation::SearchActivatedEventArgs^ args)
{
    Platform::String^ message = "App Activated by the Search Contract";
    auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
    dia->ShowAsync();

    ...
```

5. Deploy the application. If you haven't closed the application in the previous procedure yet, activate it and press the Close button or use Task Manager to kill the application.

6. Press Windows+Q, type something in the Search box, and select the ALMEvents application.

7. Verify that the dialog box displays the message "App Activated by the Search Contract."

   You will not receive the dialog box for the application launching because the application was not launched by the user but activated for a search. Visual Basic, C#, and C++ application base classes expose different methods to respond to launch and search activations, while WinJS exposes just a generic activation function where you can test the activation kind property of the event args.

8. Close the application using Alt+F4.

If the user shares some content from another application, the target application receives a different activation called *sharing target activation* (*OnSharingTargetActivated* is the name of the corresponding method). You will learn about this kind of activation and the Sharing contract in Chapter 6.

There are other types of activation, each one corresponding to an operation done by the user. Table 4-1 summarizes the principal activation types.

TABLE 4-1 List of Windows Runtime activations

Method name	Description
Activated	Invoked when the application is activated by tile activation
File Activated	Invoked when the application is activated through file open
File Picker Activated	Invoked when the application is activated through file-dialog association
Search Activated	Invoked when the application is activated through search association
Sharing Target Activated	Invoked when the application is activated through sharing association

Since there are many types of activations, if you want to perform some action not related to a specific type of activation, you can override the *OnInitialize* method on the application class. This method is called from the runtime immediately after the creation of the application instance and before the specific method for a particular activation.

Suspending

WinRT introduces a new concept in the application life cycle that consists of a two-phase process in which the application is suspended when the user leaves it to launch or activate a different one and resumed when the user switches back to it.

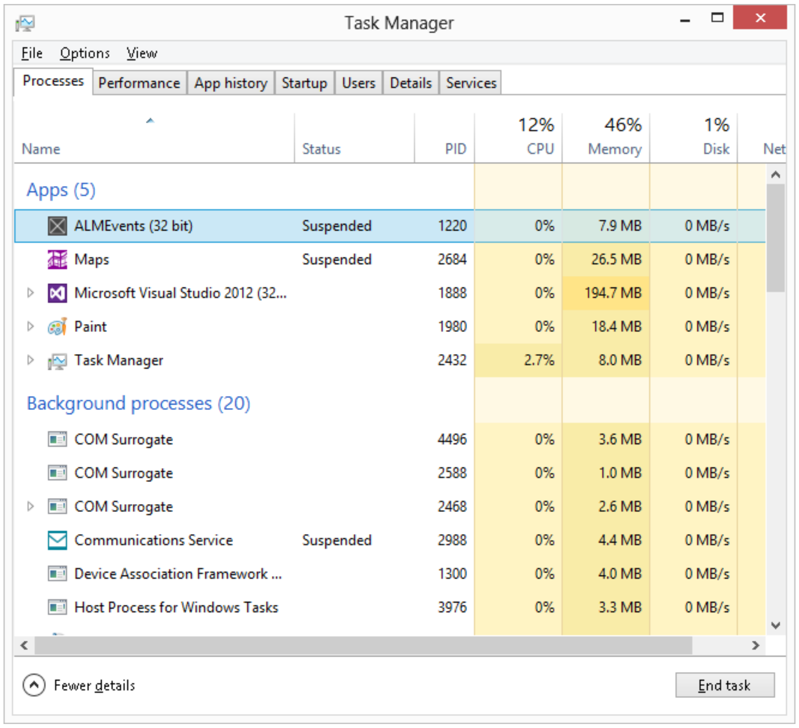
The idea behind this mechanism is to maintain the system responsiveness even if the user launches many applications. Only the foreground application uses processor time, while other applications are suspended by the system. There can be a maximum of two running apps when they are in snapped mode. Usually, when not in snapped mode, there will be only one foreground app. To avoid latency when the user comes back to a previously launched application, WinRT freezes the application memory when suspending an application and places it in a special idle state. No CPU cycle, disk, or network access is given to a suspended application. The result of this mechanism is that the system remains responsive while the resuming phase is practically instantaneous.

Verify the suspension of the application

In this procedure, you will test the suspension mechanism using the application you are building in this chapter.

1. Launch the application from the Start screen. Avoid using the Visual Studio 2012 debugger to test the standard suspension behavior, because while debugging this behavior changes slightly.
2. Close the dialog box that displays the launch message.
3. Press Alt+Tab or the Windows key to put the current application in the background.
4. Open Task Manager and wait until the application goes into the suspended status. To open Task Manager, you can press Windows+Q and search the term “Task Manager” in the Apps list, or you can activate the desktop from the Start screen and right-click the taskbar.

The result of this procedure is shown in Figure 4-10.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	PID	12% CPU	46% Memory	1% Disk	Net
Apps (5)						
ALMEvents (32 bit)	Suspended	1220	0%	7.9 MB	0 MB/s	
Maps	Suspended	2684	0%	26.5 MB	0 MB/s	
Microsoft Visual Studio 2012 (32...		1888	0%	194.7 MB	0 MB/s	
Paint		1980	0%	18.4 MB	0 MB/s	
Task Manager		2432	2.7%	8.0 MB	0 MB/s	
Background processes (20)						
COM Surrogate		4496	0%	3.6 MB	0 MB/s	
COM Surrogate		2588	0%	1.0 MB	0 MB/s	
COM Surrogate		2468	0%	2.6 MB	0 MB/s	
Communications Service	Suspended	2988	0%	4.4 MB	0 MB/s	
Device Association Framework ...		1300	0%	4.0 MB	0 MB/s	
Host Process for Windows Tasks		3976	0%	3.3 MB	0 MB/s	

FIGURE 4-10 Using Task Manager to determine application state and resources.

Note Your screen may be slightly different from Figure 4-10 depending on the columns shown by Task Manager. For instance, the suspension status column has to be manually enabled in order to be shown.

As you can see, the ALMEvents application (PID 1220) is placed in the suspended state. It uses no processor time or disk access at all but, as stated earlier, it uses 7.9 MB of frozen memory. This value may be different on your system.

Switch back to the application by pressing Alt+Tab again and note that the application resumes instantly without showing a launch message dialog box because the application was just resumed from the suspended state.

WinRT will suspend the app as soon as it is in the background at least for 10 seconds. In case you put the app in the background for a time shorter than 10 seconds and you come back, the app probably will not be suspended.

In case of suspension, the system informs the application immediately before the suspension manager starts its work, and the application will have only five seconds to perform any suspension operations. If the app takes longer than five seconds to perform suspension operations, WinRT will terminate it forcibly.

It is very important to understand the complete flow of suspension/resuming before coding against it. The system suspends your app whenever the user switches to another app or to the desktop. The system resumes your app whenever the user switches back to it. When the system resumes your app, the content of your variables and data structures is the same as it was before the system suspended the app. The system restores the app exactly where it left off, so that it appears to the user as if it's been running in the background. In practice, there is no need to save the data the user is working on during the suspension phase if the user comes back to the application. However, if the system does not have the resources to keep your app in memory, or it needs more resources for other applications launched by the user, the system will terminate your app. Your app will not be notified of the termination because WinRT assumes you have already saved any data or state information in the suspension phase. When the user switches back to a suspended app that has been terminated, the app receives a different launch, where you have to write the code that restores the application data.

Now that you understand the complete flow, you will add some code to the application you are developing in this chapter.

Use the *Suspending* event

In this procedure, you will modify the code for the App.xaml.cpp file to intercept the suspension and display a message dialog box. This is not what you would do in a real application, but it is important to understand the complete process.

1. Open the App.xaml.cpp file.
2. In the constructor, the Visual Studio Blank App template prepares the code to hook up the *Suspending* event as follows:

```
App::App()
{
        InitializeComponent();
        Suspending += ref new SuspendingEventHandler(this, &App::OnSuspending);
}
```

3. Use the following code for the event handler for the *Suspending* event, replacing the existing code:

```
void App::OnSuspending(Object^ sender, SuspendingEventArgs^ e)
{
        (void) sender;        // Unused parameter
        (void) e;        // Unused parameter

        Platform::String^ message = "App Suspending";
        auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
        dia->ShowAsync();
}
```

4. Deploy the application and launch it from the Start screen.
5. Close the dialog box that displays the launch.

6. Press the Windows key to put the current application in the background.
7. Open Task Manager and wait until the application is suspended.
8. When the application is suspended, press Alt+Tab again to return to the application and verify that the message "App Suspending" appears.

This dialog box was shown during application suspension but, since the application was not in the foreground anymore, you saw nothing during the system operation. When you reactivate the application, WinRT resumes the application as it was prior to the suspension; this is why you can see the dialog box on the screen only during the resuming operation.

In practice, the dialog box is shown because the application has been resumed exactly where it was left off. The last thing the application did before the suspension was execute the call to display this message. You did not see this message during the suspension because the application was sent to the background.

Simulate an incorrect suspension

The application has only five seconds to respond to the suspension event. If the application needs more time, WinRT kills the application. In this procedure, you will try this behavior.

1. Close the application if you left it open in the previous procedure.
2. Open the App.xaml.cpp file, comment out the existing code of the *OnSuspending* method, and insert the bold line of code in the following excerpt:

```
void App::OnSuspending(Object^ sender, SuspendingEventArgs^ e)
{
        (void) sender;        // Unused parameter
        (void) e;        // Unused parameter

        //Platform::String^ message = "App Suspending";
        //auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
        //dia->ShowAsync();
        Concurrency::wait( 10000 );
}
```

   This code simply waits 10 seconds—too much time for the system, which will kill the application after 5 seconds.

3. Deploy the application.
4. Open an instance of Task Manager and minimize it.
5. Go to the Start screen by pressing the Windows key and launch the application.

6. Maximize Task Manager.

   You can verify that after some time (maybe 20 seconds or more, depending on the system) the application disappears from the application list. This means that the application was killed by the system because the code for the suspending event exceeded the allowed time.

7. Launch the application again and verify the message in the dialog box, which indicates the previous state as Terminated because the application was terminated (killed) by WinRT. This procedure can be slightly unpredictable since the runtime can decide to terminate the application later. This mechanism makes the debugging of the *OnLaunched* event very difficult and time consuming if you are trying to test for a previous termination, but don't worry—at the end of this chapter, you will learn how you can simulate suspension, resuming, and termination from Visual Studio 2012 during a debugging phase.

Request more suspension time

If you need some more time—for instance, to persist some temporary data via web services or in the cloud—you can inform the system that you are executing an asynchronous operation. Call the *SuspendingOperation.GetDeferral* method to indicate that the app is saving its application data asynchronously. When the operation completes, the handler calls the *SuspendingDeferral.Complete* method to indicate that the app's application data has been saved. If the app does not call the *Complete* method, the system assumes the app is not responding and terminates it. When the user launches the application, you should not rely on the validity of the saved application data.

The *SuspendingOperation* method has a deadline time. Make sure all your operations are completed by that time. You can ask the system for the deadline using the *Deadline* property of the *SuspendingOperation* method.

In this procedure, you will change the code for the event handler to write the suspension time on disk using an asynchronous deferred operation. Theoretically, this operation cannot last longer than five seconds, but this example shows the correct code to implement an asynchronous operation.

1. Comment the line with the wait call.

2. Add the code shown in bold in the following block:

```
void App::OnSuspending(Object^ sender, SuspendingEventArgs^ e)
{
    (void) sender;        // Unused parameter

    //Platform::String^ message = "App Suspending";
    //auto dia = ref new Windows::UI::Popups::MessageDialog(message, "ALM Events");
    //dia->ShowAsync();
    //Concurrency::wait( 10000 );
```

```
auto deferral = e->SuspendingOperation->GetDeferral();

auto settingsValues = Windows::Storage::ApplicationData::Current->
    LocalSettings->Values;
if (settingsValues->HasKey("SuspendedTime"))
{
    settingsValues->Remove("SuspendedTime");
}

Windows::Globalization::Calendar^ now = ref new Windows::Globalization::Calendar();
now->SetToNow();
settingsValues->Insert("SuspendedTime", DateTimeFormatter::LongTime::get()->Format(
   now->GetDateTime()));
// Perform the async operation
deferral->Complete();
```

The first uncommented line gets the deferral from the *SuspendingOperation* property of the *SuspendingEventArgs* class. At the end, the code reports the completion of the deferred operation to the system.

The code gets the *LocalSettings* property of the application data and inserts a key called *SuspendedTime* with the current time in the collection. The *LocalSettings* class lets the developer save simple key/value pairs in the local application data folder. As you will learn in Chapter 10, "Architecting a Windows 8 app," WinRT denies access to the classic file system and provides a local or roaming space, called *application data*, that applications can use to store data. This kind of storage recalls in many aspects the *IsolatedStorage* provided by the Silverlight and the Windows Phone runtime. You can also use a *RoamingSettings* property, instead of the *LocalSettings* one, if you want to share your app data across multiple devices. The *RoamingSettings* property is a cloud-based isolated storage, which relates the data to the current user's Windows Live ID account.

Warning Remember that the entire method must return within the deadline.

You can also hook the suspending event inside the code of an application page. This is very useful to save the state of the page during the suspension to restore it in case of termination. Be aware that the *Suspending* event is not raised in the UI thread, so if you have to perform some UI operations, you need to use a dispatcher.

You can debug the code for the suspending method as usual, and you can also force a suspension during a debugging session from Visual Studio. You will try this functionality during the next procedure.

Resuming

In the "Suspending" section of this chapter, you implemented a suspension event handler in the application class to calculate and save the current suspension time to the application data storage.

In this procedure, you will read the saved time from the application data storage during the resume operation from the application class, and then you will implement the code to show the same data within a page.

The resume operation is useless if the application was suspended by the system because the memory dedicated to the application is just frozen and not cleared. Instead, if the system needed more memory and decided to terminate the application, the resume operation is the right place to read the data saved in the suspension procedure.

You can intercept the resume operation hooking up the *Resuming* event of the application class if you want to perform some operations on the application. For instance, you can save the page the user had open before the suspension and, in case of application termination, open that page instead of the default one, as you can see in the following code sample:

```
static Platform::String^ currentPage;

App::App()
{
        InitializeComponent();
        Suspending += ref new SuspendingEventHandler(this, &App::OnSuspending);
        Resuming += ref new EventHandler<Platform::Object^>(this, &App::OnResuming);
}

void App::OnSuspending(Object^ sender, SuspendingEventArgs^ e)
{
    (void) sender;        // Unused parameter

    auto def = e->SuspendingOperation->GetDeferral();

    auto settingsValues = Windows::Storage::ApplicationData::Current->LocalSettings->Values;
    if (settingsValues->HasKey("Page"))
    {
            settingsValues->Remove("Page");
    }
    settingsValues->Insert("Page", currentPage);

    def->Complete();

}

void App::OnResuming(Object^ sender, Platform::Object^ e)
{
    (void) sender;        // Unused parameter
    auto settingsValues = Windows::Storage::ApplicationData::Current->LocalSettings->Values;
    if (settingsValues->HasKey("Page"))
    {
        if (dynamic_cast<Platform::String^>(settingsValues->Lookup("Page")) ==
            "CustomerDetails")
        {
            // Activate the Customer Details Page
        }
    }

}
```

The code is straightforward: the *OnSuspending* event handler saves the name of the current page in the local application data store, and the *OnResuming* event handler reads that value when the application is resumed from a terminated state.

An application can leverage the resuming operation, performing some actions even if the application was not terminated. For instance, you can request data taken from a web service or remote source if the suspend operation was done some minutes before the resume, in order to present fresh content to the user.

Refresh data during resume

In this procedure, you will modify the code for the MainPage.xaml.cpp file to display the time the page was launched, suspended, and resumed.

1. Open the MainPage.xaml file and add three *TextBlock*s. The first one will display the time the page was first opened, the second will display the time the page was suspended, and the third will display the time the page was resumed. Use this code as a reference:

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

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel Orientation="Vertical" VerticalAlignment="Top" Margin="10,01,10,10">
            <Button Click="Close_Click" Content="Close" />
            <TextBlock Name="firstTime" FontSize="24" Margin="10,10,10,10" />
            <TextBlock Name="suspendTime" FontSize="24" Margin="10,10,10,10" />
            <TextBlock Name="resumeTime" FontSize="24" Margin="10,10,10,10" />
        </StackPanel>
    </Grid>
</Page>
```

2. Open the MainPage.xaml.h file and add check that the code corresponds to the following listing:

```
#pragma once
#include "MainPage.g.h"
namespace ALMEvents
{
        /// <summary>
        /// An empty page that can be used on its own or navigated to within a Frame.
        /// </summary>
        public ref class MainPage sealed
        {
        public:
                MainPage();
        protected:
                virtual void OnNavigatedTo(
                    Windows::UI::Xaml::Navigation::NavigationEventArgs^ e) override;
        private:
                void Close_Click(Platform::Object^ sender,
                    Windows::UI::Xaml::RoutedEventArgs^ e);
                void Current_Resuming(
                    Platform::Object^ sender, Platform::Object^ e);
        };
}
```

3. Open the MainPage.xaml.cpp file and use the following code as a reference to hook up the resuming event to display the suspended time restored from the application state and the resumed time.

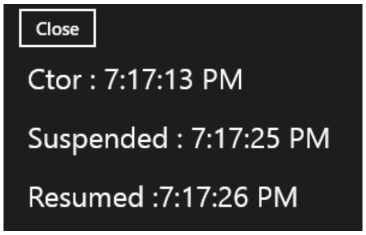
```
#include "pch.h"
#include "MainPage.xaml.h"

using namespace ALMEvents;

using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;
using namespace Windows::Globalization::DateTimeFormatting;

// The Blank Page item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=234238

MainPage::MainPage()
{
    InitializeComponent();
    Windows::Globalization::Calendar^ now = ref new Windows::Globalization::Calendar();
    now->SetToNow();
    firstTime->Text = "Ctor : " + DateTimeFormatter::LongTime::get()->
       Format(now->GetDateTime());
    App::Current->Resuming += ref new EventHandler<Platform::Object^>(this,
       &MainPage::Current_Resuming);
}
```

```
void MainPage::Current_Resuming(Platform::Object^ sender, Platform::Object^ e)
{
    this->Dispatcher->RunAsync(
        Windows::UI::Core::CoreDispatcherPriority::Normal,
            ref new Windows::UI::Core::DispatchedHandler(
                [this]() {
            auto settingsValues = Windows::Storage::ApplicationData::Current->
               LocalSettings->Values;
            if (settingsValues->HasKey("SuspendedTime"))
            {
                suspendTime->Text = "Suspended : " + settingsValues->
                   Lookup("SuspendedTime")->ToString();
            }
            Windows::Globalization::Calendar^ now =
               ref new Windows::Globalization::Calendar();
            now->SetToNow();
            resumeTime->Text = "Resumed :" + DateTimeFormatter::LongTime::get()->
                Format(now->GetDateTime());
        }));
}

void ALMEvents::MainPage::Close_Click(Platform::Object^ sender,
     Windows::UI::Xaml::RoutedEventArgs^ e)
{
        Application::Current->Exit();
}

/// <summary>
/// Invoked when this page is about to be displayed in a Frame.
/// </summary>
/// <param name="e">Event data that describes how this page was reached. The Parameter
/// property is typically used to configure the page.</param>
void MainPage::OnNavigatedTo(NavigationEventArgs^ e)
{
        (void) e;        // Unused parameter
}
```

In the *MainPage* class constructor, the second and subsequent lines of code assign the current time to the first label. This code is executed only when the application instantiates the page, which occurs when the user launches the application or when the application is resumed from a terminated state. This code is not executed when the application is resumed from the suspended state.

The *Current_Resuming* event handler reads the value of the *SuspendedTime* key in the application data store and assigns it to the second label. It then assigns the current time to the last label. This code is not executed in the UI thread, which is why the code is executed by a dispatcher.

4. Deploy the application.
5. Launch the application from the application tile on the Start screen and close the initial dialog box.
6. Press the Windows key and then go to the desktop.
7. Open Task Manager and wait until the application is suspended by the system.
8. Minimize Task Manager.
9. Return to the application by pressing Alt+Tab. The result is shown in the following screen shot.



```
Ctor : 7:17:13 PM

Suspended : 7:17:25 PM

Resumed :7:17:26 PM
```

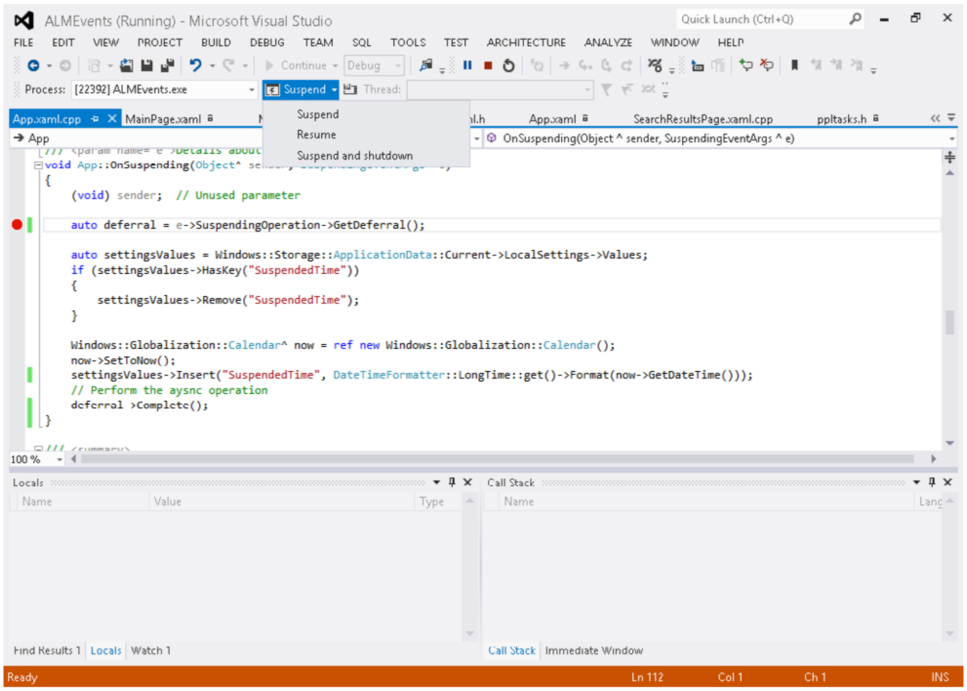
10. Close the application using the Close button.

To facilitate debugging the suspending and resuming events, Visual Studio provides two menu items that enable you to ask WinRT to suspend and resume the application during a debugging session. This feature is useful because it lets you avoid using Task Manager, and you can invoke this event as needed.

Use Visual Studio to debug the suspending and resuming events

In this procedure, you will use Visual Studio to debug the suspending and resuming events.

1. Open the App.xaml.cpp file and place a breakpoint in the first line of the *OnSuspending* method.
2. Open the MainPage.xaml.cpp file and place a breakpoint in the first line of the *Current_Resuming* method.
3. Press F5 to start a debugging session and wait until the application is visible on the screen.
4. Press Alt+F4 to return to Visual Studio, and in the Debug Location toolbar choose Suspend. If this toolbar is not visible, you can enable it by choosing the Toolbars item from the View menu, and then selecting the Debug Location item. The toolbar is visible in the following graphic.



```
void App::OnSuspending(Object^ se
{
    (void) sender;  // Unused parameter

    auto deferral = e->SuspendingOperation->GetDeferral();

    auto settingsValues = Windows::Storage::ApplicationData::Current->LocalSettings->Values;
    if (settingsValues->HasKey("SuspendedTime"))
    {
        settingsValues->Remove("SuspendedTime");
    }

    Windows::Globalization::Calendar^ now = ref new Windows::Globalization::Calendar();
    now->SetToNow();
    settingsValues->Insert("SuspendedTime", DateTimeFormatter::LongTime::get()->Format(now->GetDateTime()));
    // Perform the aysnc operation
    deferral->Complete();
}
```

The breakpoint in the suspend event handler will be hit. Press F5 to continue. The breakpoint in the resume event handler will be hit soon because the application was taken to the foreground by Visual Studio when you pressed F5.

5. Press F5 again and verify that the application is visible and presents the three labels with different times.

6. Press Alt+F4 to return to Visual Studio, and use the Debug Location toolbar to select Resume to verify you can debug directly the resume procedure without the need to debug the suspend procedure first. You can also click the Resume button on the Debug Location toolbar.

7. Using the Debug Location toolbar, select Suspend and Shutdown. The application will first go in the suspended state and then will be terminated by the runtime. With this option, you can debug the code for the *OnLaunched* event to test a previous termination.

To summarize, the system suspends your app whenever the user switches to another app or to the desktop, and the system resumes your app whenever the user switches back to it. When the system resumes your app, the content of your variables and data structures is the same as it was before the system suspended the app—in other words, the system restores the app exactly where it left off, so that it appears to the user as if it's been running in the background the whole time. However, the app may have been suspended for a significant amount of time, so it should refresh any displayed content that might have changed while the app was suspended, such as news feeds or the user's location.

Summary

In this chapter, you saw the complete application life cycle at run time. You saw how to package and install an application in the local system, and how to create a package for the Windows Store. Finally, you were treated to an exploration of the various events that the Windows Runtime (WinRT) fires to launch, activate, suspend, resume, and terminate a Windows 8 application.

Quick reference

To	Do this
Create the application package	Access the Store menu from Visual Studio, and choose Create App Package.
Install an application locally for testing	You can use the classic F5 button to deploy and run the app automatically, or you can choose Deploy from the project contextual menu, or you can create the app package and launch the batch file.
Save temporary data	Use the *Suspending* event from the application class.
Test suspend and resume	Debug the application and use the Suspend and Resume buttons on the Visual Studio Debug Location toolbar.
Uninstall an application	Go to the Start screen, right-click the tile, and choose Uninstall. You can also swipe down your finger on the tile to activate the lower toolbar.

Index

About the authors

LUCA REGNICOLI is a consultant, trainer, and author who has specialized in user interface technologies for .NET applications since 2003. He developed the presentation tier of many enterprise applications in Windows Presentation Foundation, Silverlight, and Windows Phone. Luca is a cofounder of DevLeap, a company focused on providing high-value content and consulting services to professional developers. He is the author of a book in Italian language about ASP.NET. He has also been a regular speaker at major conferences since 2001.

PAOLO PIALORSI is a consultant, trainer, and author who specializes in developing distributed application architectures and Microsoft SharePoint enterprise solutions. He is the author of about 10 books, including *Programming Microsoft LINQ in Microsoft .NET Framework 4* and *Microsoft SharePoint 2010 Developer Reference*. Paolo is a cofounder of DevLeap, a company focused on providing content and consulting to professional developers. He is also a popular speaker at industry conferences.

ROBERTO BRUNETTI is a consultant, trainer, and author with experience in enterprise applications since 1997. Roberto is a cofounder of DevLeap—together with Paolo Pialorsi, Marco Russo, and Luca Regnicoli—a company focused on providing high-value content and consulting services to professional developers. He is the author of a few books: one about ASP.NET, published in 2003, another about Windows Azure Beta, and the last one on Windows Azure published by Microsoft Press in 2011. He has also been a regular speaker at major conferences since 1996 and he works closely with Microsoft in events and training courses.