

Windows® Phone 8 Development Internals

Preview 1

Andrew Whitechapel
Sean McKenna

Vision and Architecture

This chapter covers three core topics: the principles behind the Windows Phone UI and the role that Windows Phone Store apps play in it; a primer on the architecture of the Windows Phone development platform; and an overview of what is required to build and deliver Windows Phone apps. Together, these topics form a critical foundation that will support the detailed examinations of individual platform features that follow in subsequent chapters. And, just so you don't leave this chapter without getting your hands a little bit dirty, you will walk through a simple "Hello World" project to ensure that you're all set to tackle the more involved topics ahead.

A Different Kind of Phone

When Windows Phone 7 was released in the fall of 2010, it represented a significant departure not only from previous Microsoft mobile operating systems, but also from every other mobile operating system (OS) on the market. The user interface was clean, bold, and fluid, with a strong focus on the user's content, rather than app chrome. The Start screen (see Figure 1-1) provided a level of personalization available nowhere else. Live Tiles provided key information at a glance as well as the ability to launch not only apps, but specific parts of those apps, such as opening a favorite website, perhaps, or checking a friend's Facebook status. The developer platform offered unrivalled efficiency and familiar tools, and gave app developers the ability to extend core phone experiences rather than building isolated apps.

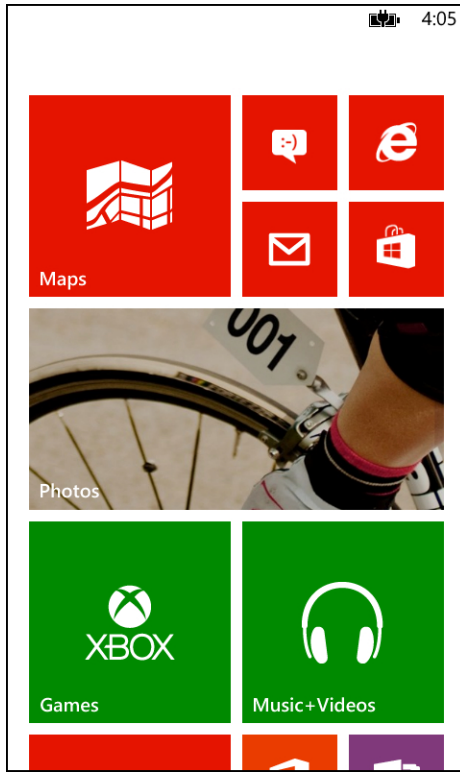


FIGURE 1-1 The distinctive Windows Phone Start screen offers unrivalled personalization.

With Windows Phone 8, Microsoft has significantly expanded the capabilities of the OS, but the fundamental philosophy remains the same. Indeed, much of the original Windows Phone philosophy is now being adopted in the core Windows OS, Microsoft Office, Microsoft Xbox, and other Microsoft products, making it all the more valuable to understand its basic tenets.

The User Interface

The distinctive Windows Phone user interface (UI) is built upon a set of core principles. Understanding these principles will help you to understand not only why the phone looks the way it does, but how you can build beautiful apps that integrate well into the overall experience. After all, in the mobile app marketplace, it is generally not the app with the most features that wins out, but the one which is the easiest and the most enjoyable to use.

For an in-depth review of these principles, watch the talk from Jeff Fong, one of the lead designers for Windows Phone on Channel9. (<http://channel9.msdn.com/blogs/jaime+rodriguez/windows-phone-design-days-metro>)

Light and Simple

The phone should limit clutter and facilitate the user's ability to focus on completing primary tasks quickly. This is one of the principles that drew significant inspiration from the ubiquitous signage in major mass transit systems around the world. In the same way that a subway station needs to make signs bold and simple to comprehend in order to move hundreds of thousands of people through a confined space quickly, Windows Phone intelligently reveals the key information that the user needs among the dozens of things happening at any one time on the phone, while keeping the overall interface clean and pleasing to the eye.

Typography

One element that is common across virtually any user interface is the presence of text. Sadly, it is often presented in an uninteresting way, focusing on simply conveying information rather than making the text itself beautiful and meaningful. Windows Phone uses a distinct font, Segoe WP, for all of its UI. It also relies on font sizing as an indicator of importance. The developer platform provides built-in styles for the various flavors of the Segoe WP typeface, making it simple to incorporate into your app.

Motion

Someone who only experienced the Windows Phone UI through screenshots would be missing out on a significant part of what makes it unique: motion. Tactical use of motion—particularly when moving between pages—not only provides an interesting visual flourish at a time when the user could not otherwise be interacting with the phone, but also a clear connection between one experience and the next. When the user taps an email in her inbox and sees the name of the sender animate seamlessly into the next screen, it provides direct continuity between the two views, such that there can be no doubt about what is happening.

Content, Not Chrome

If you've ever tried browsing around a new Windows Phone that has not yet been associated with a Microsoft Account, you'll find that there isn't very much to look at. Screen after screen of white text on a black background (or the reverse if the phone is set to light theme), punctuated only by the occasional endearing string—"It's lonely in here."—encouraging you to bring your phone to life. The moment when you sign in with a Microsoft Account, however, everything changes. The phone's UI recedes to the background and your content fills the device; contacts, photos, even your Xbox Live avatar all appear in seconds and help to make your phone incredibly personal.

Honesty in Design

This is perhaps the most radical of the Windows Phone design principles. For years, creators of graphical user interfaces (GUIs) have sought to ease the transition of users moving critical productivity tasks from physical devices to software by incorporating a large number of *skeuomorphic* elements in

their designs. Skeuomorphic elements are virtual representations of physical objects, such as a legal pad for a note-taking app or a set of stereo-like knobs for a music player. Windows Phone instead opts for a look that is “authentically digital,” providing the freedom to design UI that’s tailored to the medium of a touch-based smartphone, breaking from the tradition of awkwardly translating a set of physical elements into the digital realm.

The Role of Apps

In addition to its distinctive UI, Windows Phone takes a unique approach to the role of Store apps in the experience. Historically, mobile operating systems only provided simple entry points for users to launch apps—Apple’s iPhone is the canonical example of this, with each app able to display one and only one icon on the phone’s home screen. Although this model is simple and clean, it creates a disjointed environment that obstructs how users want to interact with their content.

With Windows Phone, Microsoft made an explicit shift from the app-focused model to a content and experience-focused model, in which the user is encouraged to think primarily about what he wants to do, rather than how he wants to do it. Something as simple as making a phone call, for example, should not require remembering which cloud services your friend is a member of so that you can launch the appropriate app to look up her phone number. Rather, you should simply be able to launch a unified contacts experience which aggregates information from all of your apps and services.

The content and experience-focused approach doesn’t make Store apps less important; it just changes how they fit in the experience. Windows Phone provides an immersive “hub” experience for each of the primary content types on the phone—photos, music, people, and so on—and each of these hubs offers a rich set of extensibility points for apps to extend the built-in experience. These extensibility points offer additional ways for users to invoke your app, often with a specific task in mind for which you might be uniquely positioned to handle.

Consider photos as an example. There are thousands of apps in the Store that can do something with photos: display them, edit them, or post them to social networks. In a purely app-focused world, the user must decide up-front which tasks he wants to perform and then remember which app would be the most appropriate for that task. In the Windows Phone model, he simply launches the Photos hub, in which he will not only see all of his photos, aggregated across numerous sources, but all of the apps that can do something with those photos. Figure 1-2 shows an example of the photos extensibility in Windows Phone, with “My Photos App” registering as a photo viewer, which the user can access through the Apps entry in the app bar menu for a given photo.

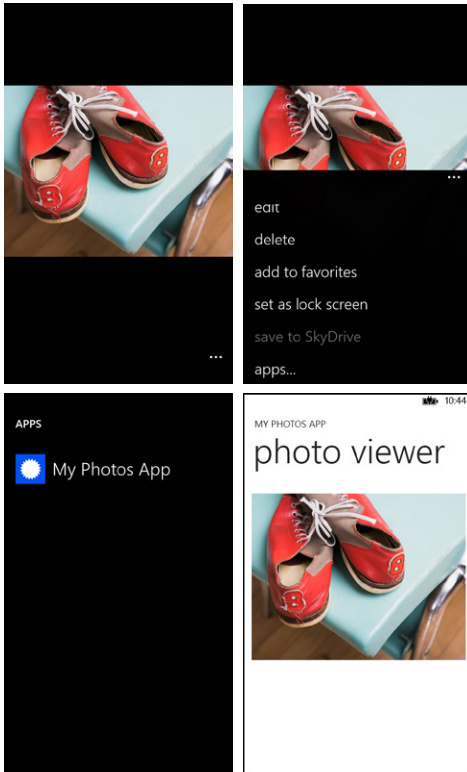


FIGURE 1-2 With Windows Phone, apps can extend built-in experiences, such as the photo viewer.

TABLE 1-1 Windows Phone Extensibility Points

App	Extensibility Point	Windows Phone 7.1	Windows Phone 8.0
Music & Videos	Now playing tile	✓	✓
Music & Videos	History list	✓	✓
Music & Videos	New List	✓	✓
Photos	Apps pivot	✓	✓
Photos	Photo viewer – share	✓	✓
Photos	Photo viewer – apps	✓	✓
Photos	Photo viewer – edit		✓
Search	Search quick cards	✓	✓
Wallet	Wallet items—coupons, transactions, loyalty cards		✓

App	Extensibility Point	Windows Phone 7.1	Windows Phone 8.0
Lock screen	Background photo		✓
Lock screen	Quick status		✓
Lock screen	Detailed status		✓
Speech	Voice command		✓
People	Custom contact stores		✓
Camera	Lenses		✓
Maps	Navigation		✓

Windows Phone Architecture

Now that you understand the user experience (UX) philosophy that drives Windows Phone, it's time to dig a little bit deeper and review some of the core parts of the phone's architecture.

Platform Stack

No chapter on architecture would be complete without the venerable block diagram, and we don't aim to disappoint. Figure 1-3 shows the basic logical components of the Windows Phone 8 platform.

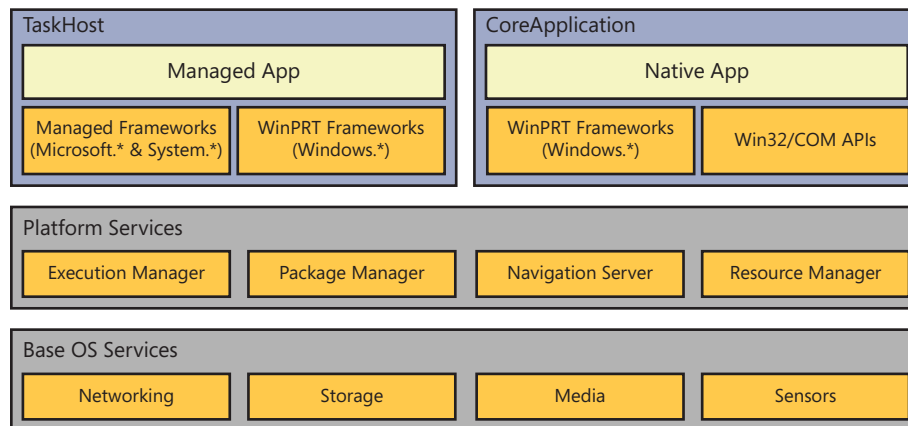


FIGURE 1-3 Windows Phone 8 layers two app models on top of a shared set of platform and OS services.

At the top of the stack sit two distinct app models. The box labeled "TaskHost" represents the XAML app model, which has been the primary model since the launch of Windows Phone 7. To its right is a box labeled "CoreApplication," a new app model for Windows Phone, which is a subset of the new Windows 8 app model. In the Windows Phone 8 release, this app model only supports pure native apps using Direct3D for UI.



Note Although Win32/COM APIs are only shown in the CoreApplication box in Figure 1-3, they are actually callable by managed apps, as well, as long as they are wrapped in a custom WinPRT component.

The two app models rely on a shared set of core platform services. For the most part, Store apps only ever see these services indirectly, but because they play a major role in ensuring that those apps work properly and this is an “Internals” book, we should explore them briefly.

- **Package Manager** The Package Manager is responsible for installing/uninstalling apps and maintaining all of their metadata throughout the app lifecycle. It not only keeps track of which apps are installed and licensed, it also persists information about any app tiles that the user might have pinned to the Start screen and the extensibility points for which an app might have registered so that they can be surfaced in the appropriate places in the OS.
- **Execution Manager** The Execution Manager controls all of the logic associated with an app’s execution lifetime. It creates the hosting process for the app to run in and raises the events associated with app startup/shutdown/deactivation. It performs a similar task for background processes, which also includes proper scheduling of those tasks.
- **Navigation Server** The Navigation Server manages all of the movement between foreground apps on the phone. When you tap an app tile on the Start screen, you are navigating from the “Start app” to the app you chose, and the Navigation server is responsible for relaying that intent to the Execution Manager so that the chosen app can be launched. Likewise, when you press and hold the Back key and choose an app that you started previously, the Navigation Server is responsible for telling the Execution Manager which app to reactivate.
- **Resource Manager** The Resource Manager is responsible for ensuring that the phone is always quick and responsive by monitoring the use of system resources (especially CPU and memory) by all active processes and enforcing a set of constraints on them. If an app or background process exceeds its allotted resource pool, it is terminated to maintain the overall health of the phone.

All of this is built on top of a shared Windows Core, which we will describe in more detail later in this chapter.

App Types

So far, we’ve been referring to Windows Phone apps generically, as if they were all built and run in basically the same way. In fact, Windows Phone 8 supports several different app flavors, depending on your needs. These are described in Table 1-2.

TABLE 1-2 Windows Phone 8 App Types

App Type	Description	Languages Supported	UI Framework	APIs supported
XAML	The most common app type for Windows Phone 7.x. These apps are exclusively written in XAML and managed code.	C# Visual Basic	XAML	Microsoft .NET Windows Phone API WinPRT API
Mixed Mode	<p>These apps follow the XAML app structure but allow for the inclusion of native code wrapped in a WinPRT component.</p> <p>This is well-suited for apps where you want to reuse an existing native library, rather than rewriting it in managed code.</p> <p>It is also useful for cases in which you want to write most of the app in native code (including Direct3D graphics) but also need access to the XAML UI framework and some of the features that are only available to XAML apps such as the ability to create and manipulate Start screen tiles.</p>	C# Visual Basic C/C++	XAML Direct3D (via DrawingSurface)	.NET Windows Phone API WinPRT API Win32/COM API (within WinPRT components)
Direct3D	Best suited for games, pure native apps using Direct3D offer the ability to extract the most out of the phone's base hardware. Also, because they are based on the Windows app model, they offer the greatest degree of code sharing between Windows and Windows Phone.	C/C++	Direct3D	WinPRT API Win32/COM API

What About XNA?

In Windows Phone 7.x, there were two basic app types from which to choose: Silverlight and XNA. As described earlier, managed Silverlight applications are fully supported in Windows Phone 8, but what of XNA? In short, the XNA app model is being discontinued in Windows Phone 8. Existing version 7.x XNA games (and new games written targeting version 7.x), which includes a number of popular Xbox Live titles, will run on 8.0, but developers will not be able to create new XNA games or new Silverlight/XNA mixed-mode apps targeting the version 8.0 platform. Many of the XNA assemblies, such as *Microsoft.Xna.Framework.Audio.dll*, will continue to work in version 8.0, however. Further, version 7.x XNA games are allowed to use some features of Windows Phone 8, such as in-app purchase, using reflection.

Background Processing

When it comes to background execution on a mobile device, users often have conflicting goals. On one hand, they want their apps to continue providing value even when they're not directly interacting with them—streaming music from the web, updating their Live Tile with the latest weather data, or providing turn-by-turn navigation instructions. On the other hand, they also want their phones to last at least through the end of the day without running out of battery and for the foreground app they're

currently using to not be slowed down by a background process that needs to perform significant computation.

Windows Phone attempts to balance these conflicting requirements by taking a scenario-focused approach to background processing. Rather than simply allowing apps to run arbitrarily in the background to perform all of these functions, the platform provides a targeted set of multitasking features designed to meet the needs (and constraints) of specific scenarios. It is these constraints which ensure that the user's phone can actually last through the day and not slow down unexpectedly while performing a foreground task.

Background OS Services

Windows Phone offers a set of background services that can perform common tasks on behalf of apps.

Background Transfer Service The Background Transfer Service (BTS) makes it possible for apps to perform HTTP transfers by using the same robust infrastructure that the OS uses to perform operations such as downloading music. BTS ensures that downloads are persisted across device reboots and that they do not impact the network traffic of the foreground app.

Alarms With the Alarms API, apps can create scenario-specific reminders that provide deep links back into the app's UX. For example, a recipes app might provide a mechanism for you to add an alarm that goes off when it's time to take the main course out of the oven. It might also provide a link that, when tapped, takes the user to the next step in the recipe. Not only does the Alarms API remove the need for apps to run in the background simply to keep track of time, but they can take advantage of the standard Windows Phone notification UI for free, making them look and feel like built-in experiences.

Background Audio Agents

Background audio playback is a classic example of scenario-based background processing. The simplest solution to permitting Store apps to play audio from the background would be to allow those apps to continue running even when the user navigates away. There are two significant drawbacks to this, however:

- Windows Phone already includes significant infrastructure and UI for playing and controlling background audio using the built-in Music & Video app. Leaving every app to build this infrastructure and UI itself involves a significant duplication of effort and a potentially confusing UX.
- A poorly written app running unconstrained in the background could significantly impact the rest of the phone

To deal with these drawbacks, Windows Phone reuses the existing audio playback infrastructure and invokes app code only to provide the bare essentials of playlist management or audio streaming. By constraining the tasks that an audio agent needs to perform, it can be placed in a minimally invasive background process to preserve the foreground app experience and the phone's battery life.

Scheduled Tasks

Scheduled tasks offer the most generic solution for background processing in Windows Phone apps, but they are still ultimately driven by scenarios. There are two types of scheduled tasks that an app can create, each of which is scheduled and run by the OS based on certain conditions:

- **Periodic tasks** Periodic tasks run for a brief amount of time on a regular interval—the current configuration is 25 seconds approximately every 30 minutes (as long as the phone is not in Battery Saver mode). They are intended for small tasks which benefit from frequent execution. For example, a weather app might want to fetch the latest forecast from a web service and then update its app tiles.
- **Resource-intensive agents** Resource-intensive tasks can run for a longer period, but they do not run on a predictable schedule. Because they can have a larger impact on the performance of the device, they only execute when the device is plugged in, nearly fully charged, on Wi-Fi, and not in active use. Resource-intensive agents are intended for more demanding operations such as synchronizing a database with a remote server.

Continuous Background Execution for Location Tracking

In the case of background music playback described earlier, there is very little app code that needs to execute once the initial setup is complete. The built-in audio playback infrastructure handles outputting the actual sound, and the user generally performs tasks such as play, pause, and skip track by using the built-in Universal Volume Control (UVC) rather than reopening the app itself. For the most part, all the app needs to do is provide song URLs and metadata (or streaming audio content) to the audio service.

This is not the case for location tracking and, in particular, turn-by-turn navigation apps. These apps generally need to receive and process up-to-date location information every few seconds to determine whether the user should be turning left or right. They are also likely to offer a rich UX within the app, like a map showing the full route to the destination and the time/distance to go, which will encourage the user to frequently relaunch it. As a result, the audio playback model of using a constrained background task is less suitable in this case. Instead, Windows Phone 8 introduces a concept known as Continuous Background Execution (CBE), which simply refers to the ability of the current app to continue running even if the user navigates away, albeit with a restricted API set.

Security Model

Modern smartphones are by far the most personal items that people have ever owned—in the palm of your hand are the names, phone numbers, and addresses of all of your family and friends, thousands of photos, location history, email correspondence, and, increasingly, financial information stored in mobile wallet apps. Ensuring that all of this information remains safe while the phone moves between physical locations and navigates a variety of websites and apps requires a robust security model.

The Windows Phone security model is based on the notion of *security chambers*, which are isolated containers in which processes are created and executed. The chamber is the security principal to which access rights are granted in the system. The system grants those rights based on the

longstanding security principle of *least privilege*, which holds that an app should not be granted the rights to do anything beyond what is strictly necessary to perform its stated functions. For example, the email app should not have the ability to arbitrarily start the camera and take a picture, because that is clearly not necessary to perform its core function.

So, how does Windows Phone ensure this principle of least privilege? Every security chamber, whether it contains code owned by Microsoft or by an external software developer, starts out with a limited set of privileges—enough to write a self-contained app such as a calculator or a simple game, but not enough to enable the full range of scenarios consumers expect from a modern smartphone. If an app wants to access resources that reside outside of its chamber, such as sending traffic over the network or reading from the user’s contacts, it must be explicitly granted that access via *capabilities*. Capabilities act as a set of access control mechanisms that gate the usage of sensitive resources. The system must explicitly grant capabilities to a chamber.

Windows Phone developers encounter these capabilities directly when building their apps, because accessing any privileged resource from your app requires including the appropriate capability in your app manifest. The graphical manifest editor includes a Capabilities tab that lists all of the available options, as shown in Figure 1-4.

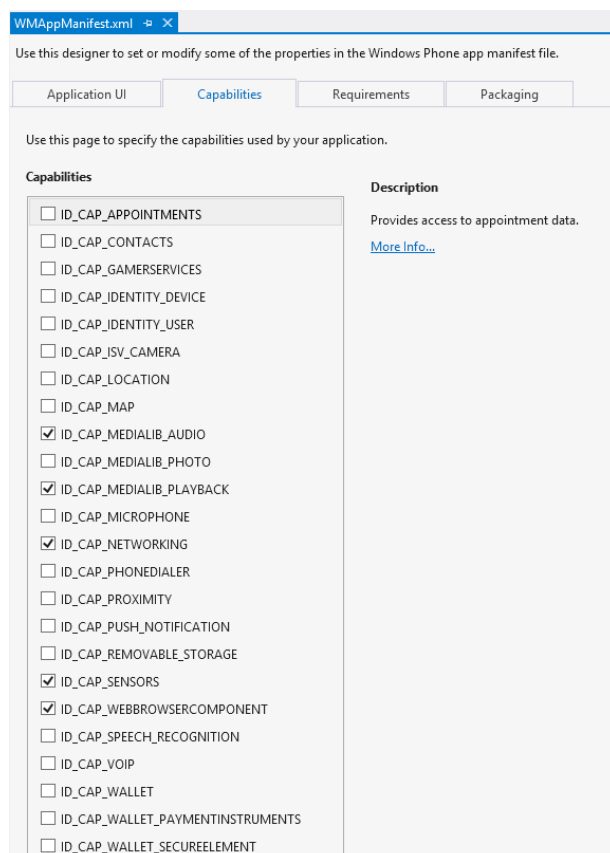


FIGURE 1-4 You select the required capabilities for a chamber in the manifest editor.

Because all of the capabilities listed in the manifest editor are available for Store apps to use, you might ask how the principle of least privilege is being maintained. The answer is that it is the user who decides. The capabilities listed in the manifest are translated into user-readable line items on the Store details page for the app when it's eventually published. The user can then decide whether he feels comfortable installing an app which requires access to a given capability—for example, the user should expect that an app that helps you find nearby coffee shops will need access to your location, but he would probably be suspicious if a calculator app made the same request. Figure 1-5 presents the user-readable capabilities for a weather app. As you can probably guess, “location services” corresponds to ID_CAP_LOCATION, and “data services” is the replacement for ID_CAP_NETWORKING.



FIGURE 1-5 Security capabilities are displayed as user-readable strings in an app's details page.

Capability Detection in Windows Phone 8

It's worth mentioning that Windows Phone 8 has introduced a subtle but important change in how capabilities are detected during app ingestion. In Windows Phone 7.x, the capabilities that the app developer included in the manifest that was submitted to the Store were discarded and replaced with a set determined by scanning the APIs used in the app code. In other words, if you included the `ID_CAP_LOCATION` capability in your manifest but never used any of the location APIs in the *System.Device.Location* namespace, that capability would be removed from the final version of your XAP package (XAP [pronounced "zap"] is the file extension for a Silverlight-based application package [.xap]) and the Store details page for your app would not list location as one of the resources it needed. Given this Store ingestion step, there was no reason for a developer to limit the capabilities that her app was requesting during development. Anything that she didn't end up using would simply be discarded as part of her submission.

With the introduction of native code support in Windows Phone 8, this approach is no longer feasible, and developers are now responsible for providing the appropriate list of capabilities in their app manifests. If an app fails to list a capability that is required for the functionality it is providing, the associated API calls will simply fail. On the other hand, if an app requests a capability that it doesn't actually need, it will be listed on its Store details page, potentially giving the user pause about installing it.



Note For managed code apps, developers can continue to use the CapDetect tool that ships with the Windows Phone SDK to determine which capabilities they need.

Windows and Windows Phone: Together at last

Even though the distinctive UX described earlier in this chapter did not change significantly between Windows Phone 7 and Windows Phone 8, there have been dramatic shifts happening below the surface. For the first time, Windows Phone is built on the same technology as its PC counterpart. In this section, we will describe the two core parts of that change which impact developers: the shared Windows core, and the adoption of the Windows Runtime.

Shared Core

By far the most significant architectural change in Windows Phone 8 is the adoption of a shared core with Windows, but you might be wondering what a “shared core” actually means. In fact, it contains two distinct components. At the very bottom is the Windows Core System, the most basic functions of the Windows OS, including (among other things) the NT kernel, the NT File System (NTFS), and the networking stack. This minimal core is the result of many years of architectural refinement, the goal of which was to provide a common base that could power multiple devices, including smartphones.

Above the Core System is Mobile Core, a set of Windows functionality that is not part of Core System but which is still relevant for a smartphone. This includes components such as multimedia, CoreCLR, and Trident, the rendering engine for Internet Explorer. Figure 1-6 illustrates some of the shared components on which Windows and Windows Phone rely. Note that Mobile Core is only a distinct architectural entity in Windows Phone. Windows contains the same components as Mobile Core, but they are part of a larger set of functionality. This is depicted by a dashed line around the Mobile Core components in the Windows 8 portion of the diagram.

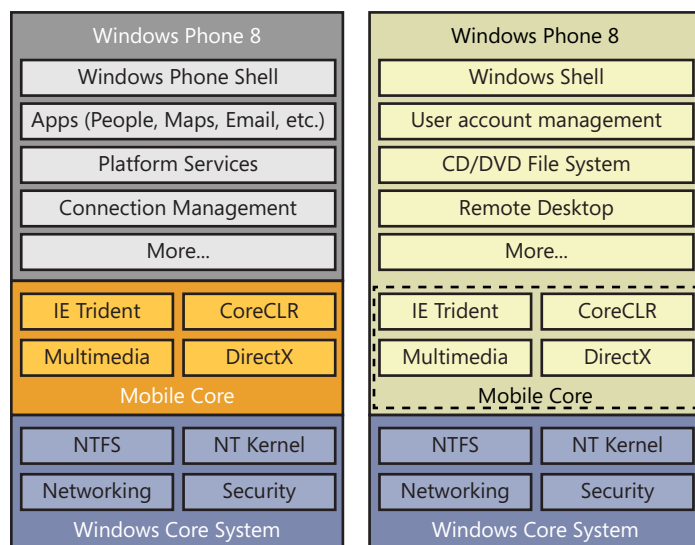


FIGURE 1-6 Windows 8 and Windows Phone 8 share a common core.

Core System and Mobile Core only represent the alignment of Windows and Windows Phone where the two operating systems are running exactly the same code. There are numerous other areas where APIs and behavior are shared, but with slightly different implementations to account for the different environments. For example, the location API in Windows Phone automatically incorporates crowd-sourced data about the position of cell towers and Wi-Fi access points to improve the accuracy of location readings, an optimization which is not part of the Windows 8 location framework.

Windows Runtime

For consumers, the most radical change in Windows 8 is the new UI. For developers, it is the new programming model and API set, collectively known as the Windows Runtime, or *WinRT*. Although Microsoft has delivered a variety of new developer technologies on top of Windows over the years (most notably .NET), the core Windows programming model has not changed significantly in decades. WinRT represents not just a set of new features and capabilities, but a fundamentally different way of building Windows apps and components.

The WinRT platform is based on a version of the Component Object Model (COM) augmented by detailed metadata describing each component. This metadata makes it simple for WinRT methods and types to be “projected” into the various programming environments built on top of it. In Windows Phone, there are two such environments, a CoreCLR-based version of .NET (C# or Visual Basic) and pure native code (C/C++). We will discuss WinRT throughout the book, covering both consumption of WinRT APIs from your apps as well as creation of new WinRT components.



Note Even though the core architecture of the Windows Runtime and many of the APIs are the same for Windows and Windows Phone, the two platforms offer different versions of the API framework which sits on top of it. For instance, Windows Phone does not implement the *Windows.System.RemoteDesktop* class, but does add some phone-specific namespaces, like *Windows.Phone.Networking.Voip*. To avoid any confusion, the term Windows Phone Runtime (WinPRT) is used to refer to the implementation of the Windows Runtime and API framework on Windows Phone. We will use WinPRT throughout the remainder of the book.

Building and Delivering Apps

Now that you understand the fundamentals of Windows Phone, it's time to start looking at how you can build and deliver apps that run on it.

Developer Tools

Everything you need to get started building Windows Phone 8 apps is available in the Windows Phone 8 SDK, which is available as a free download from the Windows Phone Dev Center at <http://dev.windowsphone.com>. In particular, the Windows Phone 8 SDK includes the following:

- Microsoft Visual Studio 2012 Express for Windows Phone
- Microsoft Blend 2012 Express for Windows Phone
- The Windows Phone device emulator

- Project templates, reference assemblies (for managed code development), and headers/libraries (for native code development)

As with previous versions of the Windows Phone SDK, Visual Studio Express and Blend Express can be installed on top of full versions of Visual Studio and Blend, seamlessly merging all of the phone-specific tools and content directly into your existing tools. Throughout the book, we will refer to Visual Studio Express 2012 for Windows Phone as the primary development environment for Windows Phone 8, but everything we describe will work just as well with any other version of Visual Studio once you have the Windows Phone 8 SDK installed.



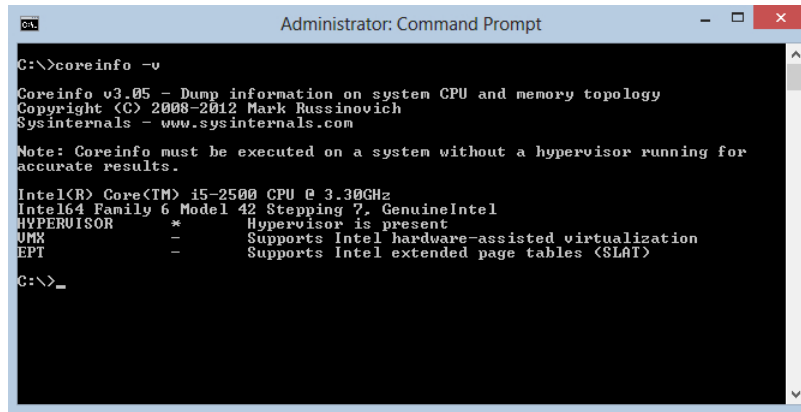
Note Visual Studio 2012, including Visual Studio 2012 Express for Windows Phone, can only be installed on Windows 8.

Windows Phone Emulator System Requirements

The Windows Phone 8 SDK includes a new version of the Windows Phone emulator for testing apps directly on your desktop. The new emulator is built on the latest version of Microsoft Hyper-V, which requires a 64-bit CPU that includes Second Level Address Translation (SLAT), a memory virtualization technology included in most modern CPUs from Intel and AMD.

To check if your CPU supports SLAT, do the following:

1. Download the Coreinfo tool from <http://technet.microsoft.com/en-us/sysinternals/cc835722>.
2. Open a command prompt as an administrator. From the Start menu, type **cmd** to find the command prompt, right-click it, and then choose Run As Administrator.
3. Navigate to the location where you downloaded Coreinfo and run `CoreInfo -v`.
4. Look for a row labeled EPT (for Intel CPUs) or NP (for AMD). If you see an asterisk, as in Figure 1-7, you're all set. If you see a dash, your CPU does not support SLAT and will not be capable of running the new Windows Phone Emulator. Note that if you have already activated Hyper-V on your computer, you will see an asterisk in the HYPERVISOR row and dashes elsewhere else. In this case, you can safely ignore the dashes as your computer is already prepared to run the Windows Phone Emulator.



```
C:\>coreinfo -v

Coreinfo v3.05 - Dump information on system CPU and memory topology
Copyright (C) 2008-2012 Mark Russinovich
Sysinternals - www.sysinternals.com

Note: Coreinfo must be executed on a system without a hypervisor running for
accurate results.

Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz
Intel64 Family 6 Model 42 Stepping 7, GenuineIntel
HYPERVISOR      *      Hypervisor is present
VMX              -      Supports Intel hardware-assisted virtualization
EPT              -      Supports Intel extended page tables (SLAT)

C:\>_
```

FIGURE 1-7 Use the free Coreinfo tool to determine if your computer can run the new Windows Phone emulator



Note SLAT is required only to run the Windows Phone emulator. You can still build Windows Phone 8 apps on a non-SLAT computer; you will simply need to deploy and test them on a physical device.

Building for Windows Phone 7.x and 8.x

Because Windows Phone 8 requires new hardware, it will take some time for the installed base of Windows Phone 8 devices to surpass the existing Windows Phone 7.x phones. During that time, you will likely want to deliver two versions of your app, one for 7.x and one for 8.0. The Windows Phone 8 developer tools have full support for this approach.

In Visual Studio 2012 Express for Windows Phone, you can create new projects for Windows Phone 7.1 and Windows Phone 8.0, and each will be deployed to the appropriate emulator image for its target platform. You can also run your version 7.1 apps on the Windows Phone 8 emulator to ensure that it behaves as expected—even though Windows Phone 8 is backward-compatible with version 7.0 and 7.1 apps, it is always worth verifying that there aren't any nuances in the platform behavior for which you might want to account.

Lighting up a Windows Phone 7.1 app with new tiles

To truly take advantage of the new platform features in Windows Phone 8, you must build a version of your app which explicitly targets version 8.0. Because there is some additional overhead to creating and managing a separate XAP for version 8.0, Windows Phone 8 allows Windows Phone 7.1 apps to create and manage the new Live Tile templates available in the latest release. This approach is based on reflection and is described in detail in Chapter 12, "Tiles and Notifications."

App Delivery

Windows Phone 7.x offered a single broad mechanism for distributing apps: the Windows Phone Application Store (previously, the Windows Phone Application Marketplace). In Windows Phone 8, the Store will continue to be the primary source of apps for most customers. However, the distribution options have been expanded to include additional channels for distributing enterprise apps—enterprise customers will be able to deliver apps to their employees via the Internet, intranet, email, or by loading them on a microSD card and inserting the card into the phone. The options for app deployment in Windows Phone 8 are depicted in Figure 1-8.

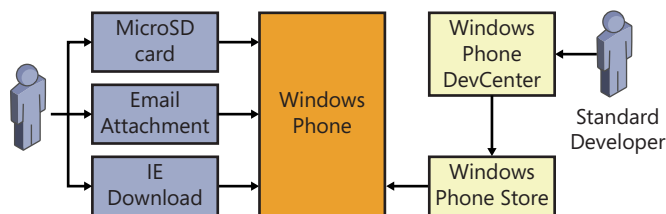


FIGURE 1-8 Windows Phone 8 adds multiple enterprise deployment options.

If you're familiar with any flavor of .NET technology, you know that building a project doesn't generally convert your code into something that's directly executable by a CPU. Rather, it is converted into Microsoft Intermediate Language (MSIL), a platform-independent instruction set, and packaged into a dynamic-link library (DLL). In the case of Windows Phone, these DLLs are then added to your app package for delivery to the phone, where it remains until the user launches the app. At that point, the just-in-time (JIT) compiler turns those DLLs into native instructions targeting the appropriate platform—ARM for physical devices and x86 for the Windows Phone emulator.

In Windows Phone 8, this process changes, such that all apps are precompiled as part of the Windows Phone Store submission process. This means that when a user downloads an app from the Store, the app package already contains code that is compiled for ARM. Because no "JITing" is required when the app is starting up or running, users should experience faster app load times and improved runtime performance.



Note Existing Windows Phone 7.1 apps are automatically precompiled in the Windows Phone Store. No action is required from the owners of those apps.

Getting Started with “Hello World”

By now, you are well versed in the fundamentals of Windows Phone. Go ahead and file all of that knowledge away, because it's time to get into some code. Those of you who are seasoned Windows Phone developers will no doubt be tempted to skip this section, but you might want to at least ensure that your installation of the Windows Phone Developer Tools is working properly before diving into more advanced topics. In particular, you should try to launch a project in the Windows Phone emulator to ensure that Hyper-V is fully enabled, and then navigate to a web page in Internet Explorer to verify that networking is properly set up.

Creating a Project

Once you've installed the Windows Phone SDK from the Dev Center, get started by launching Visual Studio. The first screen you see is the Visual Studio Start Page, as demonstrated in Figure 1-9.

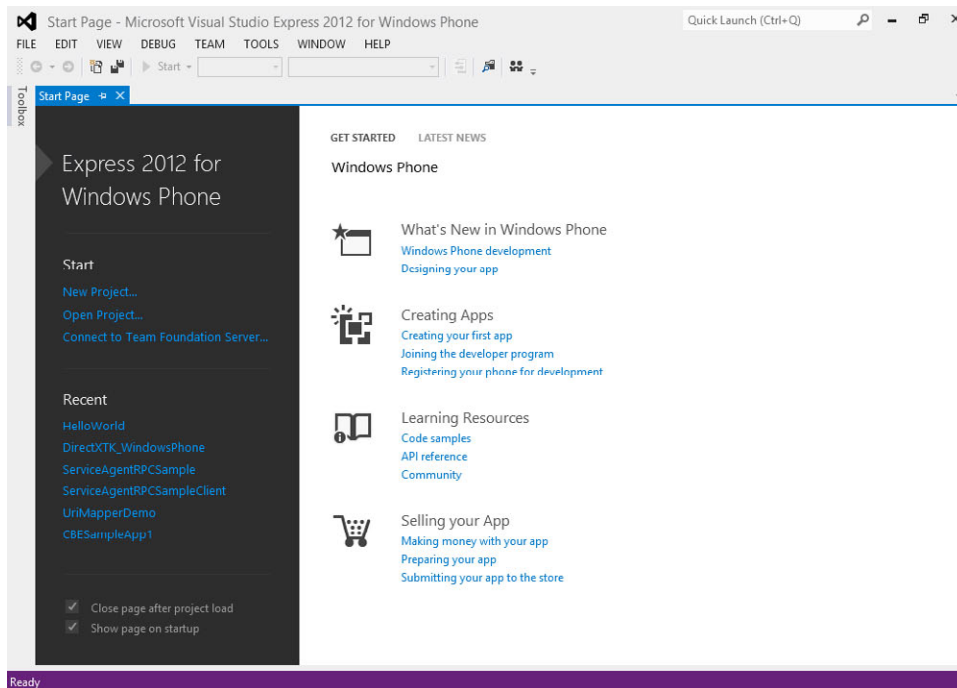


FIGURE 1-9 The first screen you see upon launching Visual Studio is the Start Page, which offers a quick way to start a new project.

On the left side of the Start Page, in the navigation pane, Choose New Project. This brings up the New Project dialog box, in which you can choose the type of project that you want to create and the language in which you want to write it. XAML apps written in C# are the most common type on Windows Phone, so we will start there. Under Templates, click Visual C#, choose Windows Phone App, and then name it **HelloWorld**, as shown in Figure 1-10.

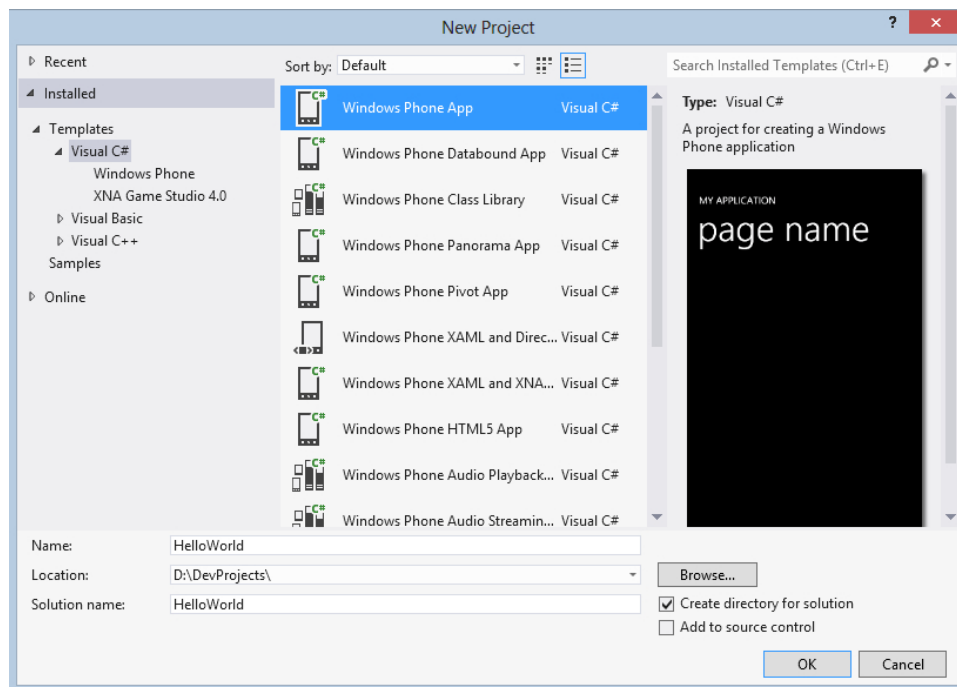


FIGURE 1-10 The New Project dialog box offers a number of templates for creating new apps, class libraries, background agents, and more. To get started, create a simple Windows Phone App in C#.

If your project was created successfully, you should be looking at a screen that resembles Figure 1-11, with *MainPage.xaml* already opened for you.

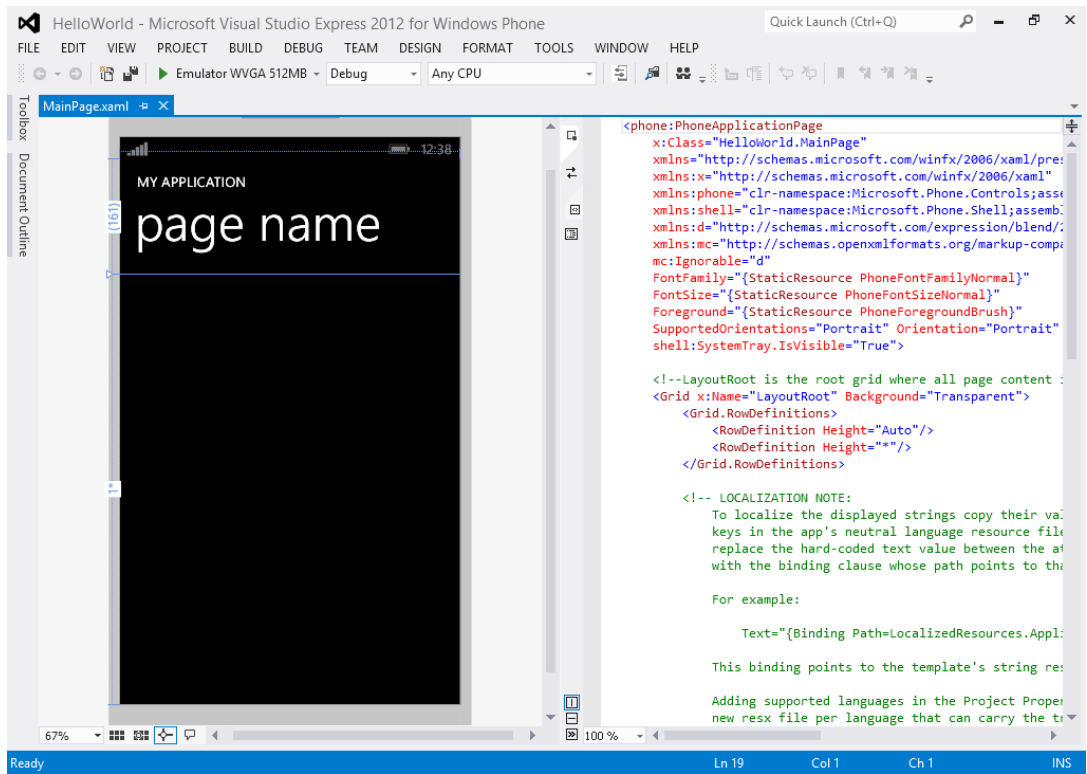


FIGURE 1-11 By default, for a Windows Phone project in C#, Visual Studio starts on *MainPage.xaml*.

Understanding the Project Structure

MainPage.xaml is one of a number of folders and files included in the default Windows Phone project template. Some of these have special meaning which might not be obvious at first glance, so it's worth taking a quick tour of the standard project structure while you're building your "Hello World" app. Figure 1-12 shows an expanded view of Solution Explorer for "Hello World."



Note The structure of the project template and the list of default files included will differ for other project types, especially those involving native code and Direct3D. See Chapter 21, "C/C++ Development in Windows Phone 8," for a detailed description of the native project templates.

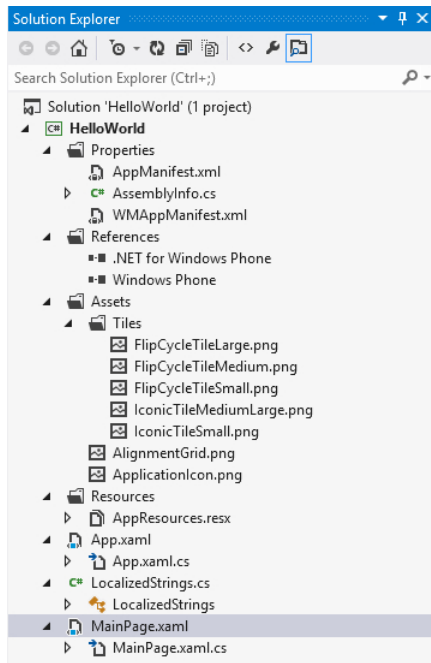


FIGURE 1-12 Windows Phone project templates include a number of special files to get you started.

The first important file is *WAppManifest.xml*, the app manifest. The app manifest contains all of the information that the OS needs to know about the app in order to surface it properly on the phone. Some elements of the manifest (for example, hardware requirements) are also used during the Store submission process. The manifest includes (among other things) the following:

- The app name
- A reference to its app list icon and default Start tile
- Its supported resolutions (new in Windows Phone 8)
- The list of security capabilities it requires, such as location and photos, and any hardware requirements, such as NFC or a front-facing camera.
- Any extensibility points for which the app is registering—for example, as an entry in the Photos share picker

In Visual Studio Express 2012 for Windows Phone, many of these manifest attributes are now configurable through a simple GUI. However, some features, such as registering for extensibility points, still require direct editing of the underlying XML file.



Tip By default, Visual Studio will always display the GUI tool when you double-click *WMAppManifest.xml*. To configure additional settings with the raw XML editor, in Solution Explorer, right-click the app manifest file. In the options menu that opens, select Open With, and then click XML (Text) Editor. To return to the GUI tool, double-click the file again.

The Assets folder is provided as a location to include the core images that your app should provide. At the root of the Assets folder is a file called *ApplicationIcon.png*. This is a default icon which is shown for your app in the app list. The Tiles subfolder is prepopulated with a handful of icons for use in the FlipCycle and Iconic tile templates, which are discussed in detail in Chapter 12. All of these files are placeholders intended to show you which images need to be provided. You can (and should) change them to something representative of your app before submitting it to the Store or distributing it to others.

Together, *Resources\AppResources.resx* and *LocalizedStrings.cs* provide the initial framework for developing a fully localized app. Localization is outside the scope of this book, but it is well documented on MSDN. See [http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff637520\(v=vs.92\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff637520(v=vs.92).aspx) for details on building a fully localized Windows Phone app.

App.xaml provides a convenient location to store resources that you intend to use throughout your app such as UI styles. Its code counterpart, *App.xaml.cs*, contains critical startup code that you should generally not modify and some empty handlers for the core app lifetime events—Launching, Activated, Deactivated, and Closing. If you want to take any action when your app is opened, closed, paused, or resumed, you will need to fill these in. This is discussed in more detail in Chapter 2, “App Model and Navigation.”

MainPage.xaml is the default starting point for your app, which we will return to momentarily to make some changes for our “Hello World” app. You can think of pages in Windows Phone as being the effective equivalent to web pages. They contain both the definition of the UI that will be displayed to the user as well as the bridging code between that UI and the rest of the app’s functionality. The role of pages in the Windows Phone navigation model is explored in depth in Chapter 2.



Tip Remember that the project templates are just a starting point for your app; you don’t need to be locked into to their structure or content. For instance, if you’re following a Model-View-ViewModel (MVVM) pattern, you might want to consolidate all of your views in a single subfolder. If so, don’t hesitate to move *MainPage.xaml* to that folder or create a new page to act as your main page in that location. Just remember to update the Navigation Page setting in your manifest so that the system knows where to start your app.

Adding a Splash Screen

New Windows Phone 7.1 projects also include a file named *SplashScreenImage.jpg*, which, as the name implies, is rendered as a splash screen while the app is loading. Given the improvements in app launch time in Windows Phone 8, it is assumed that most apps will not need a splash screen, so the file has been removed from the default project template. If you believe your app could benefit from a splash screen, simply add a file named *SplashScreenImage.jpg* to the root of the project, with its Build Action set to Content.

Greeting the World from Windows Phone

Now that you understand what all the files in the default project mean, return to *MainPage.xaml*, which has been waiting patiently for you to bring it to life. By default, Visual Studio displays pages in a split view, with the Visual Studio designer on the left and the XAML markup that defines the UI on the right. You can make changes to your page by manipulating controls in the visual designer or by directly editing the XAML.

Start by using the designer to make changes to the default text that the project template has included at the top of the page. Double-click the words MY APPLICATION and change the entry to **HELLO, WORLD**. Likewise, double-click "page name" and change it to **welcome**.

Now, redirect your attention to the right side of the screen, where the XAML markup for your *MainPage* is shown. You will probably be able to spot where the changes you just made were reflected in the underlying XAML. The `<StackPanel>` element with the name *TitlePanel* should now look like this:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
  <TextBlock Text="HELLO, WORLD" Style="{StaticResource PhoneTextNormalStyle}" Margin="12,0"/>
  <TextBlock Text="welcome" Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
```

Directly below the *TitlePanel*, you should find a *Grid* element called *ContentPanel*. Replace this element with the following XAML:

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <TextBlock
    x:Name="helloTextBlock"
    Text="Hello from Windows Phone 8!"
    Foreground="{StaticResource PhoneAccentBrush}"
    Grid.Row="0"
    HorizontalAlignment="Center"/>

  <Button
    x:Name="goodbyeButton"
    Content="Say goodbye!"
    Grid.Row="1" Click="Button_Click_1"/>
</StackPanel>
```

This markup creates a simple *StackPanel* in your app, which is then filled with a *TextBlock* and a *Button*. The *TextBlock* contains the critical greeting, whereas the *Button* suggests that the meeting might not last very long. You use a *StackPanel* in this case because it is a simple and efficient way of displaying a set of visual elements in a horizontal or vertical arrangement.

As mentioned earlier, a page contains both the markup describing how the UI looks and the connective code that bridges the gap between the UI and the rest of the app's logic. In this case, the button acts as your first glimpse into that code. Double-click the *Button* in the Visual Studio designer. This opens *MainPage.xaml.cs*, which is known as a *code-behind* file because, as its name implies, it contains the code behind a given page. A code-behind file is created automatically for every page you create in a managed Windows Phone project.

You will notice that Visual Studio has not only opened the code-behind file, but it has taken the liberty of creating a click event handler (named *goodbyeButton_Click*) for the button that you added to your page. You will use this event handler to add some code that makes your app actually do something.



Note It might seem odd to be handling “click” events in an app built for an exclusively touch-based platform. The reason is that the managed UI framework for Windows Phone is primarily based on Microsoft Silverlight, which was initially built as a web browser plugin. Because a “tap” in a phone app is semantically equivalent to a click on a web page, there was no compelling reason to rename the event.

Add the following code to the click event handler:

```
helloTextBlock.Visibility = System.Windows.Visibility.Collapsed;  
goodbyeButton.IsEnabled = false;
```

As you can probably discern, this code will do two things: it makes your “Hello from Windows Phone 8!” text disappear, and it disables your button. To be sure, it's time to run the app.

Cross your fingers and press F5.

Within a few seconds, you should see the Windows Phone Emulator starting up. By default, Windows Phone 8 projects target the WVGA 512MB emulator, meaning a virtualized version of a Windows Phone 8 device with a WVGA (800x480) screen and 512 MB of memory. You can easily change this through a drop-down menu in the Visual Studio toolbar, as shown in Figure 1-13.

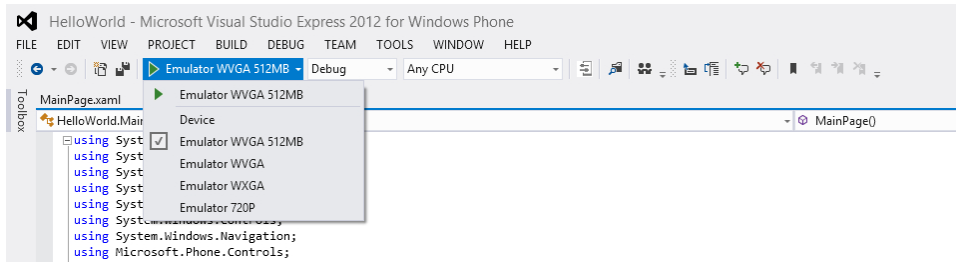


FIGURE 1-13 By default, Visual Studio deploys Windows Phone 8 projects to a WVGA 512MB emulator. You can change this target through a drop-down in the toolbar.

If all has gone according to plan, you should see your app running in the emulator, with your MainPage displayed and ready. Go ahead and click the Say Goodbye! button and you should see your “Hello from Windows Phone 8!” text disappear and your button become grayed out, indicating that it’s been disabled.

That’s it! If you’ve made it this far successfully, you should have everything set up correctly and you should be ready for the more detailed topics in the chapters ahead.

Deploying to a Windows Phone Device

The Windows Phone Emulator is sufficient for most of the app development you will do, especially while learning about the platform as you’ll be doing throughout this book. Once you start building a real app that you intend to submit to the Windows Phone Store, however, you will want to deploy it to a real device.

The first step in deploying to a device is registering as a developer in the Windows Phone Dev Center at <http://dev.windowsphone.com>. Once you’re registered, you can “unlock” your device for deploying apps by using the Windows Developer Registration Tool, which is included in the Windows Phone SDK. Simply connect your device to your computer via USB, run the tool, and then enter your Microsoft Account details. Within a few seconds, your device will be ready for deployment directly from Visual Studio.

Summary

In this chapter, you learned many of the principles driving the development of Windows Phone, including the distinctive UI, the architectural convergence with Windows, and the importance of developers and apps. These principles will act as the foundation as you proceed through the remainder of the book and delve into the details of specific features.

You were also introduced to the Windows Phone developer tools and SDK, so with “Hello World” in the rearview mirror, it’s time to move on.

App Model and Navigation

This chapter focuses on the Windows Phone app model, including lifecycle events, as well as the page-based navigation model used by the Silverlight app framework. Users have come to expect that smartphones offer many of the same apps and overall user experience (UX) as a desktop computer. However, compared to a desktop computer, a phone has significant constraints in terms of memory, processor capacity, disk storage, screen size, input capabilities, and power.

The app platform and the underlying operating system both execute a number of services in the background, and several of these will be executing at any given period of time. Apart from services, there might also be background tasks running at the same time such as email sync or generic background agents.

Restricted processing power and system memory make it difficult for a phone to run more than one foreground app at the same time, as is common with desktop computers. By the same token, it's really not possible for the user to see multiple app windows on the phone screen at the same time, because the screen real estate is so constrained. On the other hand, the user expects to be able to run multiple apps and to switch seamlessly between them.

Windows Phone resolves these conflicting requirements by carefully managing system resources (primarily CPU and memory) so that priority is given to the app with which the user is directly interacting, while simultaneously making it possible for other background tasks to continue, albeit with a smaller set of resources.

As the user navigates between pages in an app, and between apps, the app platform surfaces a number of events. The developer can use these events to make his app more robust, and to integrate more closely with the system to provide a seamless UX. Note that the lifetime events in the native app model differ slightly, as discussed in Chapter 21, "C++ Development in Windows Phone 8."

The App Lifecycle

One of the advantages of the Windows Phone platform is that it is extremely consistent in the way it works. All apps follow the same broad guidelines, which leads to a very predictable UX. These guidelines are as follows:

- There can be multiple apps or agents running at the same time, but only one is active in the foreground, taking over the screen.

- Only the active app (or the user) can initiate navigation from one app to another.
- There can only be one active page at a time, and the only way to activate a page is to navigate to it.
- The user can always tap the hardware Start button to switch from the current app to the Start app.
- The user can always tap the hardware Back button to return to the previous screen or dismiss temporary UI. This navigates backward—either to the previous page in the current app or to the previous app if she is currently on the first page of an app.
- Whenever the user returns to an app after navigating forward, away from the app (including using a launcher or chooser), it should appear to be in the same state as when she left it, not as a fresh instance.



Note The Continuous Background Execution (CBE) feature introduced in Windows Phone 8 adds another dimension to the app lifecycle and navigation behavior. This is discussed in more detail in Chapter 16, “Location and Maps.”

There are several ways by which a user can start an app, including via a toast notification, from an extensibility point, or from the app list or a pinned tile on the Start screen. If the user navigates away from an app and if it is not a CBE app, the app will be either deactivated or closed. Specifically, if the user navigates away by pressing the Back key, it will be closed; if he navigates away from it by any other mechanism, such as by pressing Start or launching a chooser, it will be deactivated. The system maintains what is known as a *backstack* of recently deactivated apps as well as a minimal set of information about each app in the backstack so that it can quickly reactivate an app if called upon to do so.

Launching an app creates a new instance of the app and, by convention, always displays the same initial UX, such as a startup menu. Deactivated apps in the backstack can be reactivated by using the Back key or the Task Switcher. Reactivating apps is typically (but not always) faster than launching from scratch and preserves the app’s context rather than returning to the startup page.

All Store apps are single-instance, so starting a fresh copy of an app always removes any previous instance from the backstack. However, this is not the case with certain built-in apps, such as email, for which you can have multiple instances on the backstack.

To ensure a well-performing user interface (UI), the foreground app must be allocated the maximum amount of resources that the system can offer (after carving out a suitable reserve for the OS itself, drivers, and services). On a resource-constrained mobile device, the best way to ensure that this happens is to reclaim resources from apps that are not currently the foreground app.

Internally, an app can go through several logical states as it transitions from the foreground app to the backstack and final termination. From an app developer’s perspective, there are two lifetime scenarios to be handled:

- **The Closing case** This is when an app terminates and receives the *Closing* event. This is unambiguous and simple to handle: it happens when a user presses the hardware Back button from the first page in the app, which kills the app instance and the process itself.
- **The Deactivated case** This is when an app is moved to the background and receives the *Deactivated* event. This case is a little more complicated to handle. It happens when the user leaves your app in the backstack and the system has no knowledge as to whether the user might later return to the app. The app must save sufficient transient state to recreate its current UX in case the user returns to the app instance, even if the process has been terminated in the meantime. The app must also save enough persistent state so as not to lose critical user data if the user starts a new instance of the app (for example, after it falls off the backstack). You can further divide the Deactivated case into two scenarios:
 - **Tombstoning** The app is deactivated and the process is killed, but the app instance is maintained. There's nothing actively running or even sitting in memory, but the system remembers the app's critical details and can bring it back to life if needed.
 - **Fast app resume** The app is deactivated and then immediately reactivated, without being tombstoned in between.

When you create a Windows Phone app with Microsoft Visual Studio, the templates generate code that includes stub handlers in the *App* class for the four lifecycle events: *Launching*, *Activated*, *Deactivated*, and *Closing*. To take part in navigation events, you can also optionally override the *OnNavigatedTo* and *OnNavigatedFrom* methods in your derived *Page* class or classes.

This is your opportunity to take part in your app's lifetime management and make informed decisions about when to perform certain operations. At a simple level (and for quick operations), the appropriate actions that your app should take for each lifecycle event (or virtual method override) are summarized in Table 2-1.

TABLE 2-1 Expected Behavior During Lifecycle Events

Class	Event/Override	Suitable Actions
<i>App</i>	<i>Launching</i>	A fresh instance of the app is being launched. You should not do anything here that might slow down the launch.
<i>App</i>	<i>Deactivated</i>	The app is being deactivated and the process can be killed. In sequences for which this event is raised, this is your last opportunity to save app state.
<i>App</i>	<i>Activated</i>	The app is activating. This is your first opportunity to reload app state.
<i>App</i>	<i>Closing</i>	The app is closing. In sequences for which this event is raised, this is your last opportunity to save any unsaved state and clean up. Be aware that you have limited time in which to do anything here.
<i>Any Page</i>	<i>OnNavigatedFrom</i>	The user is navigating away from this page. This is your last opportunity for this page to save transient page state.
<i>Any Page</i>	<i>OnNavigatedTo</i>	The user is navigating to this page. At this time, you can load page state.

See Table 2-2, later in this chapter, for implementation details of how and where to save and load state. A naïve developer might take the *Launching/Activated/OnNavigatedTo* methods as a good time to perform initialization, and the *OnNavigatedFrom/Deactivated/Closing* events as a good time to persist state and clean up, and leave it at that. As just indicated, it is true that these trigger points are your first or last opportunity to perform such operations. However, a more sophisticated developer will understand that although you can perform minimal initialization/termination functionality in these events, you should also adopt a model in which you distribute costly operations to normal running time, outside the critical lifecycle and navigation events. Again, the reason for this is the CPU/memory constraints on a mobile device.

For example, it is a Store certification requirement that the app must show its first screen within 5 seconds of launch, and be responsive to user input within 20 seconds. App initialization can be done in the *Launching* event handler (or even in the *App* class constructor), but you should limit this to only essential work and defer as much as possible to a later point to provide the best UX. You must also complete activation, deactivation, and navigation within 10 seconds or you will be terminated. So, the various lifecycle events give you an opportunity to take action, but you should design your app so that whatever you do in these events, you do it quickly. The time to do lengthy operations is during normal running, not when you're handling a lifecycle event. Thus, while it might be tempting to persist all of your app state in *Deactivated*, a smart developer will persist state incrementally throughout the session, and not leave it all to the end (unless of course the state and any related computations are very small). Furthermore, your app must be robust such that it can handle the possibility that anything you do write in your *Deactivated* handler might not complete before you're shut down, which risks leaving incomplete or even corrupt data to read when you start again.

If you need to load content, you should do this "just in time." That is to say, just before it is actually needed, and no sooner. If you need to save content, you should do this as soon as possible. This model typically means loading only enough content for the current page you are on, and saving that content when you navigate away. The data model used by an app should support this incremental load/save model, not a monolithic structure that must be serialized/deserialized in one go, because it clearly won't scale as the user adds more and more data over time.

Normal Termination

When the user taps the Back button from the first page of an app, the system raises a *Closing* event on the app. The app's hosting process is then terminated. The app lifecycle sequence, from initial launch, through loading and navigating to the app's first (or only) page, to final termination is illustrated in Figure 2-1.

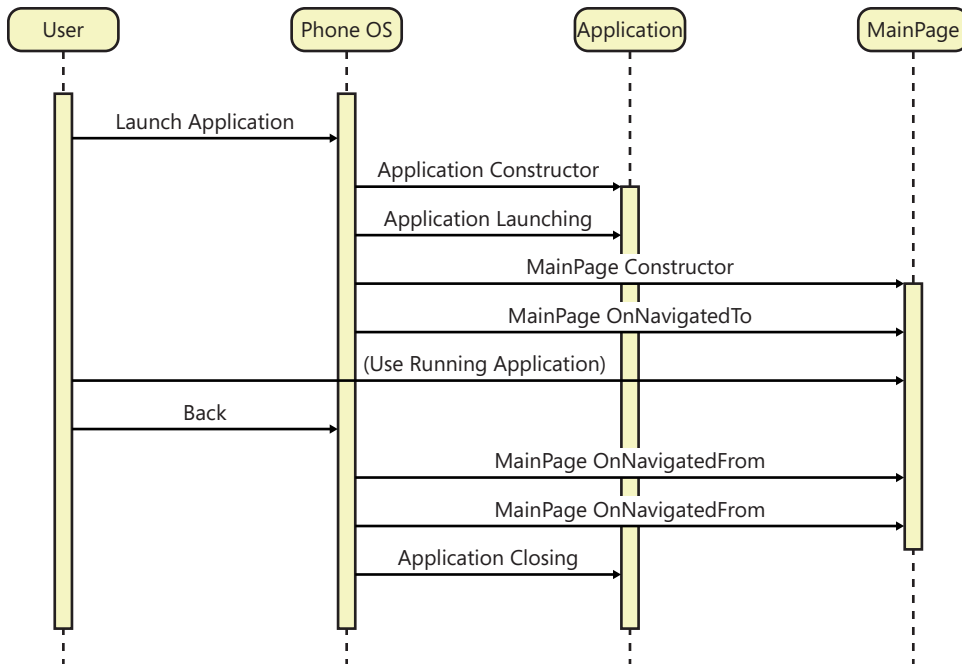


FIGURE 2-1 A diagram of the closing (normal termination) sequence.

App Deactivated—Fast-App Resume

The second lifetime scenario (see Figure 2-2) is when the user runs an app and then performs some action that navigates forward away from the app; for example, by tapping the Start button. In this sequence, the app process is not terminated; it is deactivated (and the system sends it a *Deactivated* event), and it remains in memory, taking up space. This is what makes it possible for the system to reactivate and resume the app quickly, if and when the user navigates back to it.

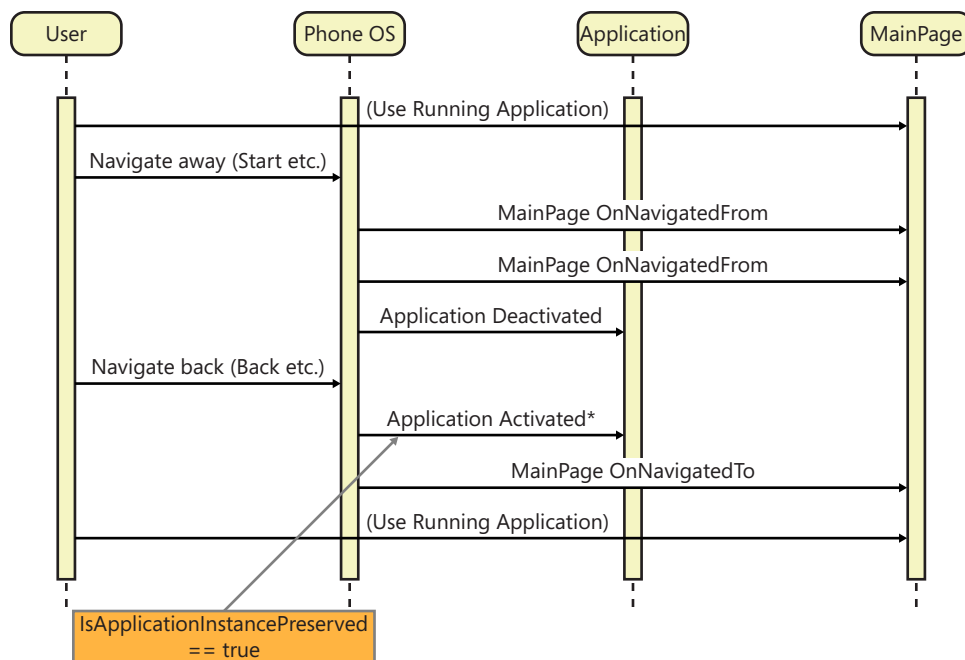


FIGURE 2-2 This diagram portrays the deactivation (fast resume) sequence.

The deciding factor here is performance. It is not efficient for the app platform to terminate the process and bring up a new one if the old one is still viable.

App Deactivated—the Tombstone Case

The tombstone case (see Figure 2-3) is a variation on the deactivated case, in which the app's hosting process is terminated. The app instance is still valid, because the system keeps any saved state information such that it can quickly reactivate the app (and send it an *Activated* event) if the user subsequently navigates back (via the Back button or the task switcher) to go back to it.

When an app is terminated in this way, all of the resources that it was consuming (CPU and memory) are taken away from it and made available to other processes. The system retains the barest minimum it needs to be able to reactivate the app at a later point, should it be called upon to do so. There will be an entry for the app in the backstack, including a note of the page within the app that the user was last viewing, the intra-app page backstack, and some limited transient state stored by the app itself (such as the state of UI controls).

If the tombstoned app is later reactivated, the lifecycle events are similar to the fast-resume case, except that the app and page constructors are called again.



Note The exact sequence of events varies according to circumstances. There are only two guarantees for a given app instance: *Launching* always happens exactly once, and *Activated* is always preceded by *Deactivated*.

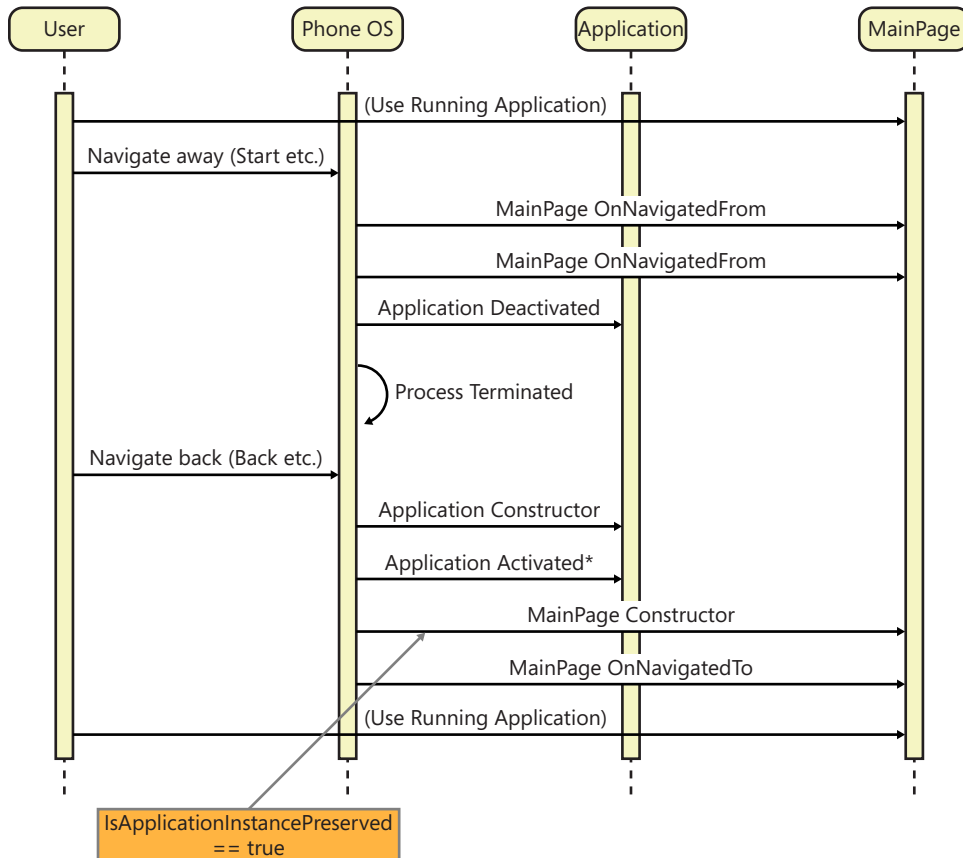


FIGURE 2-3 A typical tombstone sequence.

The sequence diagrams in Figures 2-2 and 2-3 show that there are two critical differences:

- In the fast app resume scenario, the app is maintained in memory, which means that the various app objects are not destroyed after deactivation; therefore, there is no need to run the *App*, *MainPage*, or other constructors upon subsequent activation. In the tombstone scenario, the app's memory is reclaimed by the system, so constructors must therefore be run again when the user switches back to the app.
- In the fast app resume scenario, the *IsApplicationInstancePreserved* property is *true* on *Application.Activated*, whereas in the tombstone scenario, the *IsApplicationInstancePreserved* property is *false* on *Application.Activated*.

The diagrams make the simplification that the user was on a page called *MainPage* when she navigated away from the app. In fact, the behavior holds true regardless of which page she was on. The system keeps track of the page she was on, and then upon reactivation, constructs the page, if necessary (that is, if the app was tombstoned), and invokes the *OnNavigatedTo* method.

The system retains only five apps on the backstack, including the one that is currently active. As soon the user launches the sixth app, the app at the beginning of the backstack (that is, the one that was used least recently) is discarded completely. In this situation, the discarded app would have received a *Deactivated* event when the user navigated away from it. It does not receive any further events, and there is no indication when it is discarded from the backstack. If memory pressure increases to the point at which the system needs to reclaim memory from deactivated apps, it will first start tombstoning apps from the end of the back-stack (but they remain on the backstack and can be reactivated by the user).

A further consideration is resource management. Figure 2-4 shows the sequence when an app is deactivated. In this state, you don't want it consuming resources, especially hardware resources such as sensors, and most especially resources such as the camera, which can only be used by one app at a time. The standard *OnNavigatedFrom* and *Deactivated* events are your opportunity to relinquish resources. However, if you do not proactively release resources, the framework will do the job for you. It's best to keep control of this yourself so that you can track which resources you were using and reinstate them as needed, if the app is later reactivated. When an app is deactivated, its resources are detached, and threads and timers are suspended. The app enters a deactivated state in which it cannot execute code, it cannot consume runtime resources, and it cannot consume any significant battery power. The sole exception to this is memory: the deactivated app remains in memory. Note that the work of detaching resources and suspending timers and threads is done by the platform for the app, as part of the app framework, but this does not surface any events to the app code itself.

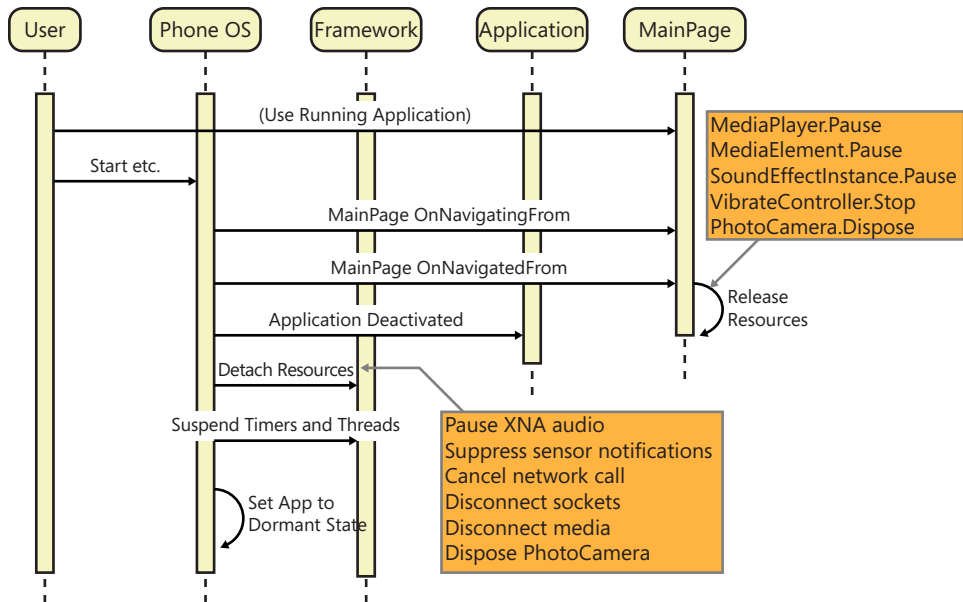


FIGURE 2-4 Bringing an app to the deactivated state.

Conversely, when an app is reactivated from the deactivated state, the framework resumes timers and threads, and reattaches some (but not all) resources that it previously detached (see Figure 2-5). The developer is responsible for reconnecting/resuming media playback, HTTP requests, sockets, and the camera.

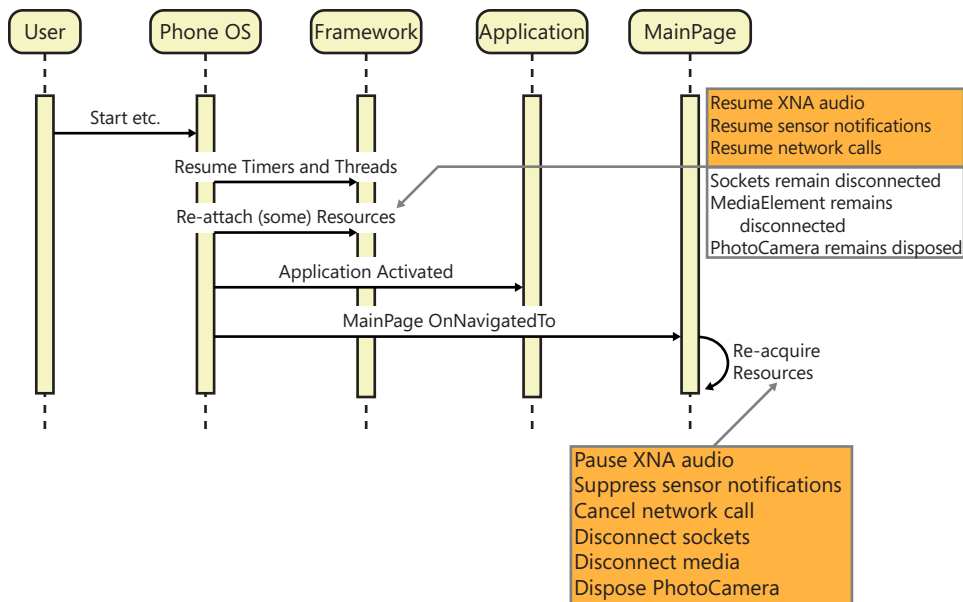


FIGURE 2-5 Resuming an app from the deactivated state.



Note There are two cases for which no lifecycle events are raised: when the app throws an exception which is not handled, and when the app calls *Application.Terminate*. If there is an *UnhandledException* handler in place—which is something the Visual Studio templates insert by default—code execution will jump there. The default handler put in place by Visual Studio merely checks to see if the app is being debugged, and if so, it invokes *Debugger.Break* to break into the debugger. If the app is not being debugged, the exception is then handled in the app platform itself, and the app is immediately terminated and removed from the backstack. *Application.Terminate* causes a rude termination—no events are sent to the app, and the process is killed instantly. This is not generally a useful mechanism for you to use.

The various sequences of lifecycle and navigation events can be challenging to keep straight in your mind. To help internalize the behavior, you can try this hands-on exercise. Simply create a new app and put *Debug.WriteLine* statements into each of the interesting events or method overrides, as shown in the example that follows. You can see an example of this in *TestLifecycle* solution in the sample code.

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    Debug.WriteLine("Application_Launching");
}
```

The methods in the *App* class to tag like this include the *App* constructor and the handlers for the *Launching*, *Activated*, *Deactivated*, and *Closing* events. For the *MainPage* class, you can add a *Debug* statement to the constructor. You can then also override the virtual methods *OnNavigatedTo* and *OnNavigatedFrom*, as shown in the following:

```
public MainPage()
{
    Debug.WriteLine("MainPage ctor");
    InitializeComponent();
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Debug.WriteLine("OnNavigatedTo");
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    Debug.WriteLine("OnNavigatedFrom");
}
```

Build the project, and then try the following operations:

1. Start the app in the debugger.
2. Tap Start in the emulator to navigate forward away from the app.
3. Tap Back to navigate back to the app.
4. Tap Back again to navigate back out of the app altogether.

This should result in the following output sequence:

```
App ctor
Application_Launching
MainPage ctor
OnNavigatedTo
OnNavigatedFrom
Application_Deactivated
Application_Activated
OnNavigatedTo
OnNavigatedFrom
Application_Closing
```

You should also then set the Tombstone Upon Deactivation While Debugging setting in your project properties (on the Debug tab) and repeat these operations. This should produce a slightly different output sequence. In particular, you will see the *App* and *MainPage* constructors invoked again.

```
App ctor
Application_Launching
MainPage ctor
OnNavigatedTo
OnNavigatedFrom
Application_Deactivated
App ctor
Application_Activated
MainPage ctor
OnNavigatedTo
OnNavigatedFrom
Application_Closing
```

Obscured and Unobscured

Activation/deactivation happen when the user navigates away from the app and when the app invokes Launchers and Choosers. On the other hand, some external operations merely result in the app becoming temporarily obscured. In this scenario, there is no *NavigatedFrom* or *Deactivated* event. Instead, the system raises an *Obscured* event. A common example of such an external operation is when a notification for an incoming call or a reminder is received.



Note An incoming Short Message Service (SMS) toast does not raise the *Obscured* event.

Obscuring does not cause navigation away from the app—the app continues to run in the foreground—it's just that some higher-priority UI is obscuring the app's UI. Note that this does not cause a frozen app display; the app does actually continue running, executing whatever operations it was performing when it was interrupted.

If you want to handle the *Obscured* and *Unobscured* events, you attach event handlers to the *RootFrame* object. You should also do this in the *App* class constructor, as shown in the following example (and demonstrated in the *TestObscured* solution in the sample code):

```
public App()
{
    UnhandledException += Application_UnhandledException;
    InitializeComponent();
    InitializePhoneApplication();
    InitializeLanguage();

    RootFrame.Obscured += RootFrame_Obscured;
    RootFrame.Unobscured += RootFrame_Unobscured;
}

private void RootFrame_Obscured(object sender, ObscuredEventArgs e)
{
    Debug.WriteLine("RootFrame_Obscured");
    if (e.IsLocked)
    {
        Debug.WriteLine("IsLocked == true");
    }
}

private void RootFrame_Unobscured(object sender, System.EventArgs e)
{
    Debug.WriteLine("RootFrame_Unobscured");
}
```

The *Obscured* event does not imply that the entire app UI is obscured. In many cases, including for an incoming phone call, the UI is only partially obscured (at least until the call is accepted). Another scenario in which this event is raised occurs when the phone lock screen is engaged. An app can determine whether this is the cause of the obscuring by testing the *IsLocked* property on the *ObscuredEventArgs* object passed in as a parameter to the *Obscured* event handler, as shown in the preceding example.

Note that the app will not always receive a matching *Unobscured* event for every *Obscured* event. For example, the app does not receive matching events for the scenario in which a user navigates away from the app by pressing the Start button. It's also true in the case for which the *Obscured* event is the result of the lock screen engaging. When the user later unlocks the screen, the app is not sent an *Unobscured* event. So, if you get an *Obscured* event and then the lock screen engages, your app will be deactivated (sent the *Deactivated* event) and then later reactivated.

If you disable the lock screen, you obviously won't get any *Obscured* events for this case, because the screen will not lock. You can disable the lock screen by setting *UserIdleDetectionMode* to *Disabled*, as demonstrated in the code that follows. This statement is generated in the *App* constructor by the

standard Visual Studio project templates. The Visual Studio code generation is intended only for debugging scenarios. In general, you should use this setting only after very careful consideration; it is legitimate only for an app that absolutely must continue running, even when the user is not interacting with the phone.

```
PhoneApplicationService.Current.UserIdleDetectionMode = IdleDetectionMode.Disabled;
```

The term “interacting with the phone” normally implies touching the screen, but another common use for this setting is games that are accelerometer driven as opposed to touch driven. In that case, the user is clearly interacting with the phone, even if he’s not touching the screen. If you actually need to use the feature in normal situations, you should not set it globally at startup. Instead, you should turn it on only when the user is actively using the feature that requires non-locking, and then turn it off again as soon as he is done with that activity. For example, in a game, you should not disable lock while the user is on a menu screen or has already paused the game; you would turn it on only while he is actively playing the game.

A related setting is *ApplicationIdleDetectionMode*. The system’s normal assumption is that if an app is running and the lock screen engages, it is reasonable to deactivate the app. By disabling *ApplicationIdleDetectionMode*, the app can continue to run under screen lock. If you do disable *ApplicationIdleDetectionMode*, the system does not deactivate idle apps. In this case, when the user eventually unlocks the screen again, the app receives the *Unobserved* event.

```
PhoneApplicationService.Current.ApplicationIdleDetectionMode = IdleDetectionMode.Disabled;
```

If you do disable *ApplicationIdleDetectionMode*, you should also do as much as possible to minimize battery consumption. Specifically, you should stop all active timers, animations, use of the accelerometer, GPS, isolated storage, and network. You would then use the *Unobserved* event to reinstate any of those things, as appropriate.



Note In Windows Phone 7.x, the technique of running under lock was used for such features as background audio and background location. Those are now first-class citizens in the platform, so they no longer require running under lock. In Windows Phone 8, therefore, there are very few valid scenarios for which you would use this technique .

The Page Model

Apart from Direct3D games (and legacy XNA games), Windows Phone apps employ a page-based model that offers a UX that is similar in many respects to the browser page model. The user starts the app at an initial landing page and then typically navigates through other pages in the app. Each page typically displays different data along with different visual elements. As the user navigates forward, each page is added to the in-app page backstack (also called the *journal*) so that she can always navigate backward through the stack, eventually ending up at the initial page. Although the inter-app backstack of app instances is limited to five apps, there is no hard limit to the number of intra-app

pages that can be kept in the page backstack. However, in practice it is uncommon to have more than six or so pages in the backstack; any more than that degrades the UX. Usability studies show that the optimal number is somewhere between four and ten. That doesn't mean that an app can't have dozens or even scores of pages, it just means that the navigation tree should keep each branch relatively short. You should also remember to clean up unused resources on pages in the backstack (such as images or large data context items) because they continue to consume memory and can often be recreated cheaply.

The app can support forward navigation in a wide variety of ways: through *HyperlinkButton* controls, regular *Button* controls, or indeed any other suitable trigger. An app can use the *Navigation Service* to navigate explicitly to a relative URL. Relative URLs are used for navigation to another page within the app. Absolute URLs can be used with *HyperlinkButton* controls to navigate to external web pages via the web browser (Internet Explorer). Backward navigation should typically be done via the hardware Back button on the device. If the user navigates back from the initial page, this must terminate the app, as depicted in Figure 2-6.

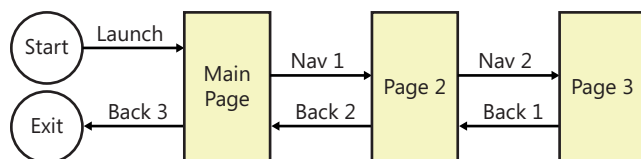


FIGURE 2-6 An overview of intra-app page navigation.

Users can also navigate forward out of an app and into another app. This can be achieved both from within the app via links with external URLs (which use the web browser) or by directly invoking Launchers and Choosers. At any time, the user can navigate away from the app by pressing the hardware Start or Search buttons on the device. Whenever the user navigates forward out of an app, that app is added to the system's app backstack. As the user navigates backward from within an app, she would move backward to that app's initial page and then back out of the app to the previous app in the backstack. Eventually, she would navigate back to the beginning of the backstack. Navigating backward from there takes her to the Start screen.

Here's a simple example. Suppose that a user launches an app. This navigates first to the default page in the app. The user then navigates to page 2 within the app. This is the second navigation. Then, the user taps the Start button. This navigates to the default page in the Start app. From there, the user launches a second app, which navigates to the default page in that second app.

Pressing the Back button from the initial page of the second app takes the user back to the Start app. Pressing Back again takes her back to page 2 in the first app. She would continue to navigate backward within the in-app page backstack until she arrives at the initial page in the first app. After that, pressing Back again takes her back to the Start app, and so on. The user's perspective of this workflow is illustrated in Figure 2-7. It's important to clarify that, internally, each app has its own internal page backstack, which is separate from the platform's app backstack.

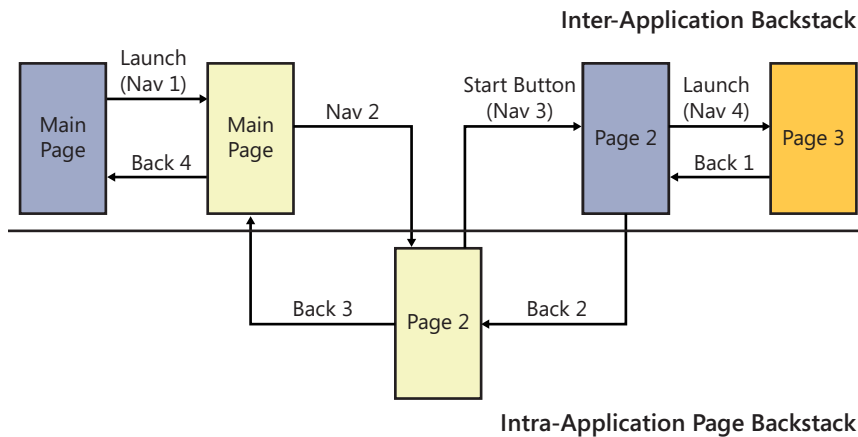


FIGURE 2-7 The inter and intra-app navigation model.

Page Creation Order

As part of its backstack management, the app platform keeps track of which page (in a multipage app) the user was on when he navigated away. If an app was tombstoned such that the pages need to be reconstructed if the user navigates back to the app, the order of page creation is not necessarily the same as the original order of creation. For example, if the user is on the second page when he navigates away from the app and then goes back, he will end up going back to the second page. This causes the page to be re-created. If he subsequently navigates back to the main page from there, at that point, the main page will be re-created. So, the order of page creation in the app can change according to circumstances. The main or initial page in an app is not always constructed first. In fact, if the user has navigated forward through the in-app page hierarchy and then forward to another app that uses a lot of memory (causing the original app to be tombstoned), and then navigates back to the first app, the pages will be constructed in reverse order. Figure 2-8 shows this for the tombstone behavior; Figure 2-9 illustrates it for the non-tombstone behavior. You can verify this behavior by using the *PageCreationOrder* solution in the sample code.

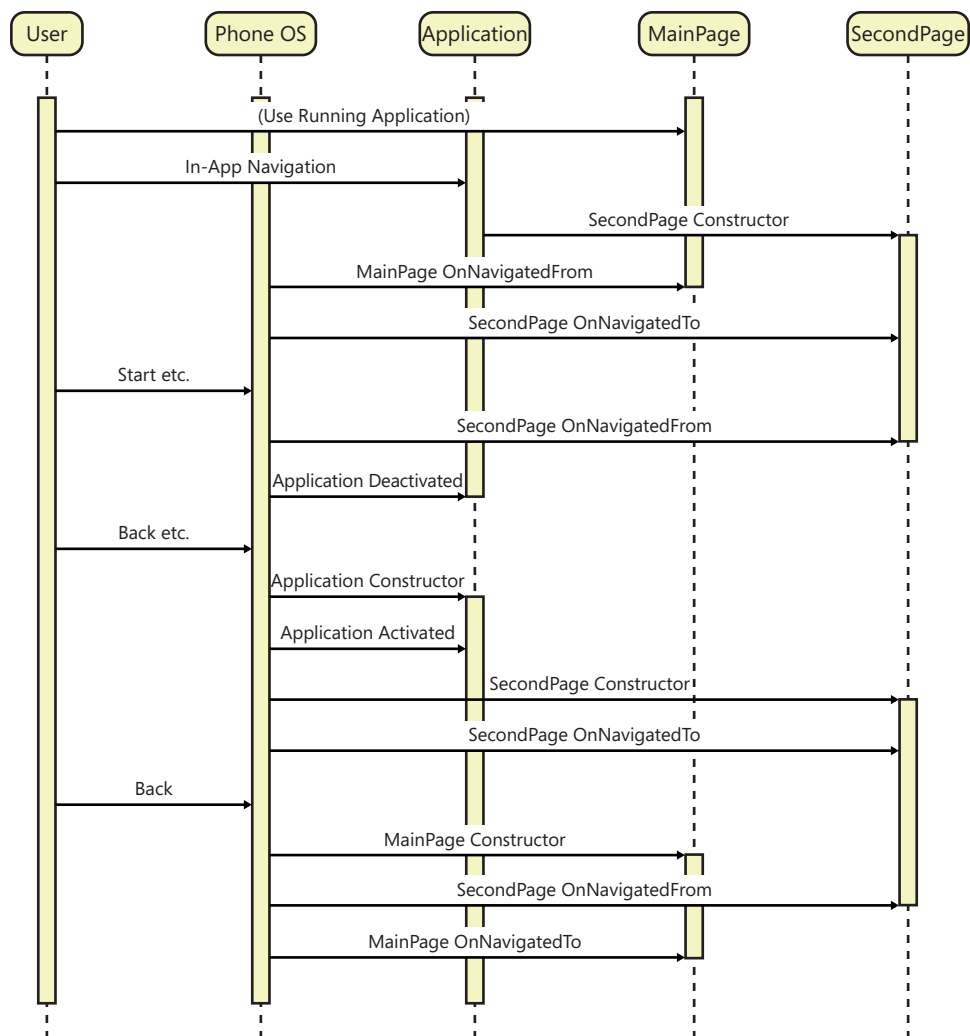


FIGURE 2-8 Unexpected page creation ordering (tombstone case). Here, *SecondPage* is constructed before *MainPage*.

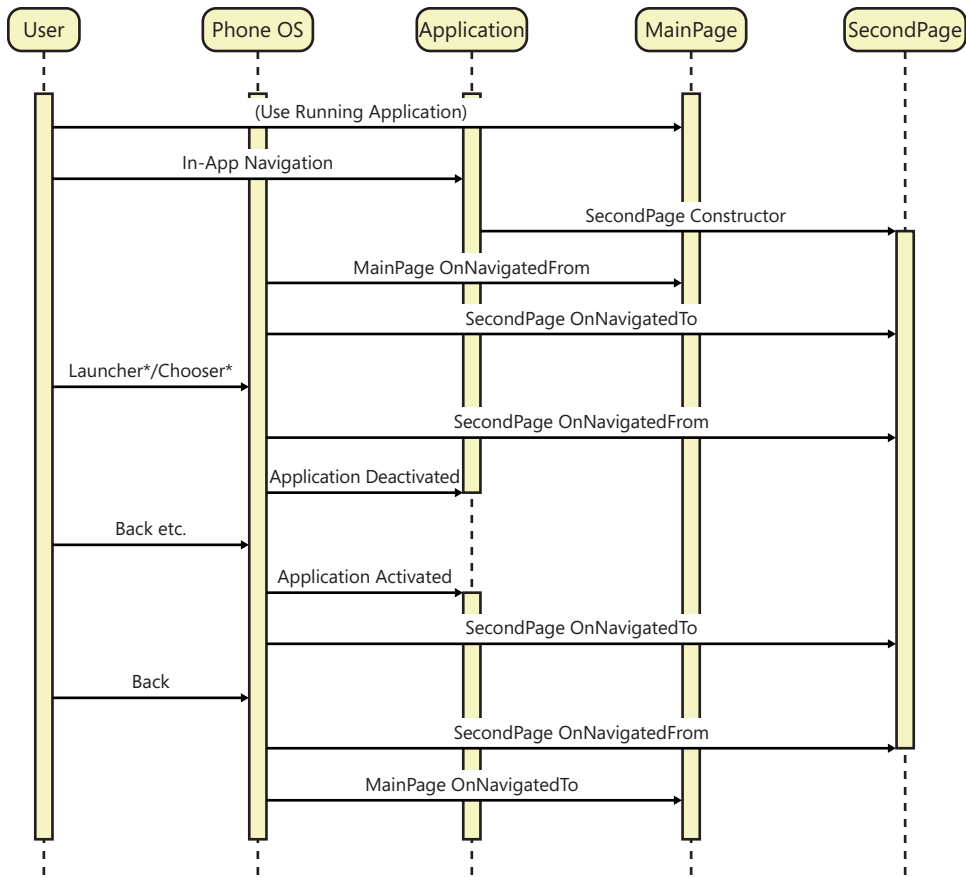


FIGURE 2-9 Page creation ordering (non-tombstone case) with no fresh page construction.

One consequence of this is that the app should not rely on a hierarchical relationship between pages in terms of object lifetime. That is, don't construct objects in Page X that are required in Page Y. Instead, all pages should be responsible for maintaining their own private state, and any state that is used across multiple pages should be held in the viewmodel (see Chapter 4, "Data Binding and MVVM," for details on viewmodels). Furthermore, the viewmodel should be accessible to all pages at all times, with predictable finite lifetime characteristics, which pretty much means it should be held in the *App* class (or be declared statically and be exposed via a singleton pattern).

To ensure consistent state in the face of navigation requires that you understand the navigation sequences and that you do work to persist state where necessary.

Navigation and State

The app model presents a UX of multiple apps running concurrently, and the navigation model supports this by providing a framework for the user to navigate between pages within an app as well as between apps. At both the page level and the app level, the system raises navigation events to which you can listen to maintain your app’s state. As the user navigates away from one of your pages or from your app altogether, you can persist any state you might need. Later, as the user navigates back to that page, or to your app, you can restore that state. All of this helps to support the UX of seamless switching between pages and between apps.

It is important to have a good understanding of the navigation model so that your app can integrate seamlessly with the phone ecosystem and behave in a manner that is consistent with other apps and with users’ expectations. This section examines the navigation model as well as the events and methods with which you can take part in the model to provide the best possible UX.

In the context of app navigation (both intra-app and inter-app), app state can be divided into three categories, as summarized in Table 2-2.

TABLE 2-2 Categories of App and Page State

Type of State	Description	Guidelines
Transient page state	The state specific to a page that does not need to persist between runs of the app; for example, the value of uncommitted text changes or the visual state of a page.	Store this in the <i>PhoneApplicationPage.State</i> property in the <i>NavigatedFrom</i> event, and retrieve it in the <i>NavigatedTo</i> event.
Transient application state	The state that applies across the app that does not need to persist between runs of the app; for example cached web service data.	Store this in the <i>PhoneApplicationService.State</i> property when the app handles the <i>Deactivated</i> event, and retrieve it in the <i>Activated</i> event.
Persistent state	The state of any kind that needs to persist across runs of the app—essentially, anything that would upset the user if you didn’t save it.	Store this to persistent storage incrementally during the lifetime of the app. Your last chance to do this is during the <i>Deactivated</i> and <i>Closing</i> events (the app might not return from <i>Deactivated</i> , and will not return from <i>Closing</i>), but you should not leave all persistence to these events. Also persist this during the <i>OnNavigatedFrom</i> call (for any state modified inside a page).

Navigation consists of a pair of notifications: *OnNavigatedFrom* and *OnNavigatedTo*. The former is sent to a page when it is no longer the current page; the latter is sent to a page when it becomes the current page. This is true both within an app (Page A replaces Page B) and across apps (App X replaces App Y). A page can never be navigated “from” without first being navigated “to,” but the inverse is not true. A single page instance can receive an arbitrary number of to/from notifications if the user navigates into and out of the same page repeatedly.

The situation with page constructors is more complicated. This behavior can be summarized as follows:

- Navigating forward (through a hyperlink or a call to *NavigationService.Navigate*) always constructs the target page. Even if an existing instance of the page exists on the backstack, a new one will be created (this differs from desktop Microsoft Silverlight and Windows 8 XAML, in which you can configure it to reuse instances).
- Navigating backward (via the Back button or *NavigationService.GoBack*) will not construct the target page if it already exists (for instance, the process has not been tombstoned since you last visited that page). If the app has been tombstoned and the page instance does not exist, it will be constructed.

It should be clear from this that the critical times to consider state management are in the *OnNavigatedTo* and *OnNavigatedFrom* handlers, not in the page constructor. Furthermore, it is sometimes useful to handle the *Loaded* event for a page, or even the *LayoutUpdated* event, but neither of these are suitable places to perform state management. Both of these are called more often than you might expect and are not intended for state management operations.

App State

The *PhoneApplicationService.State* property is an *IDictionary* object that the OS saves. All you have to do is write to the collection (or read from it) at any time during execution, or in the appropriate event handler. This dictionary is not persisted across separate instances of the app; that is, across fresh launches from the Start page, and so on. The *LifecycleState* solution in the sample code demonstrates the behavior, as demonstrated in Figure 2-10.

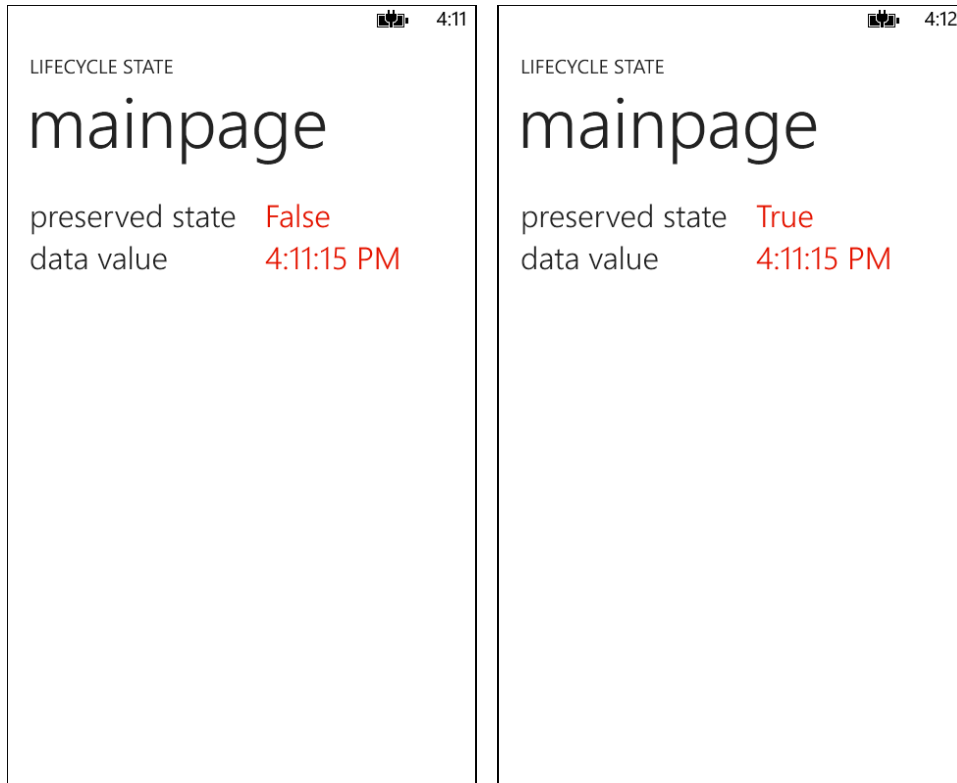


FIGURE 2-10 The LifecycleState solution demonstrates loading and saving app state: a fresh launch on the left, and a reactivated instance on the right.

In this app, the *App* class exposes a *MainViewModel* property. ViewModel classes in general are discussed in detail in Chapter 4; for now, you can consider a viewmodel as a technique for encapsulating data, and a property of your *ViewModel* class is normally maintained in the *App* class. In this example, the *MainViewModel* class is trivial because it contains just one string data member and a method for initializing that data:

```
public class MainViewModel
{
    public string Timestamp { get; set; }

    public void LoadData()
    {
        Timestamp = DateTime.Now.ToLongTimeString();
    }
}
```

An object of this type is exposed in the *App* class and initialized in the property *get* accessor, as shown in the example that follows. Keep in mind that this approach assumes that the *LoadData* call doesn't do a lot of work. If you do have a lot of data to load, you would want to build a separate *LoadData* method that loaded data asynchronously and incrementally.

```

private static MainViewModel viewModel;
public static MainViewModel ViewModel
{
    get
    {
        lock (typeof(App))
        {
            if (viewModel == null)
            {
                viewModel = new MainViewModel();
                viewModel.LoadData();
            }
        }
        return viewModel;
    }
}

```

The *App* class also declares a string for the name of the state data that will be persisted in the *PhoneApplicationService.State* collection, and a *bool* flag for tracking whether or not this instance of the app is a fresh instance or one that was preserved on the backstack from an earlier launch.

```

private const string appStateName = "AppViewModel";
public static bool IsAppInstancePreserved { get; set; }

```

In the *Deactivated* event handler, the app persists the *viewModel* field in the *PhoneApplicationService.State* collection. At the same time, it also writes the same value out to isolated storage, using the *IsolatedStorageSettings.ApplicationSettings* collection. The idea in the example that follows is that this data does need to be persisted across instances of the app, although this will not always be the case. In the *Closing* event handler, the app only persists to isolated storage, because you can be sure at this point that the app is being terminated, so there's no point writing to *PhoneApplicationService.State*.

```

private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    PhoneApplicationService.Current.State[appStateName] =
        viewModel;
    IsolatedStorageSettings.ApplicationSettings[appStateName] =
        viewModel.Timestamp;
    IsolatedStorageSettings.ApplicationSettings.Save();
}

private void Application_Closing(object sender, ClosingEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings[appStateName] =
        viewModel.Timestamp;
    IsolatedStorageSettings.ApplicationSettings.Save();
}

```

Conversely, in the *Activated* event handler, the app caches the value of the *IsApplicationInstancePreserved* property provided by the *ActivatedEventArgs* parameter and then tests this. If this is a preserved instance, the app then goes on to check whether the named state exists in the collection, and if so, retrieves it and overwrites the *viewModel* field, as presented here:

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    IsAppInstancePreserved = e.IsApplicationInstancePreserved;
    if (!IsAppInstancePreserved)
    {
        if (PhoneApplicationService.Current.State.ContainsKey(appStateName))
        {
            viewModel =
                PhoneApplicationService.Current.State[appStateName]
                as MainViewModel;
        }
    }
}
```

To complete the picture, the *MainPage* class overrides the *OnNavigatedTo* virtual method to retrieve the data properties from the *App* and set them into the UI elements.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    stateValue.Text = App.IsAppInstancePreserved.ToString();
    dataValue.Text = App.ViewModel.Timestamp;
}
```

Page State

Just as you can use *PhoneApplicationService.State* for persisting app state, so you can also use the *PhoneApplicationPage.State* property for persisting page state. The model is identical.

You can use the phone's *NavigationService* to navigate to another page. When the user navigates back from the second page, however, the second page is destroyed. If she navigates to the second page again, it will be recreated. The main page (that is, the entry point page) is not destroyed or recreated during in-app navigation; however, it might be destroyed/recreated if the user navigates away from the app and returns back to it. The sequence of creation and navigation across two pages in an app is shown in Figure 2-11.

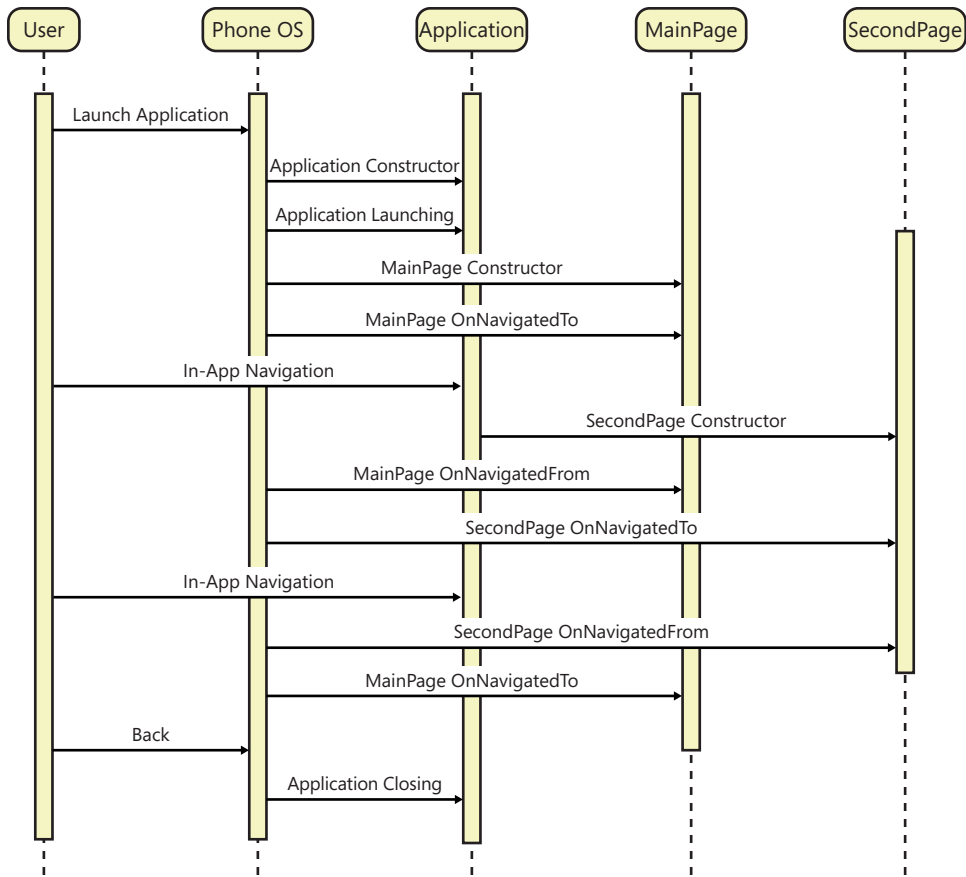


FIGURE 2-11 The sequence of page construction and navigation events.

Now, exactly how you should save and load page state depends what behavior you want. The normal behavior is that when the user navigates backward away from a page, that page is destroyed. So, the next time the user navigates forward to that page, it will be created from scratch. Typically, you use *PageNavigationState* if you only care about retaining state for the case when the user navigates forward, away from a page, and then back again to that page. On the other hand, you can use *Phone ApplicationService.State* if you want to preserve state that can be used by any page.

The *PageNavigationState* solution in the sample code demonstrates this behavior, as shown in Figure 2-12.

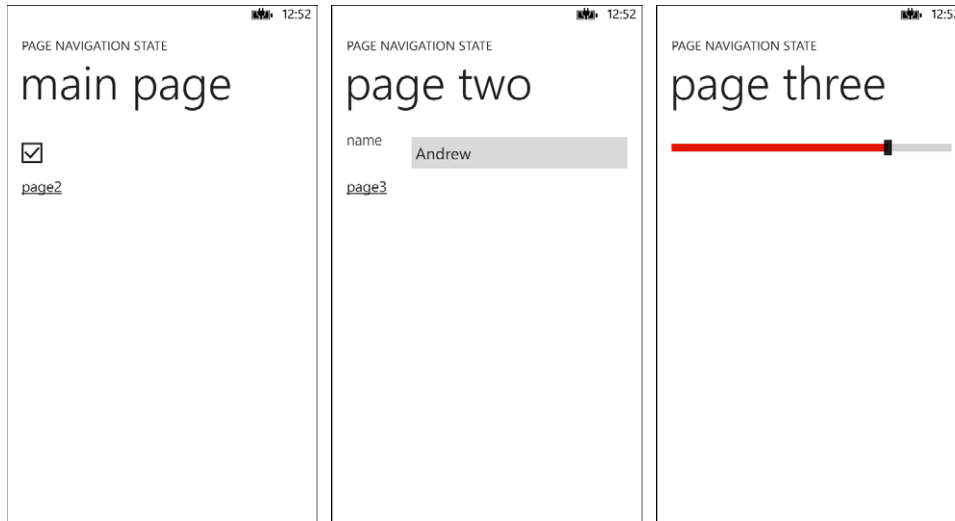


FIGURE 2-12 The PageNavigationState solution provides three pages. It persists state for each page in `PhoneApplicationService.State`.

The *MainPage* class declares a string field for the name of the state to be preserved in the *PhoneApplicationService.State* collection, and a *bool* flag to track whether or not this is a fresh instance of the page. In the *OnNavigatedFrom* override—invoked when the user navigates away from this page—the value of the UI element for this page (in this case, a *CheckBox*) is persisted in the *State* collection. Conversely, in the *OnNavigatedTo* override, the *CheckBox* state is restored from the collection if it exists. The *isNewInstance* flag is set to *true* in the page constructor, and to *false* in the *OnNavigatedTo* override. This way, state is restored if it is needed, and that code is skipped if it is not needed.

```
public partial class MainPage : PhoneApplicationPage
{
    private bool isNewInstance;
    private const string MainPageState = "CheckState";

    public MainPage()
    {
        InitializeComponent();
        isNewInstance = true;
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        PhoneApplicationService.Current.State[MainPageState] =
            checkState.IsChecked;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
```

```

{
    if (isNewInstance)
    {
        if (PhoneApplicationService.Current.State.ContainsKey(MainPageState))
        {
            checkState.IsChecked = (bool)
                PhoneApplicationService.Current.State[MainPageState];
        }
        isNewInstance = false;
    }
}
}

```

Very similar operations are performed in each of the other pages of the app, in each case using a page-specific name for the state to be stored in the app state collection.



Note The app platform enforces limits on storing state. No single page is allowed to store more than 2 MB of data, and the app overall is not allowed to store more than 4 MB. However, these values are far larger than anything you should ever use. If you persist large amounts of state data, not only are you consuming system memory, but the time taken to serialize and deserialize large amounts of data is going to affect your app's pause/resume time. In an extreme case, the platform might identify your app as being non-responsive during a critical transition phase, at which point it will rudely terminate the app. Upon rude termination, your app might well suffer data loss or corruption. The basic guidance should be to store only simple things in the state dictionaries; you should not, for example, store the entire cache of a database.

Also, any object that you attempt to store in these dictionaries must be serializable. If the app attempts to store an object that cannot be serialized, this will throw an exception during debugging and fail silently during production runtime. Finally, note that there will be issues if you have serialized a type in an assembly that isn't loaded at startup. In this case, you need to load that assembly artificially in your *App* constructor; otherwise, you get a deserialization error. Further details on serialization/deserialization are covered in Chapter 9, "Data: Isolated Storage and Local Databases."

Cancelling Navigation

Navigations that are initiated by the user by interacting with your app UI can generally be cancelled, whereas navigations initiated by the user interacting with hardware buttons or initiated by the system generally cannot be cancelled. It is common to provide navigation UI within your app, including *Hyperlink* and *Button* controls. However, there are scenarios for which, even though the user has gestured that he wants to navigate, you might want to intercept the request and prompt for confirmation. For example, if the user has edited a page or entered data, but he hasn't yet confirmed the new input or changes, you would prompt him to save first when he tries to navigate away.

One technique is to override the *OnNavigatingFrom* method. This provides a *NavigatingCancelEventArgs*, which exposes two useful properties. You can use the *Cancel* property to cancel the navigation, and the *IsCancelable* property to establish definitively whether an attempt to cancel will actually succeed. If you can't cancel the navigation, you would take other steps to handle the scenario (perhaps saving the user's input to a temporary file or other mitigating actions, depending on the context). You can see this at work in the *TestNavigating* solution in the sample code.

```
protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
    Debug.WriteLine("OnNavigatingFrom");

    if (e.IsCancelable)
    {
        MessageBoxResult result = MessageBox.Show(
            "Navigate away?", "Confirm", MessageBoxButton.OKCancel);
        if (result == MessageBoxResult.Cancel)
        {
            e.Cancel = true;
        }
    }
    else
    {
        Debug.WriteLine("Navigation NOT cancelable");
    }
}
```

Backstack Management

The user's navigation history is maintained within an app in a history list called the backstack. The backstack is managed as a last-in, first-out (LIFO) stack. This means that as the user navigates forward through the pages of an app, each page from which she departs is added to the stack. As she navigates back, the previous page in the navigation history is popped off the stack to become the new current page. The platform includes the following API support for working with the backstack:

- The *NavigationService* class exposes a *BackStack* property, which is an *IEnumerable* collection of *JournalEntry* objects. Each page in the backstack is represented by a *JournalEntry* object. The *JournalEntry* class exposes just one significant property: the *Source* property, which is the navigation URI for that page.
- The *NavigationService* class exposes a *RemoveBackEntry* method, which is used to remove the most recent entry from the backstack. You can call this multiple times if you want to remove multiple entries.
- The *NavigationService* class exposes an *OnRemovedFromJournal* virtual method, which you can override. This is invoked when the page is removed from the backstack, either because the user is navigating backward, away from the page, or because the app is programmatically clearing the backstack. When the user navigates forward, the previous page remains in the backstack.

Here's the sequencing of the APIs, in relation to the *OnNavigatedFrom* override. When the user navigates backward, away from the page, the methods/event handlers are invoked in the following order: first, the *OnNavigatedFrom* override; next, the *JournalEntryRemoved* event handler; and then finally, the *OnRemovedFromJournal* override.

In general, your app can control forward navigation—both to new pages within the app, and to external apps, Launchers and Choosers. Conversely, backward navigation is normally left under the control of the hardware Back button.

That having been said, you can modify the user's navigation experience such that going back doesn't necessarily always take her back to the previous page. To be clear, you can still only use the *NavigationService* to navigate forward to a specific URL, or back one page in the backstack. However, you can remove entries from the backstack—up to and including all entries—such that navigating back no longer necessarily takes the user back to the immediately preceding page.



Note The ability to manipulate the backstack is a powerful one that affords you a lot of flexibility, but it also gives you a way to break conformance with the Windows Phone app model. You should use this only after very careful consideration, and only if you're sure you can't avoid it..

Figure 2-13 shows the *NavigationManagement* solution in the sample code, which illustrates how you can manipulate the backstack.

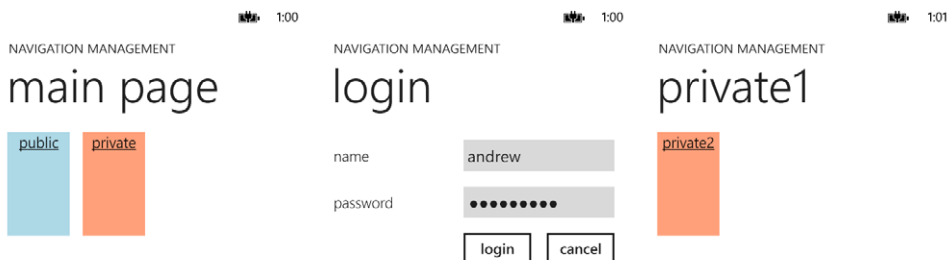


FIGURE 2-13 The *NavigationManagement* sample shows how you can manipulate the backstack.

On the *MainPage* of this example, the user can choose one of two links to navigate to the public or private pages within the app. If the user chooses the public page link, the app simply navigates to that page, as normal. On the other hand, if she chooses the private page link, the app navigates to a

login page. On the login page, if she taps the cancel button, this navigates back to the *MainPage*. If, instead, she taps the login button, this navigates forward to the first of the set of private pages. Realistically, there would be some login credential validation in there, but this example simply navigates without validation. On *PrivatePage1*, the user has a *private2* button, which navigates forward to the next private page in sequence. The idea here is that whatever page the user is on, when she presses the hardware Back button, this always skips the *LoginPage* page. The flow of navigation is presented in Figure 2-14.

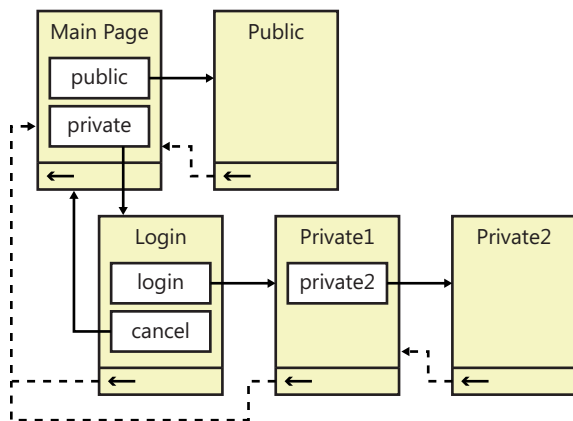


FIGURE 2-14 The navigation flow in the *NavigationManagement* sample app.

In *MainPage*, the two Click handlers for the *HyperlinkButton* controls invoke the *NavigationService.Navigate* method to navigate to explicit target pages. The public button goes to the *PublicPage*, whereas the private button goes to the *LoginPage*:

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

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

On the *LoginPage*, the login button navigates to *PrivatePage1*, whereas the cancel button invokes the *NavigationService.GoBack* method to go back one page in the backstack. The page in the backstack immediately before the *LoginPage* is always *MainPage*:

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/PrivatePage1.xaml", UriKind.Relative));
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    NavigationService.GoBack();
}
```

In *PrivatePage1*, the *private2* button navigates explicitly to *PrivatePage2*. Also in *PrivatePage1*, the override of *OnNavigatedTo* checks the value of the incoming *NavigationMode*. If the user navigated forward to this page, the previous page is removed from the backstack. This is done because the only way to navigate forward to this page is from the *LoginPage*, and the idea is that the user should not have to navigate back through the *LoginPage*. His perception is that the *LoginPage* is transient, and does not persist in the navigation backstack. Using this technique means that when he taps the hardware Back button, it navigates back to the page before the *LoginPage* because the *LoginPage* will no longer exist in the backstack.



Note It would be equally valid to implement the login page functionality via a true transient panel, using a *PopupControl* or a *ChildWindow* control. Such windows never persist in the backstack.

```
private void private2Link_Click(object sender, System.Windows.RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/PrivatePage2.xaml", UriKind.Relative));
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (e.NavigationMode == NavigationMode.New)
    {
        NavigationService.RemoveBackEntry();
    }
}
```

Navigation Options

The Windows Phone app platform offers a number of options for navigation—both within an app and externally. In addition to the *Navigate* and *GoBack* methods on the *NavigationService* class, an app can use the *NavigateUri* property of a *HyperlinkButton* and re-route navigation by either runtime interception or static URI mapping. There are also issues to consider with regard to navigation between pages across multiple assemblies, and options for passing data between pages during navigation.



Note One question that developers often ask is, “When would you ever need to invoke the *base.OnNavigatedTo/OnNavigatedFrom*?” When you use autocomplete to create an override for these virtual methods, Visual Studio generates a call to the base class version, which seems to imply that calling the base version is useful sometimes or always. In fact, this is simply an artifact of how autocomplete works; Visual Studio will always generate a base class call. With respect to *Page.OnNavigatedTo* and *OnNavigatedFrom*, you never need to invoke the base version, because the base version is empty, so you can always safely delete these calls.

Using *NavigateUri*

The normal way to use a *HyperlinkButton* is to specify the *NavigateUri* in XAML, and not to specify a *Click* event handler. With this approach, shown in the code that follows, the link is set up declaratively, and there is no event handler in the code-behind. Note, however, that this technique only supports forward navigation; there is no way to specify the equivalent of *NavigationService.GoBack*.

```
<!--<HyperlinkButton

    x:Name="privateLink" Content="private"

    Click="privateLink_Click"

    HorizontalAlignment="Center" VerticalAlignment="Top"

    FontSize="{StaticResource PhoneFontSizeMedium}"/>-->

<HyperlinkButton

    x:Name="privateLink" Content="private"

    NavigateUri="/LoginPage.xaml"

    HorizontalAlignment="Center" VerticalAlignment="Top"

    FontSize="{StaticResource PhoneFontSizeMedium}"/>
```

Pages in Separate Assemblies

From an engineering perspective, it is perfectly reasonable to divide a solution into multiple assemblies—possibly with different developers working on different assemblies. This model also works with Phone pages; thus, one or more pages for an app could be built in separate assemblies. This technique can also help with app startup performance because the code for the second and subsequent pages does not need to be loaded on startup, and the assembly load cost is deferred until the point when the user actually navigates to the second and subsequent pages, if ever. How does this affect navigation? Navigating back is always the same; *NavigationService.GoBack* takes no parameters. However, navigating forward requires a URI (either in the *NavigateUri* property or in the call to *Navigate*), and this URI must be able to be resolved. The simplest case is where the URI is relative to the app root, indicated with a leading slash. If the URI identifies a page that is in another assembly, the string format is as follows:

```
"/[assembly short name];component/[page pathname]"
```

So, for example, if you have *Page2* in a separate class library project named *PageLibrary*, to navigate to *Page2*, you would use this syntax:

```
NavigationService.Navigate(new Uri(
    "/PageLibrary;component/Page2.xaml", UriKind.Relative));
```

You can see this at work in the *NavigatingAssemblies* solution in the sample code.

Fragment and QueryString

To pass state values between pages on navigation, an app can use *PhoneApplicationService.State*, isolated storage, or state fields/properties on the App object. In addition, you can use *Fragment* or *QueryString*. Note that you cannot use *Fragment* to navigate within a page; you can only use it when navigating to another page. In general, *Fragment* is not particularly useful in the context of phone apps. Here is an example (the *NavigationParameters* solution in the sample code) in which the main page offers a choice between coffee and tea. The user's choice is passed down to *Page2* either via a *Fragment* or via a *QueryString*, depending on which button the user clicks. This is illustrated in Figure 2-14.

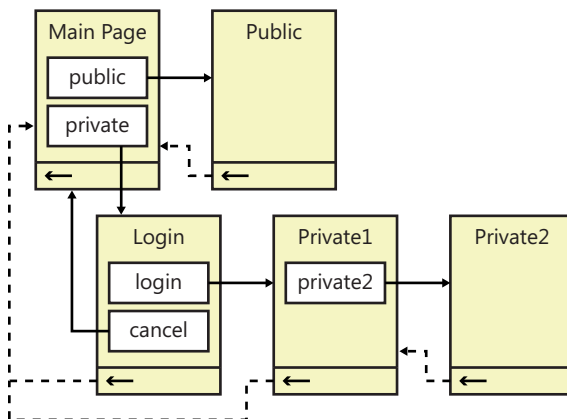


FIGURE 2-14 The *NavigationParameters* solution tests navigation by using *Fragment* and *QueryString*.

Page2 has a *ListBox* that is data-bound to one of two *ObservableCollection* objects (see Chapter 4 for details of this approach and to learn more about data binding).

```
private ObservableCollection<string> coffees =
    new ObservableCollection<string>();
private ObservableCollection<string> teas =
    new ObservableCollection<string>();

public Page2()
{
    InitializeComponent();
```

```

        coffees.Add("Blue Mountain");
        coffees.Add("Monsooned Malabar");
        coffees.Add("Sumatra");
        coffees.Add("Monkey Poo");
        coffees.Add("Tanzania Peaberry");

        teas.Add("Earl Grey");
        teas.Add("Darjeeling");
        teas.Add("Jasmine");
        teas.Add("Oolong");
        teas.Add("Chrysanthemum");
    }

```

To use a Fragment, you simply append a "#" (hash) character to the target URI, followed by the fragment value, as demonstrated here:

```

private void buttonPage2Fragment_Click(object sender, RoutedEventArgs e)
{
    if ((bool)radioCoffee.IsChecked)
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml#Coffee", UriKind.Relative));
    }
    else
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml#Tea", UriKind.Relative));
    }
}

```

On the navigation destination page (*Page2* in this example), you could override *OnNavigatedTo*, but the *Fragment* is not easily accessible in that method. You could parse the string in the *Uri* property of the *NavigationEventArgs*, but that would be a fragile approach. Instead, after *OnNavigatedTo* is called, the system calls *OnFragmentNavigation*, and it is here that you can get the *Fragment*. Note that setting the *ItemsSource* property of a *ListBox* is part of the data binding mechanism. This is explored in detail in Chapter 4. For now, you can ignore it.

```

protected override void OnFragmentNavigation(FragmentNavigationEventArgs e)
{
    switch (e.Fragment)
    {
        case "Coffee":
            SetItemsCoffee();
            break;
        case "Tea":
            SetItemsTea();
            break;
    }
}

private void SetItemsCoffee()

```

```

{
    listDrinks.ItemsSource = coffees;
    pageTitle.Text = "coffee";
    pageTitle.Foreground = listDrinks.Foreground
        = new SolidColorBrush(Colors.Brown);
}

private void SetItemsTea()
{
    listDrinks.ItemsSource = teas;
    pageTitle.Text = "tea";
    pageTitle.Foreground = listDrinks.Foreground
        = new SolidColorBrush(Colors.Green);
}

```

Conversely, you can provide a conventional query string, by appending a question mark "?" to the end of the URI, and then appending "key/value" pairs. Unlike *Fragment*, this gives you the ability to pass more than one value, in the format:

```
"/[pagename].xaml?[param1=value1]&[param2=value2]&[param3=value3]"
```

For example:

```

private void buttonPage2QueryString_Click(object sender, RoutedEventArgs e)
{
    if ((bool)radioCoffee.IsChecked)
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml?drink=Coffee", UriKind.Relative));
    }
    else
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml?drink=Tea", UriKind.Relative));
    }
}

```

If you use a query string, in the receiving page, this is provided as an *IDictionary* property on the *NavigationContext*, which itself is a property of the *Page* object. You can use both a query string and a fragment in the same URL, but this is not likely to be useful: it would require you to handle both, and to parse the URL in both cases to extract either one.

```

protected override void OnNavigatedTo(NavigationEventArgs e)

{

    string drinkType;

    if (NavigationContext.QueryString.TryGetValue(

        "drink", out drinkType))

```



```

{

    switch (drinkType)

    {

        case "Coffee":

            SetItemsCoffee();

            break;

        case "Tea":

            SetItemsTea();

            break;

    }

}
}

```

Note that although the *QueryString* property will not be null, it might be empty, so you should check for this before attempting to access its collection. Note also that, as with any *Dictionary* object, if you attempt to use an indexer that does not exist in the collection, this will throw an exception. So, rather than using the indexer, you can use the *TryGetValue* method, instead.

The *NavigationMode* and *IsNavigationInitiator* Properties

Overrides of *OnNavigatedTo* and *OnNavigatedFrom* are passed a *NavigationEventArgs* object. This exposes two useful properties: *Content*, set to the instance of the destination page; and *Uri*, which will be the URI of the destination page.

Windows Phone also exposes a *NavigationMode* property on the *NavigationEventArgs* object that is passed into the *OnNavigatedTo* and *OnNavigatedFrom* method overrides. The value of *NavigationMode* will typically be either *New* or *Back*, which identifies the direction of navigation. The *Back* value is self-explanatory; if the value is *New*, this indicates that this is a forward navigation. The *NavigationMode* type includes the values *Refresh* and *Reset*, which are discussed in Chapter 16. There's also a *Forward* value, but this is not used in Windows Phone. Typically, you would perform some conditional

operation based on this value, such as saving or restoring state. The following code snippet merely prints a string to the debug window. The app has two pages: *MainPage* and *Page2*, and the user can navigate back and forth between them.

```
public partial class MainPage : PhoneApplicationPage
{
    ... irrelevant code omitted for brevity.
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        Debug.WriteLine("MainPage.OnNavigatedTo: {0}", e.NavigationMode);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        Debug.WriteLine("MainPage.OnNavigatedFrom: {0}", e.NavigationMode);
    }
}

public partial class Page2 : PhoneApplicationPage
{
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        Debug.WriteLine("Page2.OnNavigatedTo: {0}", e.NavigationMode);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        Debug.WriteLine("Page2.OnNavigatedFrom: {0}", e.NavigationMode);
    }
}
```

When the app starts and the *MainPage* is loaded, the following output is produced:

```
MainPage.OnNavigatedTo: New
```

As the user navigates forward from *MainPage* to *Page2*, you will see the following debug output—the navigation is forward as far as both pages are concerned:

```
MainPage.OnNavigatedFrom: New
Page2.OnNavigatedTo: New
```

Then, as the user navigates back from *Page2* to *MainPage*, you would expect to see the following debug output—again, for both pages, the navigation is backward:

```
Page2.OnNavigatedFrom: Back
MainPage.OnNavigatedTo: Back
```

If the user is on *Page2* and then navigates forward out of the app by tapping the Start button, and then navigates back into the app again, you would see the following output (the *New* [forward] navigation out of the app, followed by the *Back* navigation back into the app):

```
Page2.OnNavigatedFrom: New
Page2.OnNavigatedTo: Back
```

Finally, consider another new property of both the *NavigationEventArgs* and the *NavigatingCancelEventArgs*: the *IsNavigationInitiator* property. This is a Boolean value that notifies you whether the navigation is from an external source; that is, the user navigated from outside the app into the app. This is designed so that you can avoid custom page-to-page animations in the case of an external navigation, because in that scenario, the platform will perform animation for you. In the following, you're going to modify the debug output statements to include this property value:

```
Debug.WriteLine("Page2.OnNavigatedTo: {0}, {1}", e.NavigationMode, e.IsNavigationInitiator);
```

Now, if the user starts the app (which loads *MainPage*), navigates internally to *Page2*, taps Start to navigate forward out of the app, taps the Back button to return into the app, and then finally, back from *Page2* to *MainPage*, you will see the output that follows. When the navigation is to or from an external source (including the initial launch of the app from the Start page), the value of *IsNavigationInitiator* is *False*. For internal navigation, the value is *True*.

```
MainPage.OnNavigatedTo: New, False
MainPage.OnNavigatedFrom: New, True
Page2.OnNavigatedTo: New, True
Page2.OnNavigatedFrom: New, False
Page2.OnNavigatedTo: Back, False
Page2.OnNavigatedFrom: Back, True
MainPage.OnNavigatedTo: Back, True
```

Re-Routing Navigation and URI Mappers

It is also possible to re-route navigation from one target page to another at runtime. For example, you might build an app in which the user can navigate to an *AccountInfo* page, but if the user is not logged in (or the login has timed out), you redirect him to a *Login* page. There are two distinct ways to achieve this kind of redirection.

The first approach is navigation re-routing, as demonstrated in the *ReRouting* solution in the sample code. Suppose that you have a requirement by which most days of the week, when the user asks to navigate to *Page2*, you sent him to a default *Page2a*, but on Tuesdays, you send him instead to *Page2b*. In the *MainPage* code-behind, on the UI trigger to go to *Page2*, you navigate to *Page2*:

```
private void GotoPage2_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.Relative));
}
```

However, the app does not in fact contain a *Page2.xaml*. Instead, it contains a *Page2a.xaml* and a *Page2b.xaml*. In the *App* class, you hook the *Navigating* event on the *RootFrame* and perform some navigation re-routing whenever you detect that the user is attempting to go to *Page2*.

```
private void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
```

```

if (e.Uri.ToString() == "/Page2.xaml")
{
    Uri newUri = null;
    if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        newUri = new Uri("/Page2a.xaml", UriKind.Relative);
    }
    else
    {
        newUri = new Uri("/Page2b.xaml", UriKind.Relative);
    }
    RootFrame.Dispatcher.BeginInvoke(() =>
        RootFrame.Navigate(newUri));
    e.Cancel = true;
}
}

```

Note that the handler uses *Dispatcher.BeginInvoke*. This is because the *Navigating* event is being handled during navigation, and the system does not allow overlapping navigations. Therefore, you must ensure that the second navigation is queued up. Meanwhile, you terminate the current navigation by setting *NavigatingCancelEventArgs.Cancel* to *true*.

The second technique you can use is URI Mapping (demonstrated in the *TestUriMapping* solution in the sample code). With this approach, instead of handling the *Navigating* event and manually cancelling and re-routing the navigation, you can simply provide a mapping from the original URI to a new URI. This can be either statically declared in XAML or dynamically determined in code. For example, here is a static mapping from *Page2* to *Page2b*:

```

<Application.Resources>
    <nav:UriMapper x:Name="mapper">
        <nav:UriMapping
            Uri="/Page2.xaml"
            MappedUri="/Page2b.xaml"/>
    </nav:UriMapper>
</Application.Resources>

```

This assumes that you have declared a *nav* namespace in the *App.xaml*.

```
xmlns:nav="clr-namespace:System.Windows.Navigation;assembly=Microsoft.Phone"
```

As this namespace indicates, the *UriMapper* and *UriMapping* types are declared in the *System.Windows.Navigation* namespace in the *Microsoft.Phone.dll*. Having declared the mapper and at least one mapping entry, you can use it in the app—typically after the standard initialization code.

```
RootFrame.UriMapper = (UriMapper)Resources["mapper"];
```

That would suffice for a static mapping. If you need a dynamic mapping—as you do in the following example—then you need to modify the *MappedUri* property before using it:

```
Uri newUri = null;
```

```

if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
{
    newUri = new Uri("/Page2a.xaml", UriKind.Relative);
}
else
{
    newUri = new Uri("/Page2b.xaml", UriKind.Relative);
}
UriMapper mapper = (UriMapper)Resources["mapper"];

// dynamic mapping, overwrites any MappedUri set statically in XAML.
mapper.UriMappings[0].MappedUri = newUri;
RootFrame.UriMapper = mapper;

```



Note Additional navigation techniques are discussed in Chapter 12, “Tiles and Notifications,” and in Chapter 20, “Enterprise Apps.”

Of these approaches, the *UriMapper* approach is preferred in general because it can be done mostly in XAML and doesn’t rely on manipulating the navigation system. On the other hand, if you need very dynamic redirection (such as the login timeout scenario), cancelling the navigation might be better for specific/isolated cases.

Summary

This chapter examined the app model, and in particular, the app lifecycle and related events. The tight resource constraints inherent in all mobile devices offer challenges for app developers, particularly with regard to CPU, memory, and disk space. The Windows Phone platform presents a seamless UX with reasonably fast switching between apps to provide an experience that appears to users as if multiple apps are running at the same time. More important, the system exposes just the right number and type of events so that you can hook into the system and use the opportunities presented to make the most of the phone’s limited resources. If you pay attention to these events and take the recommended actions in your event handlers, your app takes part in the overall phone ecosystem, gives users a great experience, and cooperates with the system to maintain system health.

The Windows Phone navigation model is page-based and very intuitive. Although it is possible for you to provide a navigation experience that is different from other apps on the phone, this is discouraged unless you have a very unique and compelling experience. Instead, you are strongly encouraged to take part in the standard navigation model, to respond to the standard navigation events, and to maintain navigation behavior that is consistent with users’ expectations. All but the simplest apps will have some state on both a page basis and an app-wide basis that needs to be persisted across navigation. The app platform on the phone provides targeted support for persisting limited volumes of page and app state.

Data Binding and MVVM

Sooner or later, your app will need to present data in the user interface (UI). Most modern programming frameworks provide mechanisms to make rendering data in the UI simple and robust. At the same time, these frameworks promote better engineering practices by cleanly separating the data from the UI, establishing standard mechanisms for connecting the data and UI in a loosely coupled manner and ensuring that components consuming the data are conveniently notified of any changes (either initiated in the UI or from the underlying data source) so that the app can take appropriate action. The feature that carries out all this cool behavior is called data binding. This chapter examines the data-binding support in the platform, the rationale for its existence, the various ways that it supports both the functionality of mapping data and UI, and the engineering excellence of loose coupling between layers.

Simple Data Binding and *INotifyPropertyChanged*

In a Windows Phone app, you can move data between a backing data source and UI elements manually if you want. Taking this approach, you might declare a field for each UI element (using the `x:Name=""` syntax in XAML) and get/set the displayed value of the element in code. If the data is simple, this is a reasonable approach.

However, with more complex data or where you need to perform additional processing on the data between the source and the UI, this manual back-and-forth propagation will rapidly become a burden. It will involve a lot of manual code, which inevitably increases the chance of introducing bugs. Furthermore, the data is very tightly coupled to the UI. This is a problem because as requirements or the data model change over time, this will necessitate corresponding changes to the UI. Additionally, it will entail changes to all the code that's doing the manual change propagation. Similarly, even if only the UI changes, you will still need to change the propagation code. Such tight coupling makes the whole app very fragile in the face of ongoing requirements changes.

Fortunately, the Windows Phone platform includes support for automatically initializing the UI from backing data and for automatically propagating changes, in both directions. The goals of data binding include:

- Enable a range of diverse data sources to be connected (web service calls, SQL queries, business objects, and so on). The underlying data sources are represented in the app code by a set of one or more data classes.

- Simplify the connection and synchronization of data so that you do not need to maintain propagation code manually.
- Maintain the separation between the UI design and the app logic (and therefore between design tools such as Microsoft Expression Blend and development tools such as Microsoft Visual Studio).

One way to think of data binding is as a pattern with which you can declare the relationship between your app's data and the UI that displays the data *without* hard-coding the specifics of how and when the data is propagated to the UI. This affords you to maximize the separation of concerns. This is an important principle in modern app engineering, in which you maintain clear boundaries between different functional parts of your app. Specifically, this means that the UI components are solely responsible for UI, the data components handle the data source, and so on with minimal overlap between the different responsibilities. Figure 4-1 illustrates the pattern.

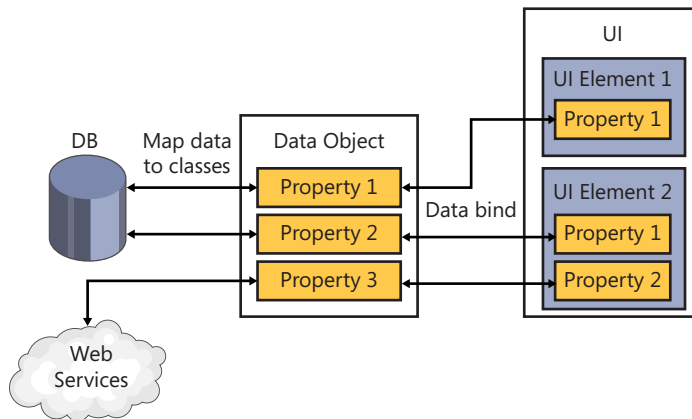


FIGURE 4-1 The data-binding pattern helps maintain the separation of concerns.

In addition to separating concerns, the declarative relationship also takes advantage of change notifications. That is, when the value of the data changes in the underlying data source, the app does not need to take explicit action in order to render the change in the UI. Instead, it relies on the class that represents the data source raising a change notification event, which the app platform picks up and uses to propagate the change to the UI. The same happens in reverse; when the user changes the data interactively in the UI, that change is propagated back to the data source.

The *SimpleDataBinding* solution in the sample code (see also Figure 4-2) demonstrates the basic mechanics of data binding.

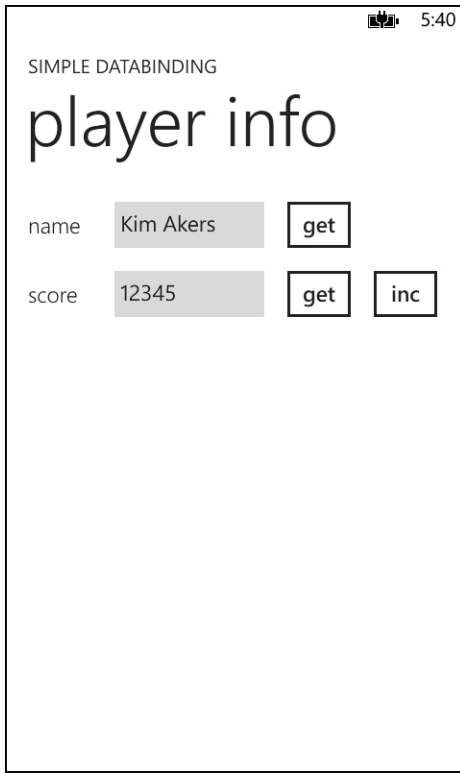


FIGURE 4-2 You can data-bind UI elements to underlying data objects.

The app provides two *TextBox* controls, each data-bound to properties of an underlying data class; in this example, it's a simple *Player* class. One *TextBox* is set to the player's *Name* property; the other is set to the *Score*. The user can edit the contents of either *TextBox*. However, when he taps the corresponding Get button, this retrieves the current value of the underlying data source. From this, it becomes clear that edits to the *Name TextBox* are propagated back to the data source, but edits to the *Score TextBox* are not. Conversely, the user can tap the Inc button, and the app will increment the *Score* value programmatically.

The UI declarations in the XAML specify the binding for each *TextBox* by using the *{Binding}* syntax. In this example, the *Score* has a *one-way binding*, which means that the data is pulled from the data class into the UI only. Any changes made to the underlying data will be propagated to the UI. However, any changes made to the value in the UI will not be propagated back to the data source. On the other hand, the *Name TextBox* has a *two-way binding*. This means that changes are propagated in both directions. Note that *OneWay* mode is the default; thus it's unnecessary to specify it (the

listing that follows only includes it to emphasize that the two *TextBox* controls have different binding modes). Also note that it would make sense in this example to set the *TextBox* control for the *Score* value to be read-only to prevent the user from editing the contents. However, it is left as read-write here to make it more obvious that, even if the user does change the *TextBox* contents, the one-way binding prevents these changes from being propagated back to the underlying data source.



Note The data-binding mechanism does not require UI elements to have defined names. Without data binding, you would need to declare element names in XAML (using the *x:Name=""* syntax) so that the system can generate class fields for these elements. The app would then use these fields to get/set the field values. Avoiding these field declarations has the added benefit of saving a little memory and initialization time.

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

To connect the data object to the UI, you instantiate the data class and assign it to the *DataContext* of the *FrameworkElement* that you want to data-bind. You can specify an individual *FrameworkElement* such as a single control or some containing parent control. At its simplest, this *FrameworkElement* might even be the main page itself, as in this example. Setting the *DataContext* at the page level is a common strategy. All child elements of the page will inherit the same *DataContext*; however, it can also be overridden at any level, if required.

Notice that the *DataContext* property is typed as *object*, which is why you can assign an object of a custom type such as *Player*—you can, of course, assign anything to an object. Assigning the *Player* object to the *DataContext* of the page is how the data-binding system resolves the references to *Score* and *Name* in the binding declarations of the individual elements, because these elements inherit the *DataContext* of the page. The target (UI element) of the binding can be any accessible property or element that is implemented as a *DependencyProperty* (the *DependencyProperty* mechanism is described in Chapter 3, “Core UI, Controls, and Touch”). The source can be any public property of any type.

In the *Click* handler for the *getScore* and *getName* *Button* controls, the app merely displays the value of the underlying data in a message box. On the other hand, whenever the user taps the *Inc* button, the *Score* is incremented on the underlying data, and data binding propagates that value to the UI on your behalf.

```
public partial class MainPage : PhoneApplicationPage
{
    private Player player;

    public MainPage()
    {
        InitializeComponent();
        player = new Player { Name = "Kim Akers", Score = 12345 };
        DataContext = player;
    }
}
```

```

private void getScore_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(player.Score.ToString());
}

private void getName_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(player.Name);
}

private void incScore_Click(object sender, RoutedEventArgs e)
{
    player.Score++;
}
}

```

If you want the system to propagate changes in data values for you, your data class needs to implement *INotifyPropertyChanged*. This defines one member: an event of type *PropertyChangedEventHandler*. For data binding to work, you must expose public properties (not fields) for the data values that you want to be data-bindable. You implement your property setters to raise this event, specifying by name the property that has changed. You can also factor out the invocation of the *PropertyChangedEventHandler* to a custom method, as shown in the code that follows. The more properties you have in the data class, the more useful this becomes.

```

public class Player : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private long score;
    public long Score
    {
        get { return score; }
        set
        {
            if (score != value)
            {
                score = value;
                NotifyPropertyChanged("Score");
            }
        }
    }

    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = value;
                NotifyPropertyChanged("Name");
            }
        }
    }
}

```

```

private void NotifyPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (null != handler)
    {
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```



Note The custom *NotifyPropertyChanged* method doesn't raise the *PropertyChanged* event directly through the class field. Instead, it declares a local *PropertyChangedEventHandler* variable and raises the event through that local variable. This seems redundant, but this is a deliberate technique to make the code more robust in the face of multithreaded calls. Behind the scenes, the app code does not explicitly instantiate the *PropertyChanged* event; rather, this is done for you by the C# compiler. By the same token, removing event handlers is also done for you. When the last event handler is removed, the handler field becomes null, as a housekeeping strategy. Declaring a local variable doesn't protect against the field becoming null before or after you assign it. However, it does protect against it being non-null before you assign it and then null after you assign it, but before you invoke it.

The ability to specify data binding in XAML confers significant benefits, but it is also possible to specify data binding in code. This might be appropriate if the binding is conditional upon some run-time behavior. For example, you could remove the *{Binding}* specifiers in your XAML and replace them with calls to *BindingOperations.SetBinding* in your code. For this to work, you must declare names for the UI elements in XAML because these are required parameters in the *SetBinding* method.

```

public MainPage()
{
    InitializeComponent();
    player = new Player { Name = "Kim Akers", Score = 12345 };
    DataContext = player;

    Binding binding = new Binding("Name");
    binding.Mode = BindingMode.TwoWay;

    // Either of these lines works.
    //BindingOperations.SetBinding(nameText, TextBox.TextProperty, binding);
    nameText.SetBinding(TextBox.TextProperty, binding);

    binding = new Binding("Score");
    binding.Mode = BindingMode.OneWay;
    //BindingOperations.SetBinding(scoreText, TextBox.TextProperty, binding);
    scoreText.SetBinding(TextBox.TextProperty, binding);
}

```

If you want to control UI formatting as part of data binding, you can use the *StringFormat* attribute in the binding definition. Figure 4-3 shows a minor enhancement of the previous app (the *SimpleData Binding_Format* solution in the sample code), in which the *Player* class exposes an additional *DateTime* property to represent the last time the player participated in the game, and two additional *String* properties (*Note* and *Motto*).

```
player = new Player
{ Name = "Kim Akers", Score = 12345, LastPlayed=DateTime.Now, Note="hello world",
Motto=null};
```

The *LastPlayed* property is displayed with custom binding formatting, as is the *Note*. In addition, the *Score* has been formatted (somewhat incongruously) with a thousands-separator and to two decimal places.

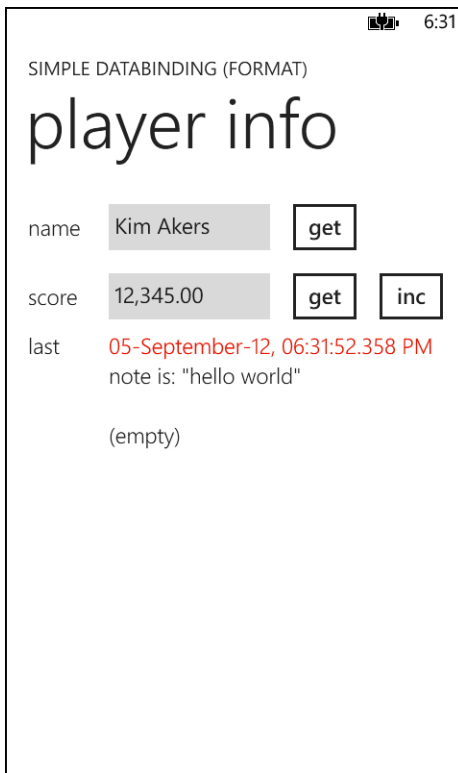


FIGURE 4-3 A demonstration of the *StringFormat* data-binding attribute.

Compare the screenshot in Figure 4-3 with the XAML in which these *StringFormat* values are declared. Observe the use of the backslash to escape the meaning of special characters in the formatting string such as the open "{" and close "}" brackets and the comma (.). Also note that you cannot use the backslash to escape double quotation marks; instead, you must use the XML *"* entity.

```

<TextBox
    Text="{Binding Score, Mode=OneWay, StringFormat=\{0:n2\}}"/>
<TextBlock
    Text="{Binding LastPlayed, StringFormat='dd-MMMM-yy, hh:mm:ss.fff tt'}"
    Foreground="{StaticResource PhoneAccentBrush}"/>

<TextBlock
    Text="{Binding Note, StringFormat='note is: &quot;\{0\}&quot;'}"/>

```

The *Motto* property makes it possible for the value to be null, using the *TargetNullValue* and/or *FallbackValue* attributes. At the bottom of the page are three more *TextBlock* controls, each bound to the same *Motto* property. In this case, the string property is set to null in code. The first variation does not specify what to do in the case of a null value, so nothing is displayed. The second specifies that the string “(empty)” should be used, via the *TargetNullValue* attribute. The third variation uses the *FallbackValue* attribute to specify that the string “unknown” should be displayed if something goes wrong with the data binding. In the screenshot in Figure 4-3, only the second variation results in displayed text, in this instance.

```

<TextBlock Text="{Binding Motto}"/>
<TextBlock Text="{Binding Motto, TargetNullValue=(empty)}"/>
<TextBlock Text="{Binding Motto, FallbackValue=unknown}"/>

```

Data Binding Collections

It is very common to have collections of items that you want to data-bind to UI lists. For example, multiple rows from a data set are commonly bound to some *ItemsControl*, such as a *ListBox*, *ListPicker*, or *LongListSelector*. To bind to a collection, at a minimum, you need to do the following:

- Maintain your individual data objects in an *IEnumerable* collection object of some type.
- Use an *ItemsControl* (or derivative) element as the display container for your list, such as the *ListBox* control.
- Set the *ItemsSource* property of the *ItemsControl* to the collection object.

Where you are displaying one or more properties of a complex data source, you should use a data template. This affords much greater control in formatting the UI. For example, to render a collection of *Player* items, you could have two columns—one for the *Name* and one for the *Score*—and you could use different fonts, colors, styles, backgrounds, and so on for each column, for each row, and so forth. Figure 4-4 presents an example (the *CollectionBinding* solution in the sample code) that demonstrates a *ListBox* bound to a collection of *Player* objects.

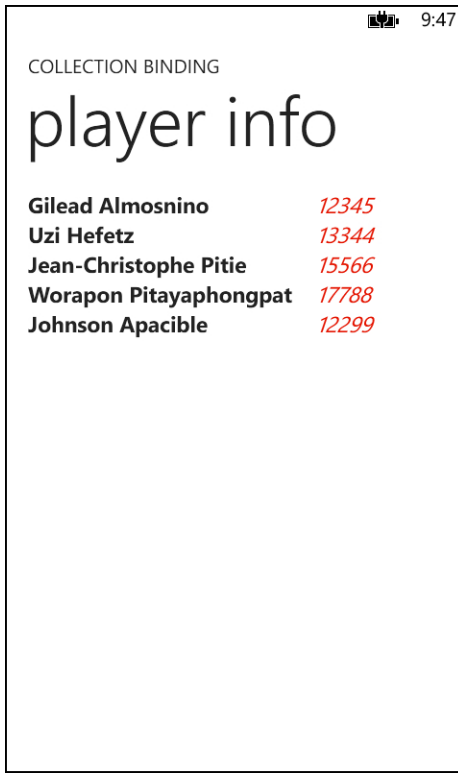


FIGURE 4-4 Data-binding to a collection.

In this example, the data collection itself is initialized in the *MainPage* constructor. To establish the data binding, you assign the collection to the *ItemsSource* property of an *ItemsControl* object; in this example, this is a *ListBox*.

```
private List<Player> players;

public MainPage()
{
    InitializeComponent();

    players = new List<Player>();
    players.Add(new Player { Name = "Gilead Almosnino", Score = 12345 });
    players.Add(new Player { Name = "Uzi Hefetz", Score = 13344 });
    players.Add(new Player { Name = "Jean-Christophe Pitie", Score = 15566 });
    players.Add(new Player { Name = "Worapon Pitayaphongpat", Score = 17788 });
    players.Add(new Player { Name = "Johnson Apacible", Score = 12299 });

    playerList.ItemsSource = players;
}
```

Observe that there is no need to assign the *DataContext* in this case, because you are explicitly assigning the collection data to the *ItemsSource*. You might do both because you might be data-binding one or more collections (with explicitly assigned *ItemsSource* properties) and also individual items (which would rely on the *DataContext* to resolve their bindings). It is also possible to assign a more specific *DataContext* on a per-element basis (at any level in the visual tree). However, it is more common to set the *DataContext* at a page level, allowing each element in the page to inherit this, and then simply assign individual *ItemsSource* collections, as required.



Note Data-binding large collections has a negative effect on performance because of all the housekeeping that the runtime's data-binding framework does in the background. Chapter 10, "Go to Market," discusses mitigation strategies for this.

The data template is defined in XAML and assigned to the *ItemTemplate* property of the *ListBox*. In this example, the template is made up of a *Grid* that contains two *TextBlock* controls, each formatted slightly differently. This is easier to do in Blend than in Visual Studio, but it's still pretty simple in Visual Studio. These use a pair of *{Binding}* attributes, one for the *Name* property of the *Player* item and one for the *Score*. Technically, these are bound to the *Name* and *Score* property of any object that exposes those properties, and not specifically to the *Player* type; although, of course, this app only provides a *Player* type.

```
<ListBox x:Name="playerList" VerticalAlignment="Top" Margin="12,0,12,0" >
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="300"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <TextBlock
          Grid.Column="0" Text="{Binding Name}"
          FontSize="{StaticResource PhoneFontSizeMedium}"
          FontWeight="Bold"/>
        <TextBlock
          Grid.Column="1" Text="{Binding Score}"
          FontSize="{StaticResource PhoneFontSizeMedium}"
          FontStyle="Italic"
          Foreground="{StaticResource PhoneAccentBrush}"/>
      </Grid>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Dynamic Data-Bound Collections

With simple data binding, you want a class that implements *INotifyPropertyChanged* so that changes in value can be notified. If you're binding to a static collection, where the number of elements in the collection doesn't change, you can use a simple collection type such as a *List<T>*. However, if you're binding to a dynamic collection, you should use a collection that implements *INotifyCollectionChanged* so that additions and deletions to the collection can be notified. If you want to be notified of changes both to the collection and to the values of the properties of the items within the collection, you need something like this: *CollectionThatImplementsINotifyCollectionChanged<ItemThatImplementsINotifyPropertyChanged>*.

The following example uses just such a collection. The individual items are a variation on the *Player* class that exposes *Name* and *LastPlayed* properties, and implements *INotifyPropertyChanged*. *Player* items are collected in a custom collection class that derives from *ObservableCollection<T>*, which itself implements *INotifyCollectionChanged*. This custom class exposes an *AddPlayer* method, which adds a new *Player* to the underlying collection, using the supplied name and the current *DateTime*.

```
public class Players : ObservableCollection<Player>
{
    public void AddPlayer(string name)
    {
        Player player = new Player { Name = name, LastPlayed = DateTime.Now };
        this.Add(player);
    }
}
```

The app provides a *TextBox* in which the user can enter the name of a new player, and a "+" *Button* to add the player to the collection. Adding a player triggers the *NotifyCollectionChangedEvent* on the collection. The collection, in turn, is bound to a *ListBox*. The template for the *ListBox* specifies a *Text Block* for the *Name* and *LastPlayed* properties as well as a *Button* with which the user can update the *LastPlayed* value for the currently selected item. This triggers the *NotifyPropertyChangedEvent* on the item. The finished app (the *DynamicCollectionBinding* solution in the sample code) is shown in Figure 4-5.

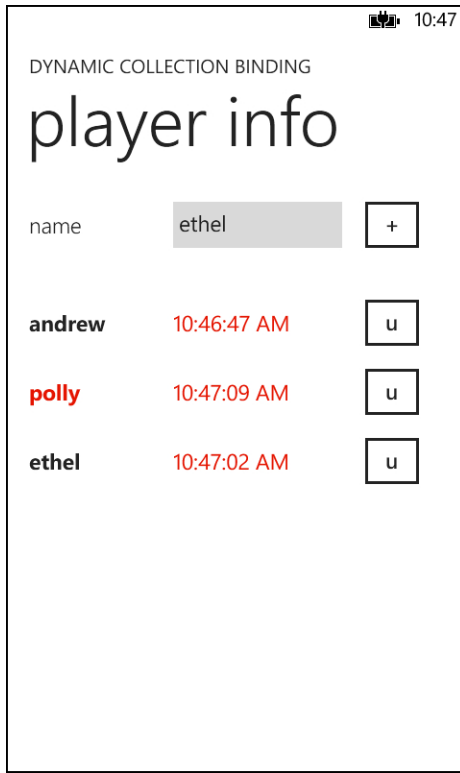


FIGURE 4-5 Data binding works with collections that change dynamically.

The *MainPage* code-behind initializes the collection and binds it to the *ItemsSource* property of the *ListBox*. The *Click* handler for the update *Button* is interesting. This is defined inside the *Data Template*, as demonstrated in the following:

```
<ListBox
    Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="3"
    x:Name="playerList" VerticalAlignment="Top" Margin="12,0,12,0">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="150"/>
                    <ColumnDefinition Width="188"/>
                    <ColumnDefinition Width="80"/>
                </Grid.ColumnDefinitions>
```

```

        <TextBlock
            Grid.Column="0" Text="{Binding Name}"
            FontSize="{StaticResource PhoneFontSizeMedium}"
            FontWeight="Bold" VerticalAlignment="Center"/>
        <TextBlock
            Grid.Column="1" Text="{Binding LastPlayed, StringFormat=hh:mm:ss,
                                   Mode=TwoWay}"
            FontSize="{StaticResource PhoneFontSizeMedium}"
            Foreground="{StaticResource PhoneAccentBrush}"
            VerticalAlignment="Center"/>
        <Button
            Grid.Column="2" x:Name="update" Content="u" Click="update_Click"/>
    </Grid>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>

```

When the user taps one of the *Button* controls in the list, this does not change the selected item—the *ListBox.SelectionChanged* event is not raised. So, to get hold of the correct data object, the *Click* handler retrieves the sending *Button* object and extracts that object's *DataContext*. This will be the bound data object for the whole item; in this case, it's a *Player* object.

```

private Players players;

public MainPage()
{
    InitializeComponent();

    players = new Players();
    playerList.ItemsSource = players;
}

private void addPlayer_Click(object sender, RoutedEventArgs e)
{
    players.AddPlayer(nameText.Text);
}

private void update_Click(object sender, RoutedEventArgs e)
{
    Button button = (Button)sender;
    Player player = (Player)button.DataContext;
    player.LastPlayed = DateTime.Now;
}

```

Template Resources

As you can do with other things such as styles (discussed in Chapter 3), you can define a data template as a resource. This is useful if it's the kind of template that lends itself to reuse or if you're working with a team in which one developer works on the template while another developer works on the page. The mechanism is straightforward. First, you define the template just as you would normally. The only difference is that when implemented as a resource, it must have a defined *Key*. The resource definition resides in the *Resources* section of the XAML element where you want it to be visible. This could be in the *App.xaml* if you want to use the template across multiple pages, or locally in the XAML for the page in which it will be used, either at the page level or at the level of any child element which is at or above the level where it will be used.

```
<phone:PhoneApplicationPage.Resources>
    <DataTemplate x:Key="playerTemplate">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="300"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <TextBlock
                Grid.Column="0" Text="{Binding Name}"
                FontSize="{StaticResource PhoneFontSizeMedium}"
                FontWeight="Bold"/>
            <TextBlock
                Grid.Column="1" Text="{Binding Score}"
                FontSize="{StaticResource PhoneFontSizeMedium}"
                FontStyle="Italic"
                Foreground="{StaticResource PhoneAccentBrush}"/>
        </Grid>
    </DataTemplate>
</phone:PhoneApplicationPage.Resources>
```

Next, in the element to which you want this template to apply, specify the template by its *Key* name. You can see this at work in the *CollectionViewSource* solution in the sample code.

```
<ListBox x:Name="playerList" VerticalAlignment="Top" Margin="12,0,12,0"
    ItemTemplate="{StaticResource playerTemplate}"/>
```

Sorting and Grouping Bound Collections

As part of data-binding a collection, you can sort or group the data by using the *CollectionViewSource* class and the *SortDescriptions* and *GroupDescriptions* collection properties. Figure 4-6 shows the *GroupBinding* solution in the sample code. When the user selects a store item from the list of stores in the first column, the view updates the second column with those products that are associated with the selected store. The key here is that this is all done via data binding—there is no *SelectionChanged* event handler in the code, for instance.

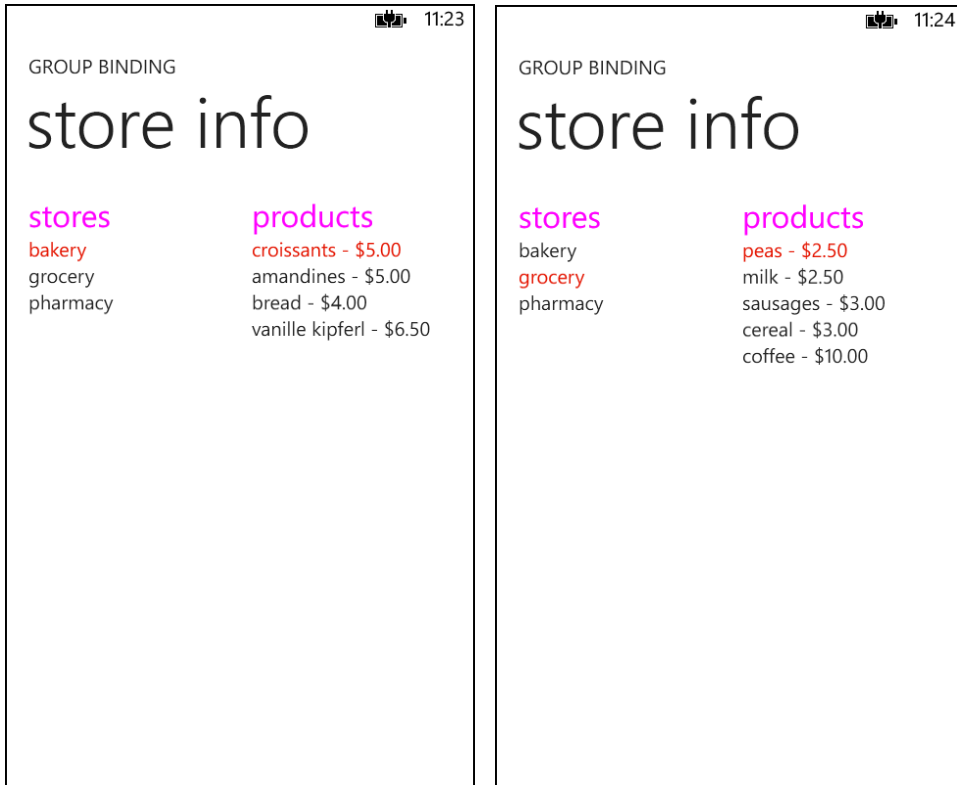


FIGURE 4-6 Data-binding with *CollectionViewSource* objects, and sorting and grouping collections.

A *Store* class represents an individual store, with a string property for the store name and a collection property for the store products. The *Product* class, in turn, consists of a string for the name and a double for the price.

```
public class Store
{
    public String Name { get; set; }
    public ObservableCollection<Product> Products { get; set; }

    public Store(String name)
    {
        Name = name;
        Products = new ObservableCollection<Product>();
    }
}
```

```

public class Product
{
    public String Name { get; set; }
    public double Price { get; set; }

    public Product(String name, double price)
    {
        Name = name;
        Price = price;
    }

    public override string ToString()
    {
        return String.Format("{0} - {1:C2}", Name, Price);
    }
}

```

The app maintains a collection of stores in an instance of the *Stores* class, where the constructor creates some demonstration data (this is an arbitrary collection of stores and products, in no particular order).

```

public class Stores : ObservableCollection<StoreModel>
{
    public Stores ()
    {
        Store grocery = new Store("grocery");
        grocery.Products.Add(new Product("peas", 2.50));
        grocery.Products.Add(new Product("sausages", 3.00));
        grocery.Products.Add(new Product("coffee", 10.00));
        grocery.Products.Add(new Product("cereal", 3.00));
        grocery.Products.Add(new Product("milk", 2.50));
        this.Add(grocery);

        Store pharmacy = new Store("pharmacy");
        pharmacy.Products.Add(new Product("toothpaste", 3.99));
        pharmacy.Products.Add(new Product("aspirin", 5.25));
        this.Add(pharmacy);

        Store bakery = new Store("bakery");
        bakery.Products.Add(new Product("croissants", 5.00));
        bakery.Products.Add(new Product("bread", 4.00));
        bakery.Products.Add(new Product("vanille kipferl", 6.50));
        bakery.Products.Add(new Product("amandines", 5.00));
        this.Add(bakery);
    }
}

```

The XAML defines two additional namespaces: one for the current assembly (where the *Stores* type is defined), and one to resolve the definition of the standard *SortDescription* type.

```

xmlns:local="clr-namespace:GroupBinding"
xmlns:scm="clr-namespace:System.ComponentModel;assembly=System.Windows"

```

The page also defines two *CollectionViewSource* objects as resources. The first is bound to the *Stores* collection; that is to say, all stores. The second *CollectionViewSource* is bound to the first *CollectionViewSource*, specifying the *Products* within that collection as the path. This effectively provides a pivot mechanism on the stores.

```
<phone:PhoneApplicationPage.Resources>

    <local:Stores x:Key="shoppingItems" />

    <CollectionViewSource x:Key="cvs1" Source="{StaticResource shoppingItems}"/>

    <CollectionViewSource x:Key="cvs2" Source="{Binding Source={StaticResource cvs1},
Path=Products}"/>

</phone:PhoneApplicationPage.Resources>
```

For the UI display itself, the XAML defines two *ListBox* controls. For the first one, its *ItemsSource* property is set to the first *CollectionViewSource*; the *TextBlock* in the item template is bound to the store *Name* property. For the second *ListBox*, its *ItemsSource* is set to the second *CollectionViewSource*; the *TextBlock* in the item template is bound implicitly to the whole *Product* item. Recall that the *Product* item overrides *ToString* to render both the product name and price.

```
<ListBox ItemsSource="{Binding Source={StaticResource cvs1}}">

    <ListBox.ItemTemplate>

        <DataTemplate>

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

        </DataTemplate>

    </ListBox.ItemTemplate>

</ListBox>

<ListBox ItemsSource="{Binding Source={StaticResource cvs2}}">

    <ListBox.ItemTemplate>

        <DataTemplate>

            <TextBlock Text="{Binding}"/>

        </DataTemplate>

    </ListBox.ItemTemplate>

</ListBox>
```

This results in the experience shown in the screenshots in Figure 4-6; the first column lists all stores, and the second column lists only those products for the currently selected store. There are also two further enhancements: sorting and grouping. The stores are sorted alphabetically, and the products within each store are grouped according to price. This is achieved very simply in XAML by specifying a *SortDescription* for the first *CollectionViewSource*, and a *GroupDescription* for the second.

```
<CollectionViewSource x:Key="cvs1" Source="{Binding Source={StaticResource shoppingItems}}">
    <CollectionViewSource.SortDescriptions>
        <scm:SortDescription PropertyName="Name"/>
    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>

<CollectionViewSource x:Key="cvs2" Source="{Binding Source={StaticResource cvs1},
Path=Products}">
    <CollectionViewSource.GroupDescriptions>
        <PropertyGroupDescription PropertyName="Price" />
    </CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

Both *SortDescriptions* and *GroupDescriptions* are collection properties. This means that you can specify multiple sorting and grouping definitions for each *CollectionViewSource*, if required.



Note Another useful class for data-binding collections is the *DataServiceCollection<T>* class. This provides simplified binding for data returned by Windows Communications Foundation (WCF) Data Services. The key to this class is that it derives from *ObservableCollection<T>*, which implements *INotifyCollectionChanged* and *INotifyPropertyChanged*, allowing it to update bound data automatically. *DataServiceCollection<T>* is discussed in Chapter 7, “Web and Cloud.”

Type/Value Converters

In addition to formatting, grouping and sorting, it is also possible to convert a data value from one type to another as part of the data-binding process. For example, suppose that you have a collection of *Player* objects that expose two properties, *Name* and *Score*. You want to bind the data values of these properties in the conventional way (each one to a *TextBlock.Text* element). However, you also want to format the *Score* differently depending on its data value; that is, if the value is greater than 10,000, it is rendered as *FontWeights.Black*; otherwise, it's rendered as *FontWeights.Normal*.

Figure 4-7 shows an implementation of this in action (the *BindingConverter* solution in the sample code).

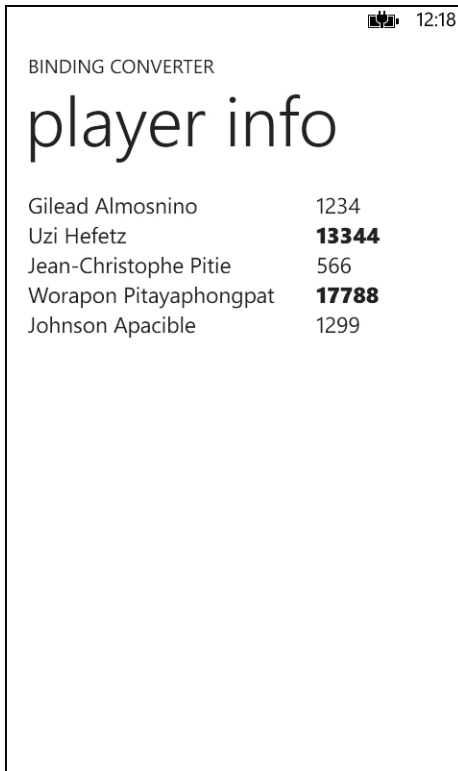


FIGURE 4-7 You can convert data values from one type to another when binding data.

The interesting code is the class that implements *IValueConverter*. This interface declares two methods: *Convert* and *ConvertBack*. If you only want one-way binding, you only need to implement the *Convert* method. For two-way binding, you would also need to implement *ConvertBack*. This example implements the *Convert* method to return a *FontWeight* whose value is computed based on the incoming value parameter. This will be used in the data-binding for the *Score* property. In this way, you convert a *Score* value into a *FontWeight* value.

```
public class ScoreLevelConverter : IValueConverter
{
    public object Convert(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        if ((long)value > 10000)
        {
            return FontWeights.Black;
        }
    }
}
```



```

        else
        {
            return FontWeights.Normal;
        }
    }

    public object ConvertBack(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```

The converter is implemented within the app code, and you want to use it in the XAML for the same app. To make the converter accessible in the XAML, you need to declare a new XML namespace for this assembly. (In this example, this is part of the *PhoneApplicationPage* declaration, alongside all the other namespace declarations. This makes sense in this simple example, but things such as converters tend to be at the app level so that they can easily be shared across many pages.) Then, specify an *ItemsControl* (in this case, a *ListBox*) with a *ScoreLevelConverter* resource. Bind the *TextBlock.Text* to the *Player.Name* in the normal way. The interesting piece is binding the *TextBlock.FontWeight* to the *Player.Score* via the converter, as shown here:

```

<phone:PhoneApplicationPage.Resources>
    <local:ScoreLevelConverter x:Key="scoreConverter"/>
</phone:PhoneApplicationPage.Resources>
...

<ListBox x:Name="playerList" VerticalAlignment="Top" Margin="12,0,12,0" >
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Grid>
                <TextBlock
                    Grid.Column="0" Text="{Binding Name}"
                    FontSize="{StaticResource PhoneFontSizeMedium}"/>
                <TextBlock
                    Grid.Column="1" Text="{Binding Score}"
                    FontSize="{StaticResource PhoneFontSizeMedium}"
                    FontWeight =
                        "{Binding Score, Converter={StaticResource scoreConverter}}"/>
            </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ListBox>

```

As before, the *ListBox.ItemsSource* property is set to the collection of *Player* objects.

Element Binding

In addition to binding to data from a data source, you can also bind from one element to another within the UI. Here is an example that binds the *Text* property of a *TextBlock* to the value of a *Slider*. As the user moves the *Slider*, the value is propagated to the *TextBlock*. Note that this also uses a simple double-to-int value converter, which takes the double values of the *Slider* position and converts them to integers for display in the *TextBlock*.

```
public class DoubleToIntConverter : IValueConverter
{
    public object Convert(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        return System.Convert.ToInt32((double)value);
    }

    public object ConvertBack(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

The critical syntax in the XAML is that which associates the *ElementName* property in the *TextBlock* with the name of the *Slider* element and specifies that the name of the property on the source element to which you want to bind is the *Value* property (set to the *Path* property on the *TextBlock*). The result is shown in Figure 4-8 (the *ElementBinding* solution in the sample code). Also note that forward references are not supported in XAML; thus, the *ElementName* must already be defined in the tree before you reference it.

```
<phone:PhoneApplicationPage.Resources>
    <local:DoubleToIntConverter x:Key="myConverter"/>
</phone:PhoneApplicationPage.Resources>
...
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Slider x:Name="mySlider" Maximum="100"/>
    <TextBlock
        Text="{Binding ElementName=mySlider,
            Path=Value, Converter={StaticResource myConverter}}"
        Style="{StaticResource PhoneTextTitle1Style}"
        Foreground="{StaticResource PhoneAccentBrush}"/>
</StackPanel>
```

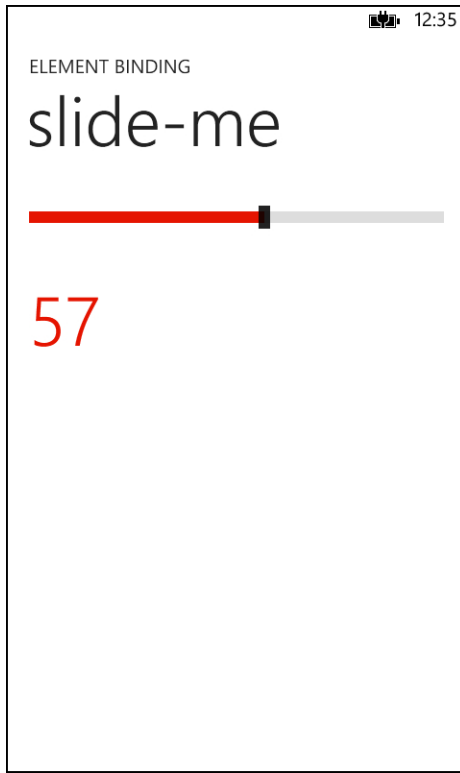


FIGURE 4-8 You can also bind data to UI elements. Here, the *Text* property of a *TextBlock* is bound to the value of a *Slider*.

Data Validation

Windows Phone supports a simple level of data validation in two-way bindings. To make use of this validation, the simplest approach is to have your data class throw an exception in its property setter when it encounters invalid data (see the *BindingValidation* solution in the sample code).

```
private long score;
public long Score
{
    get { return score; }
    set
    {
        if (value >= 0)
        {
            if (score != value)
            {
                score = value;
                NotifyPropertyChanged("Score");
            }
        }
    }
}
```

```

        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

In the XAML for the *MainPage*, you set the *NotifyOnValidationError* and *ValidatesOnExceptions* properties of the *Binding* for the *Score TextBox* to *true*. This directs the binding engine to raise a *BindingValidationError* event when a validation error is added to or removed from the *Validation.Errors* collection. To handle this event, you need to create an event handler either in the *TextBox* or on any of its parents in the hierarchy. It is common to handle validation errors on a per-page basis so that you can handle errors from multiple controls in a consistent manner for the entire page. Reading between the lines, it should be clear that this relies on the fact that the *BindingValidationError* is a routed event (as described in Chapter 3), which will bubble up the hierarchy from the control where the error occurs to the first parent that handles it.

In this example, however, you handle the event half-way up the hierarchy, in the parent *Grid*. The point of doing this is that you can short-circuit the routing and improve performance slightly. This is possible in this case because you know that you have no controls outside the *Grid* that have any validation that could trigger a *BindingValidationError*.

```

<Grid
    x:Name="ContentPanel"
    BindingValidationError="ContentPanel_BindingValidationError">
    <Grid.RowDefinitions>
        <RowDefinition Height="80" />
        <RowDefinition Height="80"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="120"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <TextBlock
        Grid.Row="0" Grid.Column="0" Text="name"
        Style="{StaticResource PhoneTextTitle3Style}"
        VerticalAlignment="Center"/>
    <TextBox
        Grid.Row="0" Grid.Column="1"
        Text="{Binding Name, Mode=TwoWay}"/>
    <TextBlock
        Grid.Row="1" Grid.Column="0" Text="score"
        Style="{StaticResource PhoneTextTitle3Style}"
        VerticalAlignment="Center"/>
    <TextBox
        x:Name="scoreText" Grid.Row="1" Grid.Column="1"
        Text="{Binding Mode=TwoWay, Path=Score,
            NotifyOnValidationError=True, ValidatesOnExceptions=True}"/>
</Grid>

```

The implementation of the event handler is in the *MainPage* class. If an error has been added to the collection, the *TextBox* background will display *Red*. When the error is corrected, and therefore removed from the collection, the standard background for a Phone *TextBox* is restored.

```
private void ContentPanel_BindingValidationError(
    object sender, ValidationErrorEventArgs e)
{
    Debug.WriteLine("ContentPanel_BindingValidationError");

    TextBox t = (TextBox)e.OriginalSource;
    if (e.Action == ValidationErrorEventAction.Added)
    {
        t.Background = new SolidColorBrush(Colors.Red);
    }
    else if (e.Action == ValidationErrorEventAction.Removed)
    {
        t.ClearValue(TextBox.BackgroundProperty);
    }

    e.Handled = true;
}
```

Note also the use of the *DependencyObject.ClearValue* method to reset the *Background Brush*. In this case, this is called on the *BackgroundProperty*. An alternative would be to determine manually which *Brush* you should use (as shown in the code snippet that follows)—for example, if you know you’re using a standard *PhoneTextBoxBrush* resource—but that would clearly be less elegant.

```
t.Background = (Brush)Resources["PhoneTextBoxBrush"];
```

Figure 4-9 shows how the app looks in action. In this scenario, the user has typed in some invalid characters and then moved the focus to another control. This triggers the validation engine in the data binding framework, which then invokes the error handler.

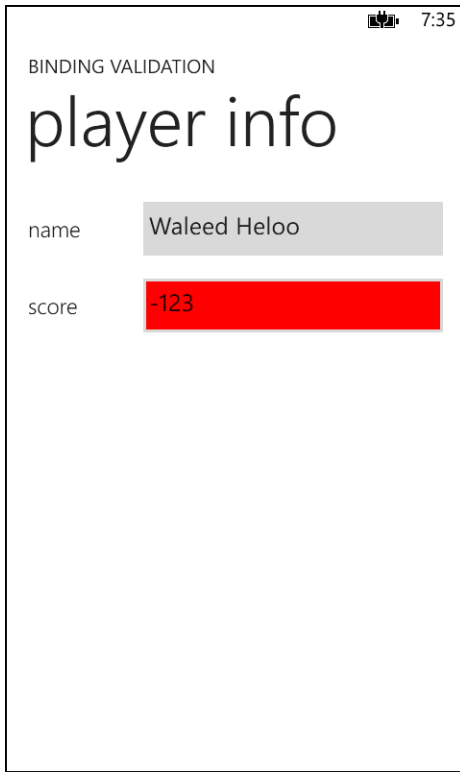


FIGURE 4-9 An invalid character triggers the validation engine in the data-binding framework.

Note that it is also not uncommon to have multiple handlers at different levels in the visual tree. For example, you might have a complex set of visual elements, perhaps several *Grid* controls each containing multiple children, for which you want to handle validation errors for each *Grid* in a different fashion. You might also want to have a catch-all handler at the page level.

```
<phone:PhoneApplicationPage
...
    BindingValidationError="PhoneApplicationPage_BindingValidationError">
        <Grid
            x:Name="ContentPanel"
            BindingValidationError="ContentPanel_BindingValidationError">
...
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

To ensure that the event does not continue routing up the tree, you simply need to set *Handled* to *true* in any handler where you have in fact completely handled the event, as shown in the *ContentPanel_BindingValidationError* method in the preceding example

```
private void PhoneApplicationPage_BindingValidationError(
    object sender, ValidationErrorEventArgs e)
{
    Debug.WriteLine("PhoneApplicationPage_BindingValidationError");
}
```

Given the implementation of *ContentPanel_BindingValidationError*, the *PhoneApplicationPage_BindingValidationError* handler at the page level would never be called, unless some other element outside the *Grid* also triggers a validation error.

If you want even more validation control, you can also consider using *INotifyDataErrorInfo*. You would implement this interface on your data class to signify whether there currently are any validation errors on the object. *INotifyDataErrorInfo* exposes an event that can be raised when there is a validation error. This removes the need for validation to be immediate; instead, you could perform validation asynchronously (perhaps querying a web service) and then raise the event when you eventually determine the result.

By the same token, you can use this to perform validation across multiple properties for which you cannot fully determine whether an individual property is valid until you have examined other properties. This applies especially in circumstances when you need to perform not just cross-property validation, but whole-entity validation. It might be that no single property is invalid, but that the combination of several (or all) of the property values is invalid.

This technique is demonstrated in the *BindingValidation_Info* solution in the sample code, which is a minor variation on the *BindingValidation* sample. The new code is in the *Player* class itself. This now implements *INotifyDataErrorInfo*, which defines the *ErrorsChanged* event, the *GetErrors* method and the *HasErrors* property. To support these, you define a *Dictionary<T>* to hold the collection of errors. When you validate one of the properties (*Score*, in this example), if there is a validation error, you add an entry to the dictionary and then raise the *ErrorsChanged* event.

```
public class Player : INotifyPropertyChanged, INotifyDataErrorInfo
{
    public event PropertyChangedEventHandler PropertyChanged;

    private long score;
    public long Score
    {
        get { return score; }
        set
        {
            if (value >= 0)
            {
                if (score != value)
                {
                    score = value;
                    NotifyPropertyChanged("Score");
                }
            }
        }
    }
}
```

```

        if (errors.ContainsKey("Score"))
        {
            errors.Remove("Score");
        }
    }
}
else
{
    if (!errors.ContainsKey("Score"))
    {
        errors.Add("Score", "value cannot be negative");
    }
}

EventHandler<DataErrorsChangedEventArgs> handler = ErrorsChanged;
if (handler != null)
{
    handler(this, new DataErrorsChangedEventArgs("Score"));
}
}

}

}

...unchanged code omitted for brevity.

private Dictionary<String, String> errors = new Dictionary<String, String>();
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

public System.Collections.IEnumerable GetErrors(string propertyName)
{
    if (String.IsNullOrEmpty(propertyName))
    {
        return errors.Values;
    }
    if (!errors.ContainsKey(propertyName))
    {
        return String.Empty;
    }
    else
    {
        return new String[] { errors[propertyName] };
    }
}

public bool HasErrors
{
    get { return errors.Count > 0; }
}

}

```


If you anticipate having to support more than one error per property, the simple *Dictionary* shown in the preceding code would not be sufficient. In that case, you'd need something like a *Dictionary<String, List<String>>*, instead.

Separating Concerns

So far, the examples in this chapter have focused on data binding, using simple objects and collections of data that are part of the *MainPage* itself. Now, it's time to pay a little more attention to engineering and to further separate the code that represents data from the code that represents UI. At a minimum, the data object(s) should be abstracted from the UI code. Figure 4-10 illustrates this first level of decoupling (the *CollectionBinding_XAML* solution in the sample code).

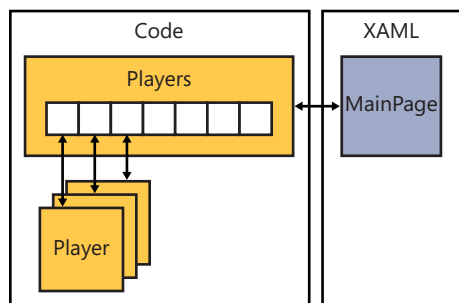


FIGURE 4-10 An example of simple separation of concerns.

The app uses a separate *Players* collection class to represent the collection of *Player* objects. This removes the data collection from the UI class. This class exposes a collection property named *Items*. The constructor initializes its collection by calling a private *GetData* method. This is a simplified stand-in for a method that would fetch the data at runtime, perhaps from some website or from a local database.

```
public class Players
{
    public ObservableCollection<Player> Items { get; private set; }

    public Players()
    {
        Items = new ObservableCollection<Player>();
        GetData();
    }

    private void GetData()
```

```

    {
        Items.Add(new Player { Name = "Gilead Almosnino", Score = 12345 });
        Items.Add(new Player { Name = "Uzi Hefetz", Score = 13344 });
        Items.Add(new Player { Name = "Jean-Christophe Pitie", Score = 15566 });
        Items.Add(new Player { Name = "Worapon Pitayaphongpat", Score = 17788 });
        Items.Add(new Player { Name = "Johnson Apacible", Score = 12299 });
    }
}

```

The *MainPage* class initializes its *DataContext* to a new instance of the *Players* collection. This one line of code is now the only connection in the UI code to the data code, providing much cleaner separation.

```

public MainPage()
{
    InitializeComponent();
    DataContext = new Players();
}

```

One obvious advantage of separating concerns, even in this simple manner, is that the *ItemsSource* value can now be assigned declaratively in XAML instead of in code, as demonstrated here:

```
<ListBox ItemsSource="{Binding Items}">
```

You could take this a step further, and assign the *DataContext* in XAML, also. To do this, you first need to add a namespace in the page's XAML for the current assembly so that you can subsequently refer to the *Players* collection class. Second, declare a keyed resource for the *Players* class. Third, set the *DataContext* to this resource in either the *ListBox* itself or in any of its parents.

```

<phone:PhoneApplicationPage
...
    xmlns:local="clr-namespace:CollectionBinding"
>
    <phone:PhoneApplicationPage.Resources>
        <local:Players x:Key="players"/>
    </phone:PhoneApplicationPage.Resources>

    <Grid
        x:Name="LayoutRoot" Background="Transparent"
        DataContext="{StaticResource players}">
        ...
        <StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <ListBox
                VerticalAlignment="Top" Margin="12,0,12,0"
                ItemsSource="{Binding Items}">
                <ItemsControl.ItemTemplate>
                    ...
                </ItemsControl.ItemTemplate>
            </ListBox>
        </StackPanel>
    </Grid>
</phone:PhoneApplicationPage>

```

Another benefit of separating concerns is that this promotes the separation of work between the design team and the development team. You can now easily set up dummy data for use by the designers. This dummy data is only used at design-time and does not form part of the final app. This gives the designers greater support in laying out the visual interface, based on realistic sample data. Here is how you do this.

First, declare the dummy data in a XAML file. In the example that follows, this is named *DesignTime Data.xaml*, but the name is arbitrary. This needs a namespace to reference the current assembly, which is where the *Players* and *Player* types are defined. In the XAML, define some *Player* items that will be in the *Items* collection in the *Players* object. The following is the entire contents of the file:

```
<local:Players
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:CollectionBinding"
>

    <local:Players.Items>
        <local:Player Name="Dummy Name 1" Score="22334" />
        <local:Player Name="Dummy Name 2" Score="22445" />
        <local:Player Name="Dummy Name 3" Score="22556" />
        <local:Player Name="Dummy Name 4" Score="22667" />
        <local:Player Name="Dummy Name 5" Score="22778" />
    </local:Players.Items>

</local:Players>
```

Note that this is one of the development tasks that is much more easily done in Expression Blend rather than Visual Studio, and that is the primary environment in which design-time data will be used. In the Properties window for this file, set the build action to *DesignData*. Finally, in the *MainPage.xaml*, declare this as design-time data by using the design-time namespace *d* that has already been defined (as <http://schemas.microsoft.com/expression/blend/2008>):

```
d:DataContext="{d:DesignData DesignTimeData.xaml}"
```

You will see this data rendered in the Design view in Visual Studio (and also in Expression Blend), as demonstrated in Figure 4-11. This is the *CollectionBinding_DTD* solution in the sample code. Be aware that this technique won't work if you also set the *DataContext* in XAML, because that will override the design-time setting.

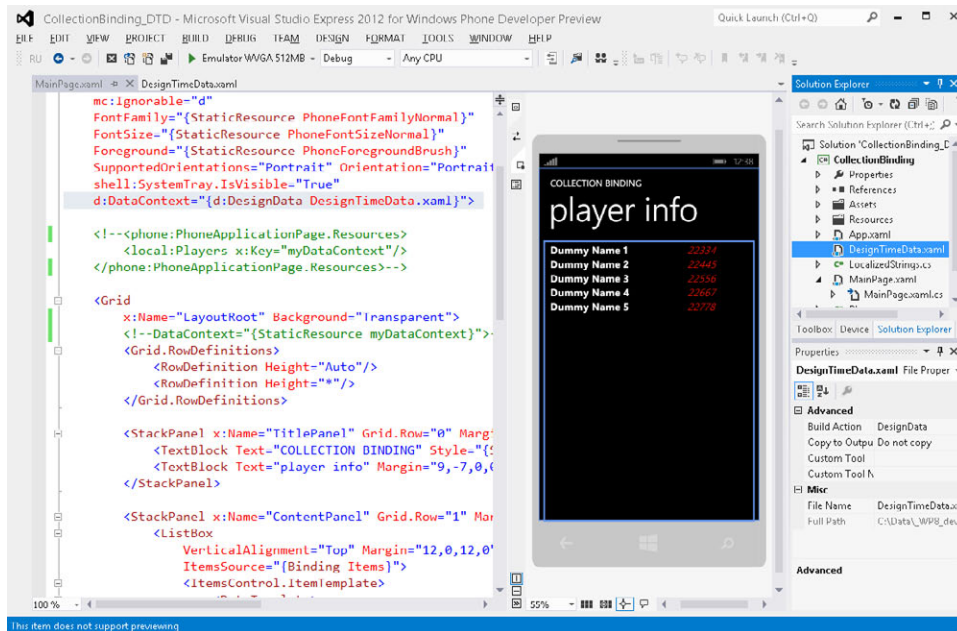


FIGURE 4-11 Viewing design-time data in Visual Studio.

The Model-View-ViewModel Pattern

The Model-View-ViewModel (MVVM) pattern is extensively used in modern apps, including Windows Phone apps. This is an evolution of the Model-View Controller (MVC) pattern. One primary reason is to separate design from code. This supports the scenario in which app UI designers work in Expression, whereas code developers work in Visual Studio—both working on the same app. It also makes testing a lot easier in that you can build automated testing independently for each logical layer (UI, Business Logic, Data Layer, and so on). The three parts of your app are decoupled:

- **View** This is the UI, represented by your XAML, and at a simple level by *mainpage.xaml*.
- **Model** These are the data objects, representing your connection to the underlying data source.
- **ViewModel** This part is the equivalent to the controller in MVC, which mediates between model and view. Typically, the view's *DataContext* is bound to an instance of the viewmodel. The viewmodel, in turn, typically instantiates the model (or the model graph).

Windows Phone also uses Dependency Injection (DI). With DI, when a component is dependent on another component, it doesn't hard-code this dependency; instead, it lists the services it requires. The supplier of services can be injected into the component from an external entity such as a factory or a dependency framework. In Windows Phone, DI is used to provide the glue between the view, the viewmodel, and the model, so that the app does not need to hard-code the connections directly.

For example, you've seen several examples wherein you set the *DataContext* or *ItemsSource* of an element to some concrete object or collection. *DataContext* is of type *object*, and *ItemsSource* is of type *IEnumerable*. These afford extremely loose coupling—you can pretty much assign anything to a *DataContext*, and a very wide range of collection objects to an *IEnumerable*. You inject the specific concrete dependency that you want at some point, either at design-time or during unit testing with some mocked-up data, or at runtime in the final product with real data from the production source.

Figure 4-12 illustrates a high-level representation of the general case. The view, viewmodel and model classes are all decoupled. Given the page-based UI model of Windows Phone apps, this is important to ensure that you can use the same viewmodel in multiple pages. For this reason, no view (page) is responsible for creating the viewmodel. Rather, the *App* creates the viewmodel and exposes it as a property, which is therefore accessible from any page.

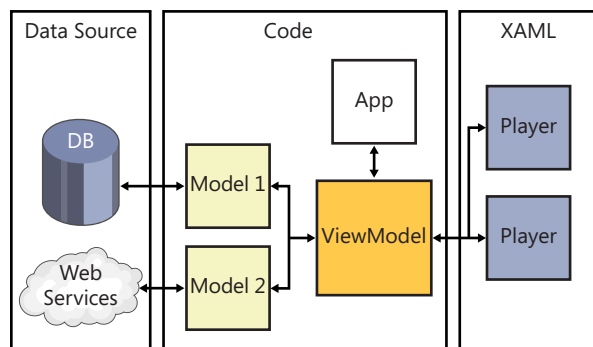


FIGURE 4-12 An overview of the MVVM layers.

The MVVM approach is encouraged in user code, and several of the Visual Studio project templates generate MVVM-based starter code.

The Visual Studio Databound Application Project

The Databound Application template in Visual Studio generates a simple MVVM project (providing the view and viewmodel, but not the model). This is the *DataBoundApp* solution in the sample code (see Figure 4-13). Take a moment to examine the anatomy of this project type. The *MainPage* includes a *LongListSelector* whose items are made up of two *TextBlock* controls. The *DetailsPage* includes two independent *TextBlock* controls.

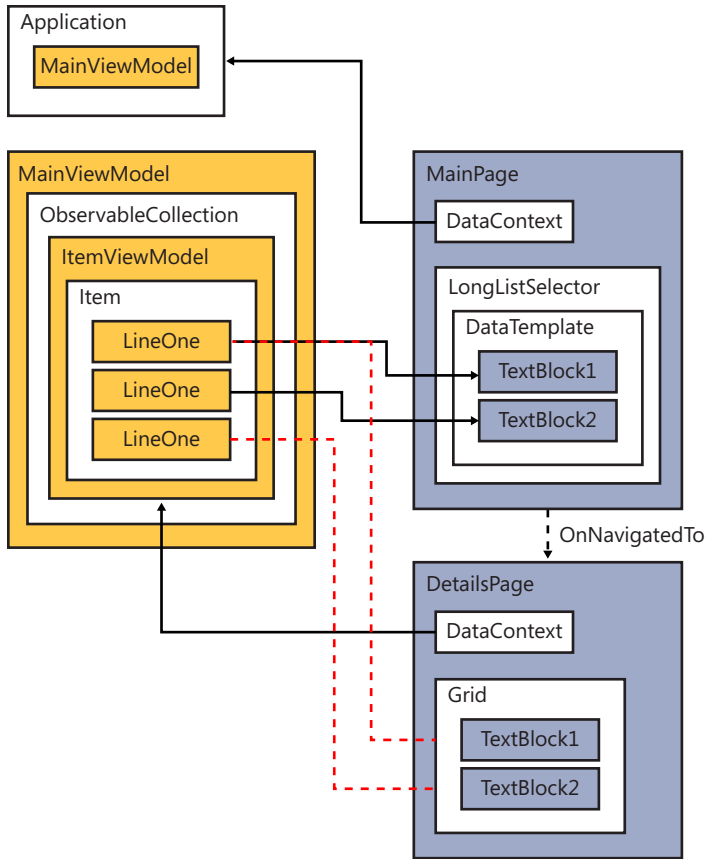


FIGURE 4-13 The Visual Studio Databound Application project template.

The *DataTemplate* for the *LongListSelector* contains two *TextBlock* controls, which are bound to the *LineOne* and *LineTwo* properties in the *ItemViewModel*.

```
<phone:LongListSelector
    x:Name="MainLongListSelector"
    Margin="0,0,-12,0"
    ItemsSource="{Binding Items}"
    SelectionChanged="MainLongListSelector_SelectionChanged">
    <phone:LongListSelector.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="0,0,0,17">
                <TextBlock
                    Text="{Binding LineOne}" TextWrapping="Wrap"
                    Style="{StaticResource PhoneTextExtraLargeStyle}"/>
                <TextBlock
                    Text="{Binding LineTwo}" TextWrapping="Wrap" Margin="12,-6,12,0"
                    Style="{StaticResource PhoneTextSubtleStyle}"/>
            </StackPanel>
        </DataTemplate>
    </phone:LongListSelector.ItemTemplate>
</phone:LongListSelector>
```

The *ItemViewModel* class models the individual items of data. It also, of course, implements *INotifyPropertyChanged*. This class exposes the *LineOne*, *LineTwo* properties to which the *LongListSelector* items are bound.

```
public class ItemViewModel : INotifyPropertyChanged
{
    private string _lineOne;
    public string LineOne
    {
        get { return _lineOne; }
        set
        {
            if (value != _lineOne)
            {
                _lineOne = value;
                NotifyPropertyChanged("LineOne");
            }
        }
    }

    private string _lineTwo;
    public string LineTwo
    {
        ...
    }

    private string _lineThree;
    public string LineThree
    {
        ...
    }
    ...
}
```

The *MainViewModel* class contains an *ObservableCollection* of *ItemViewModel* items, and at run-time it creates an arbitrary set of items in its *LoadData* method (which you would typically replace with real data from your own model). Observe also that the *LoadData* method as supplied in the template unfortunately doesn't check to see if the data has already been loaded.

```
public class MainViewModel : INotifyPropertyChanged
{
    public ObservableCollection<ItemViewModel> Items { get; private set; }
    public MainViewModel()
    {
        this.Items = new ObservableCollection<ItemViewModel>();
    }

    public bool IsDataLoaded
    {
        get;
        private set;
    }
}
```

```

        public void LoadData()
        {
            this.Items.Add(new ItemViewModel() { ID = "0", LineOne = "runtime one", LineTwo =
"Maecenas praesent accumsan bibendum", LineThree = "Facilisi faucibus habitant inceptos interdum
lobortis nascetur pharetra placerat pulvinar sagittis senectus sociosqu" });
            ...//etc
            this.IsDataLoaded = true;
        }
        ...
    }

```

The *App* class has a field that is an instance of the *MainViewModel* class. This is exposed as a property, and the property getter initializes the underlying field, if required.

```

public partial class App : Application
{
    private static MainViewModel viewModel = null;

    public static MainViewModel ViewModel
    {
        get
        {
            if (viewModel == null)
                viewModel = new MainViewModel();

            return viewModel;
        }
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        if (!App.ViewModel.IsDataLoaded)
        {
            App.ViewModel.LoadData();
        }
    }
    ...
}

```

At runtime, the *DataContext* of the *MainPage* is set to refer to the *MainViewModel* in the *App* class. When the page is loaded, it ensures that there is data, loading it if necessary. When the user selects an item from the *LongListSelector*, the app navigates to the *DetailsPage*, passing the selected item in the query string.

```

public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();

        DataContext = App.ViewModel;
    }
}

```



```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }
}

private void MainLongListSelector_SelectionChanged(
    object sender, SelectionChangedEventArgs e)
{
    if (MainLongListSelector.SelectedItem == null)
        return;

    NavigationService.Navigate(new Uri(
        "/DetailsPage.xaml?selectedItem=" +
        (MainLongListSelector.SelectedItem as ItemViewModel).ID,
        UriKind.Relative));

    MainLongListSelector.SelectedItem = null;
}
}

```

Down in the *DetailsPage* class, when the user navigates to the page, the code sets its *DataContext* to the item in the *MainViewModel.Items* collection that is specified in the query string.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (DataContext == null)
    {
        string selectedIndex = "";
        if (NavigationContext.QueryString.TryGetValue(
            "selectedItem", out selectedIndex))
        {
            int index = int.Parse(selectedIndex);
            DataContext = App.ViewModel.Items[index];
        }
    }
}

```

Recall that at design time, in the XAML, the page's *DataContext* is set to a design-time data file (*MainViewModelSampleData.xaml*).

```
d:DataContext="{d:DesignData SampleData/MainViewModelSampleData.xaml}"
```

Figure 4-14 shows the *MainPage* and *DetailsPage* of the standard Databound Application.

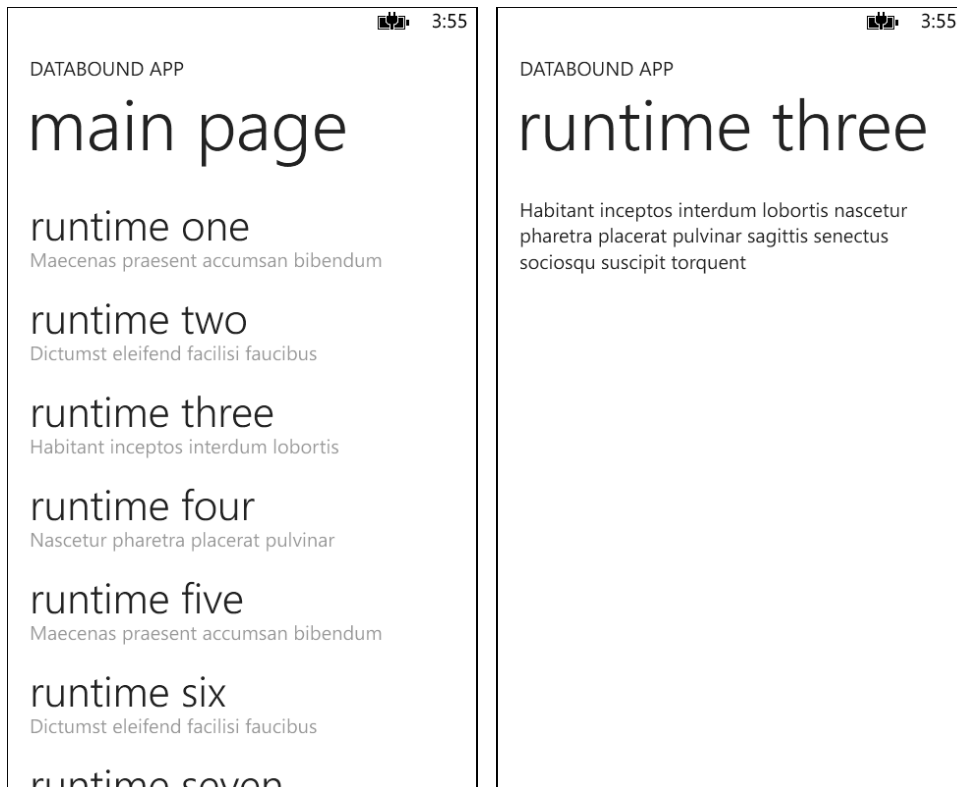


FIGURE 4-14 The Databound Application *MainPage* (on the left) and *DetailsPage* (right).

MVVM in Pivot Apps

The standard Visual Studio template-generated *Pivot* and *Panorama* projects use the same MVVM approach. The *Pivot* project is especially interesting because it illustrates a useful pattern for filtering data; all pivot items are bound to the same data source, but each one has a different “column filter” applied, as represented in Figure 4-15. You can see this at work in the *PivotApp* solution in the sample code (this is an out-of-the-box Visual Studio Pivot Application project).

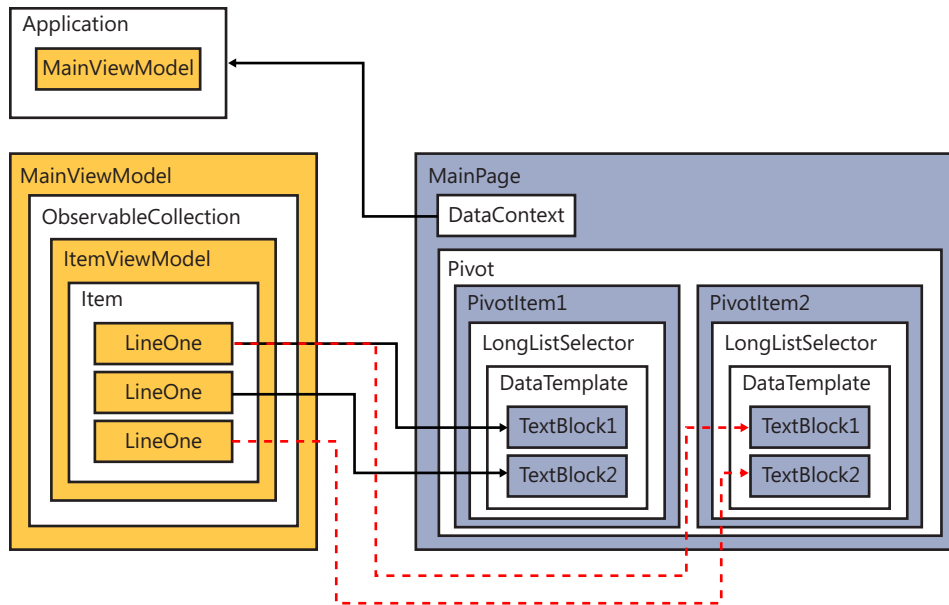


FIGURE 4-15 The Visual Studio Pivot Application.

The *column filtering* is done in the XAML. The first *PivotItem* has a *LongListSelector* whose two *TextBlock* controls are bound to *LineOne* and *LineTwo* in the viewmodel. The second *PivotItem* has a *LongListSelector* whose two *TextBlock* controls are bound to *LineOne* and *LineThree*.

```
<phone:Pivot Title="MY APPLICATION">
  <phone:PivotItem Header="first">
    <phone:LongListSelector ItemsSource="{Binding Items}">
      <phone:LongListSelector.ItemTemplate>
        <DataTemplate>
          <StackPanel>
            <TextBlock Text="{Binding LineOne}" />
            <TextBlock Text="{Binding LineTwo}" />
          </StackPanel>
        </DataTemplate>
      </phone:LongListSelector.ItemTemplate>
    </phone:LongListSelector>
  </phone:PivotItem>

  <phone:PivotItem Header="second">
    <phone:LongListSelector ItemsSource="{Binding Items}">
      <phone:LongListSelector.ItemTemplate>
        <DataTemplate>
          <StackPanel>
            <TextBlock Text="{Binding LineOne}" />
            <TextBlock Text="{Binding LineThree}" />
          </StackPanel>
        </DataTemplate>
      </phone:LongListSelector.ItemTemplate>
    </phone:LongListSelector>
  </phone:PivotItem>
</phone:Pivot>
```

Row Filtering in Pivot Apps

You could obviously take this further and represent the UI of each pivot item differently, according to the nature of the data that is bound to the elements in that item. You could also pivot on the data via *row filtering*. For example, suppose the *ItemViewModel* class also provided an integer *ID* property. Then, you could easily filter the *PivotItem* contents based on the value of this *ID*. Instead of simply allowing the two *LongListSelector* controls to pick up the complete data set from the viewmodel, you could explicitly set each *itemsSource* to some filtered subset of the data, as demonstrated in the *Pivot-Filter* app in the sample code (see Figure 4-16). In this app, the first *PivotItem* lists only odd-numbered items; the second lists only even-numbered items.

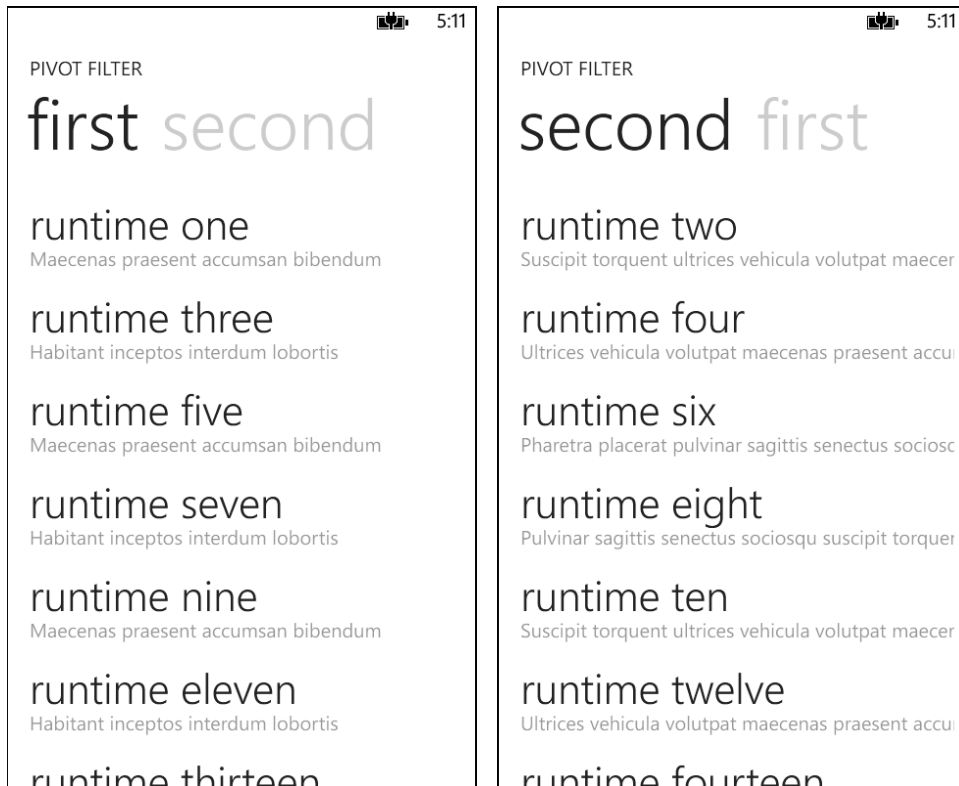


FIGURE 4-16 The Row-filtered *Pivot* app.

In this app, you can use the *MainPage* class to build a collection view on top of the binding source collection that you're using in the view—that is, a layer between the view and the viewmodel. This makes it possible for you to navigate and display the collection, based on sort, filter, and grouping queries, all without the need to manipulate the underlying source collection itself.

```
private CollectionViewSource odds;  
private CollectionViewSource evens;
```

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }

    odds = new CollectionViewSource();
    odds.Source = App.ViewModel.Items;
    odds.Filter += (s, ev) =>
    {
        ItemViewModel ivm = ev.Item as ItemViewModel;
        ev.Accepted = ivm.ID % 2 != 0;
    };

    List<object> list = new List<object>();
    foreach (var v in odds.View)
    {
        list.Add(v);
    }
    firstList.ItemsSource = list;

    evens = new CollectionViewSource();
    evens.Source = App.ViewModel.Items;
    evens.Filter += (s, ev) =>
    {
        ItemViewModel ivm = ev.Item as ItemViewModel;
        ev.Accepted = ivm.ID % 2 == 0;
    };
    list = new List<object>();
    foreach (var v in evens.View)
    {
        list.Add(v);
    }
    secondList.ItemsSource = list;
}
```



Note The mismatch between the *CollectionViewSource.View* (which is an *IEnumerable*) and the *LongListSelector.ItemsSource* (which is an *IList*) requires converting one to the other. There is no standard conversion method, so the approach taken here is to iterate the collection and construct a new *List* from the items. This would not be required if the app used a *ListBox* instead of a *LongListSelector*, because the *ListBox* implementation of *ItemsSource* is an *IEnumerable*. Of course, there are other advantages to the *LongListSelector*, which makes it the control of choice for the Visual Studio templates. The *LongListSelector* is actually an advanced *ListBox* that supports full data and UI virtualization (with the attendant performance benefits), flat lists, and grouped lists.

Improving the Visual Studio Databound Application

If you look closely, you might notice that the Visual Studio Databound Application template is slightly suboptimal. The code to test for loaded data in the viewmodel is duplicated—it's in the *Application_Activated* handler and also in the *MainPage.OnNavigatedTo* override. Not only that, but the code is also identical, using the *App* object reference in both places, even within the *App's Application_Activated* handler where the *App* reference is clearly superfluous.

```
if (!App.ViewModel.IsDataLoaded)
{
    App.ViewModel.LoadData();
}
```

The duplication is to accommodate the fact that the entry point to the app varies, depending on circumstances. This arises from navigation and app lifecycle behavior—issues which are discussed thoroughly in Chapter 2. The lifecycle aspect that is relevant here is that the app might start on the *MainPage*, or it might start on the *DetailsPage*. In normal circumstances, the user launches the app, and the *Application_Launching* event is raised, but not the *Application_Activated* event. Shortly after that, the *MainPage.OnNavigatedTo* method is invoked. It's for this code path that you need the loading code in either the *Application_Launching* handler or the *MainPage.OnNavigatedTo* method.

The second scenario is when the user runs the app, navigates to the *DetailsPage*, and then navigates forward out of the app to another app. When he comes back to the first app, the *Application_Activated* event is raised, and the system navigates to the *DetailsPage*. The key point is that in this scenario, the *DetailsPage* is often created before the *MainPage*. This is why you need the data loading code in either the *Application_Activated* handler or in a *NavigatedTo* handler for the *DetailsPage*.

One improvement would be to remove the *MainPage.OnNavigatedTo* method (which is otherwise unused in the default code) and centralize the code to the *App* class, in the *Application_Launching* and *Application_Activated* handlers. This would at least put it all in one class, and it would afford you the ability to remove the superfluous *App* object reference. An even more elegant solution would be to remove the duplication altogether. You could achieve this by simply putting the loading code in the viewmodel property getter. This would guarantee that anytime the viewmodel is accessed, it will always have loaded data, regardless of the app's launch context, and regardless of how many pages need to access the data. The slight disadvantage is the very small performance cost of doing the *IsDataLoaded* test on each access. A more significant disadvantage is that when you move to an asynchronous loading mechanism, such as from the web or from disk, this is harder to accommodate. You can see these changes in the *DataBoundApp_modified* solution in the sample code.

```
public static MainViewModel ViewModel
{
    get
    {
        if (viewModel == null)
        {
            viewModel = new MainViewModel();
        }
    }
}
```

```

        // Code ported from MainPage_Loaded and Application_Activated.
        if (!viewModel.IsDataLoaded)
        {
            viewModel.LoadData();
        }
        return viewModel;
    }
}

```

Alternatively, if you decide that you always want to load the data on first initialization of the view-model object, you could perform both operations at the same time. Keep in mind that one reason for factoring out the viewmodel to a singleton in the *App* class is to allow access from multiple UI elements. For this reason, it is safer to protect the instantiation with a lock, as shown in the following:

```

public static MainViewModel ViewModel
{
    get
    {
        lock (typeof(App))
        {
            if (viewModel == null)
            {
                viewModel = new MainViewModel();
                viewModel.LoadData();
            }
        }
        return viewModel;
    }
}

```



Note If your data arrives in an asynchronous manner (for example, from the web), you would probably want to raise an event when you've received new data and have the data consumers (viewmodels) subscribe to this event. You'd then have to implement a way to trigger loading, based perhaps on the first access to that event.

Although the MVVM pattern offers a number of benefits and is suitable for most apps that render data in the UI, it is not without its drawbacks. Whereas a reusable framework such as MVVM—or indeed data binding itself—tends to make development more RAD-like in the long run, this comes at the cost of runtime complexity and performance costs. Under the hood, the Windows Phone platform is doing work to handle *NotifyPropertyChanged* and *NotifyCollectionChanged* events and then route them appropriately, including doing reflection to get the required data values (which is always a costly operation). It is also maintaining internal caches related to the data-bound objects.

You should carefully consider the size and performance costs of any technique—the thresholds at which these might become critical are generally much lower on a mobile device than in a desktop app. That said, from an engineering perspective, once you (or your organization) has gone to the effort of setting up the MVVM framework, development effort after that point for interoperating between data and UI is measurably reduced. Furthermore, the benefits increase as the complexity of the app increases.

Summary

This chapter examined the data-binding support in the Windows Phone app platform, the benefits it brings, and the various approaches you can take to customize the behavior by taking part in the data-binding pipeline. Data binding works very well in combination with the MVVM pattern. This pattern helps to ensure clean separation of concerns, such that the discrete functional parts of the app—the data, the view, and the viewmodel—can be loosely coupled, and therefore, independently engineered and independently versioned.

Tiles and Notifications

On all smartphones, apps are represented by an icon in an app list. Windows Phone 7 introduced the notion of an additional app tile that users could pin to the Start screen. This provides several benefits: it is a way for the user to customize his phone experience, prioritizing the apps that he uses the most and positioning them in whatever order he chooses. Because tiles are bigger than app icons, this also introduces the opportunity for you to provide information on the tile that would not fit on an icon. Not only that, but with Windows Phone, the app can keep its tile up to date dynamically. There are multiple dimensions to this feature, including the following:

- An app can have a primary tile plus any number of secondary tiles, which can be created and updated programmatically.
- The style and size guidelines for tiles have evolved from Windows Phone 7 to Windows Phone 8, and now offers multiple tile sizes and new tile style templates.
- Tiles can be updated either purely locally on the phone or from remote servers via the push notification system.

The push notification system can be used to update tiles, and it can also be used to send toasts to the user, and to send any arbitrary raw data to your app—providing a highly efficient mechanism for sending up-to-date data to your app. This chapter discusses all the options that are available to you for tiles, toasts and push notifications.

Tile Sizes and Templates

Windows Phone 7 supported one tile size for Store apps, which included front-to-back flipping. Windows Phone 8 now supports three tile sizes: small, medium, and large. It also supports three tile styles, or templates: flip, iconic, and cycle. All three tile styles are available in all three sizes. You can control the styles programmatically in your app, but only the user can set the size. The user can change the size any time he likes, and the platform does not provide any API to discover the size. For this reason, you should always be prepared to provide images and data for all tile sizes.

Flip tiles have a front and back, and the medium and large flip tiles flip periodically between front and back. This flipping occurs every six seconds or so—the timing is slightly randomized to ensure that all the tiles on the Start screen don't flip all at once. Figure 12-1 presents an example of flip tiles.

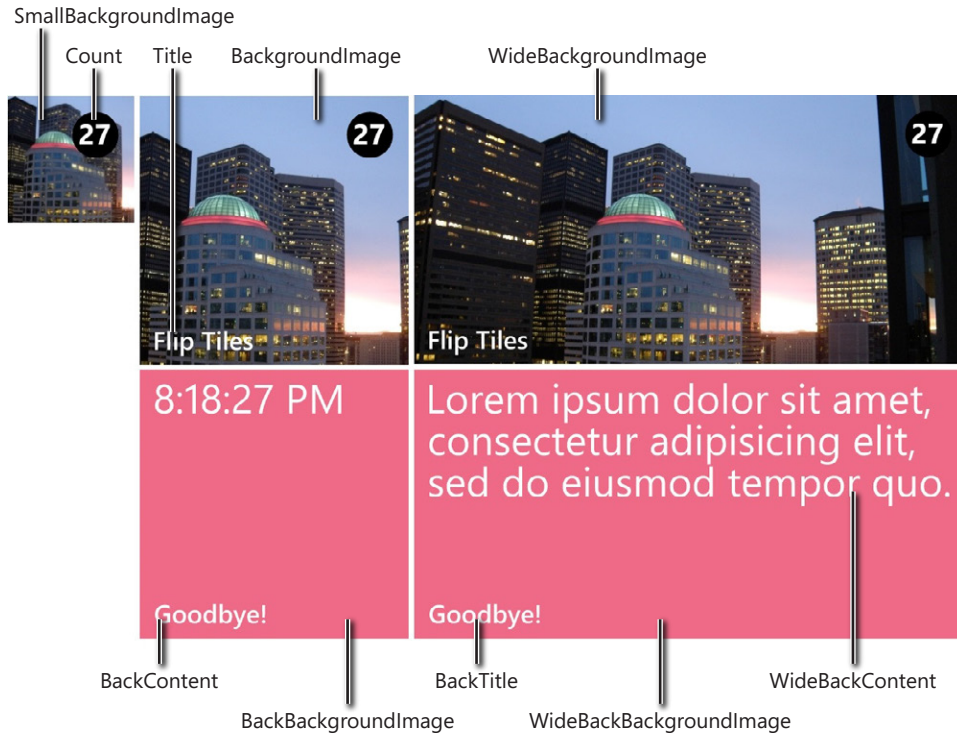


FIGURE 12-1 Flip tiles have a front (top) and back (bottom) side, and flip automatically between them.

To update a flip tile, you initialize a *FlipTileData* object and pass it to the *ShellTile.Update* method. For the primary tile, you retrieve the first tile in the *ActiveTiles* collection. You can specify values for the front and back titles, the count, three front image files, medium and large back images, and medium and large back content. For *WideBackContent*, you can specify multiple lines of text, up to about 80 characters; anything beyond that will be truncated. The number is approximate because different characters have different widths, so it depends which characters you're using. The small tile does not flip, and the count is shown only on the front.

```
ShellTile primaryTile = ShellTile.ActiveTiles.FirstOrDefault();
FlipTileData flipTileData = new FlipTileData()
{
    Title = "Flip Tiles",
    Count = now.Second,
    BackgroundImage = new Uri("/Assets/Tiles/FlipTileMedium.png", UriKind.Relative),
    SmallBackgroundImage = new Uri("/Assets/Tiles/FlipTileSmall.png", UriKind.Relative),
    WideBackgroundImage = new Uri("/Assets/Tiles/FlipTileLarge.png", UriKind.Relative),
    BackTitle = "Goodbye!",
    BackContent = now.ToLongTimeString(),
    BackBackgroundImage = new Uri("/Assets/Tiles/FlipTileMediumBack.png", UriKind.Relative),
    WideBackContent = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor quo.",
    WideBackBackgroundImage = new Uri("/Assets/Tiles/FlipTileLargeBack.png", UriKind.Relative),
};
primaryTile.Update(flipTileData);
```

Cycle and iconic tiles have only a front side and don't flip. The medium and large cycle tiles sequence between multiple images every few seconds (up to nine), in a random order, and they also scroll slowly vertically. When a cycle tile switches image, it does so by scrolling the new image up from the bottom, as shown in Figure 12-2, in which the top image is shown transitioning to the bottom image.

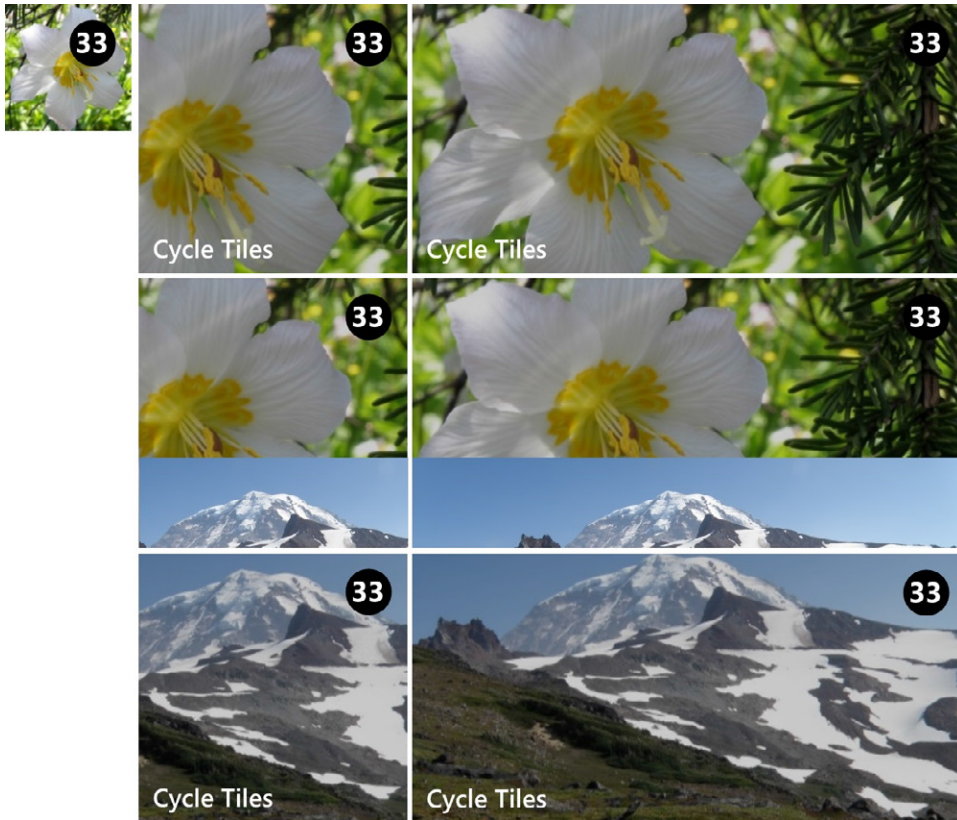


FIGURE 12-2 Cycle tiles cycle between nine images.

To update a cycle tile, you initialize a *CycleTileData* object and pass it to *ShellTile.Update*. For cycle tiles, you specify the title, count, and the image for the small tile, plus up to nine images for the cycling medium and large tiles. The small tile does not cycle, the title is shown only on medium and large tiles, and the count is shown on all three formats.

```
ShellTile primaryTile = ShellTile.ActiveTiles.FirstOrDefault();
CycleTileData cycleTileData = new CycleTileData()
{
    Title = "Cycle Tiles",
    Count = now.Second,
    SmallBackgroundImage = new Uri("/Assets/Tiles/SprayPark3Small.png", UriKind.Relative),
    CycleImages = new List<Uri>
```

```

{
    new Uri("/Assets/Tiles/SprayPark1.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark2.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark3.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark4.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark5.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark6.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark7.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark8.png", UriKind.Relative),
    new Uri("/Assets/Tiles/SprayPark9.png", UriKind.Relative),
}
};
primaryTile.Update(cycleTileData);

```

Iconic tiles use the clean, modern app style (see Figure 12-3), with a simple background color, no background images, and simple icon images that are limited to white and transparent colors. For iconic tiles, you prepare two images: small and medium. If the user selects a wide tile, the iconic style uses the small image if there is wide content; otherwise, it uses the medium image.

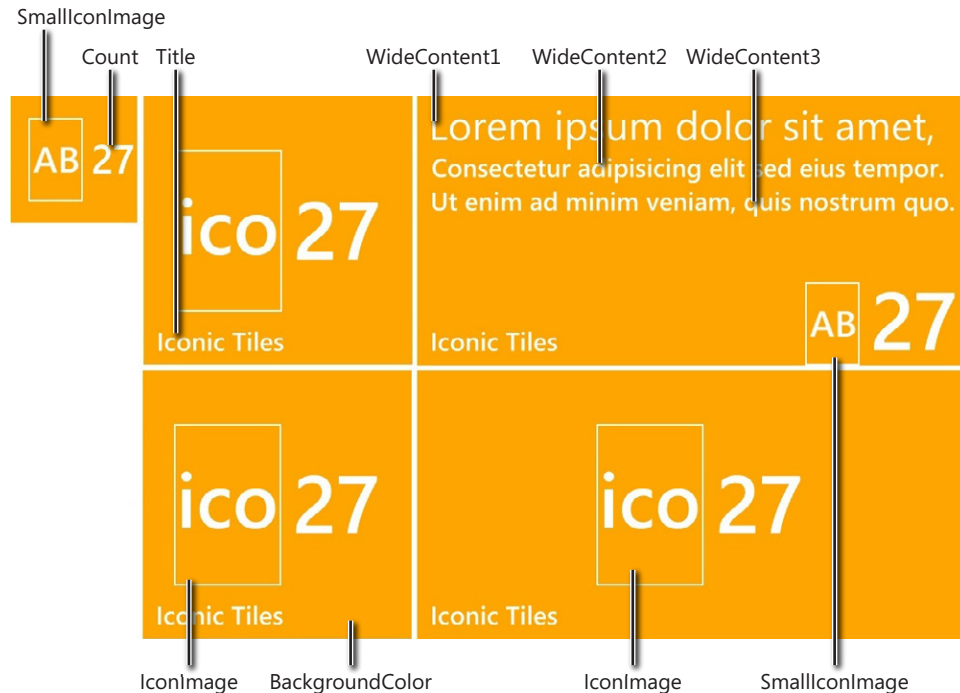


FIGURE 12-3 Iconic tiles use the clean, modern app style.

To update iconic tiles, you initialize an *IconicTileData* object and pass it to *ShellTile.Update*. If you provide *WideContent1*, you're limited to one line of about 30 characters; anything beyond that is truncated. For *WideContent2* and *WideContent3*, each is limited to one line of about 42 characters.

```

ShellTile primaryTile = ShellTile.ActiveTiles.FirstOrDefault();
IconicTileData iconicTileData = null;
if ((bool)wideContent.IsChecked)

```

```

{
    iconicTileData = new IconicTileData()
    {
        Title = "Iconic Tiles",
        Count = now.Second,
        BackgroundColor = Colors.Orange,
        IconImage = new Uri("/Assets/Tiles/IconicTileMedium.png", UriKind.Relative),
        SmallIconImage = new Uri("/Assets/Tiles/IconicTileSmall.png", UriKind.Relative),
        WideContent1 = "Lorem ipsum dolor sit amet,",
        WideContent2 = "Consectetur adipisicing elit sed eius tempor.",
        WideContent3 = "Ut enim ad minim veniam, quis nostrum quo.",
    };
}
else
{
    iconicTileData = new IconicTileData()
    {
        Title = "Iconic Tiles",
        Count = now.Second,
        BackgroundColor = Colors.Orange,
        IconImage = new Uri("/Assets/Tiles/IconicTileMedium.png", UriKind.Relative),
        SmallIconImage = new Uri("/Assets/Tiles/IconicTileSmall.png", UriKind.Relative),
        WideContent1 = "", WideContent2 = "", WideContent3 = ""
    };
}
}
primaryTile.Update(iconicTileData);

```

For the primary tile, the template must be specified at compile time. You do this by using the WMAppManifest graphical editor, as demonstrated in Figure 12-4. Using this, you can select each image and the title text. This is also where you specify the screen resolutions that you want to support.

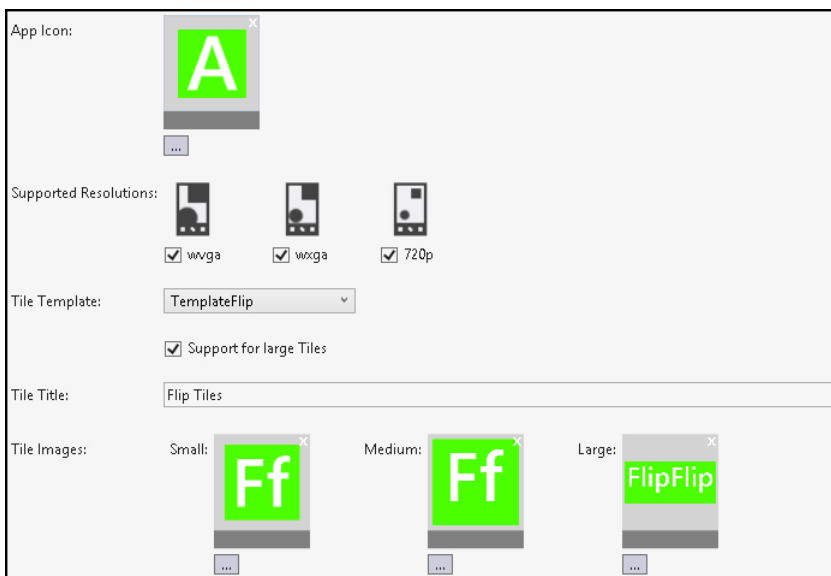


FIGURE 12-4 You can use the graphical manifest editor to set the primary tile template.



Note Windows Phone 8 supports three screen resolutions: WVGA (800x480 pixels), 720p (1280x720 pixels), and WXGA (1280x768 pixels). There's a performance trade-off with screen resolutions. If you want to view the highest quality graphics at the highest resolution screen (WXGA), your images will take up more memory. You have two choices here. One option is to use high-resolution images and allow the system to scale them down when running on a device with a lower-resolution screen. The other option is to use lower-resolution images and allow the system to scale them up when running on a device with a higher-resolution screen. The automatic scaling that the platform performs produces very good results, and in most cases will be indistinguishable to the user. On the other hand, the memory cost of using high-resolution images can be significant, and you should profile your app to see if it is worthwhile, especially if you are constructing images dynamically. This becomes even more critical if you're manipulating images from a background agent, as discussed in Chapter 8, "Background Agents."

When you use the manifest editor, it updates the XML for you. Alternatively, you can edit the *WMAppManifest.xml* file directly. The flip template settings from Figure 12-4 are represented in XML in the following:

```
<TemplateFlip>
  <SmallImageURI IsRelative="true" IsResource="false">Assets\Tiles\FlipTileSmall.png</
SmallImageURI>
  <Count>0</Count>
  <BackgroundImageURI IsRelative="true" IsResource="false">Assets\Tiles\FlipTileMedium.png</
BackgroundImageURI>
  <Title>Flip Tiles</Title>
  <BackContent>
</BackContent>
  <BackBackgroundImageURI IsRelative="true" IsResource="false">
</BackBackgroundImageURI>
  <BackTitle>
</BackTitle>
  <LargeBackgroundImageURI IsRelative="true" IsResource="false">Assets\Tiles\FlipTileLarge.png</
LargeBackgroundImageURI>
  <LargeBackContent>
</LargeBackContent>
  <LargeBackBackgroundImageURI IsRelative="true" IsResource="false">
</LargeBackBackgroundImageURI>
  <HasLarge>True</HasLarge>
</TemplateFlip>
```

The *TileSizes* app in the sample code demonstrates all three template styles. The app discovers the template style in use by reading the manifest at runtime and then offers user interface (UI) options to update the primary tile. The code for reading the information from the manifest is quite simple, and it relies on XML navigation using *XDocument*.

```
private TemplateType GetTemplateTypeFromManifest()
{
    TemplateType templateType = TemplateType.TemplateUnknown;
```

```

XDocument appManifest = XDocument.Load("WAppManifest.xml");
if (appManifest != null)
{
    var primaryTokenNode = appManifest.Descendants("PrimaryToken");
    if (primaryTokenNode != null)
    {
        var templateNode = primaryTokenNode.Descendants().FirstOrDefault();
        if (templateNode.Name == "TemplateFlip")
        {
            templateType = TemplateType.TemplateFlip;
        }
        else if (templateNode.Name == "TemplateIconic")
        {
            templateType = TemplateType.TemplateIconic;
        }
        else if (templateNode.Name == "TemplateCycle")
        {
            templateType = TemplateType.TemplateCycle;
        }
    }
}
return templateType;
}

```

When you create a new project, Microsoft Visual Studio generates six tile images, plus an image for the app icon. These are all placeholders with a default graphic, but the image files are sized appropriately, so they offer a good starting point for your own images. The usable area within the image varies according to the size of tile and the template. As a general guideline, for iconic tiles, you should configure a border of about 19 pixels all around, leaving an inner, “usable” area. This is simply a guideline to ensure that your text/data fits in with the way the system positions the icon and count data. This does not apply to cycle and flip tiles, for which you can use the entire surface of the image. Table 12-1 lists the recommended tile sizes.

TABLE 12-1 Recommended Image Sizes and Usable Areas by Template and Tile Size

Template	Tile Size	Image Size	Usable Area
Flip	Small	159x159	121x121
	Medium	336x336	298x298
	Large	691x336	653x298
Cycle	Small	159x159	159x159
	Medium	336x336	336x336
	Large	691x336	691x336
Iconic	Small	71x110	71x110
	Medium	134x202	134x202

You can use JPG or PNG image files. The trade-off here is that PNG supports transparency, which is especially recommended for iconic tiles. On the other hand, PNG image files are generally larger than

the equivalent JPG image file for photographic content with gradients and the like. For blocks of solid color, PNG are smaller. You should try both to see what looks better, and what the size trade-off is in your specific case.

Secondary Tiles

In addition to the primary tile—which all apps have, even if they’re not pinned to the Start screen—you can also create one or more secondary tiles. All the same properties apply to both primary and secondary tiles, but a key distinction is that when you create a secondary tile, you can choose at that point which template to apply. Another distinction is that a secondary tile can be linked to any page in your app, and this results in some interesting choices in the page navigation model. The *Secondary Tiles* app in the sample code demonstrates how to create, update, and delete secondary tiles. The app has two pages: *MainPage* and *Page2*. *MainPage* offers just a *HyperlinkButton* set to navigate to *Page2*. *Page2* offers three buttons to create, update, and delete a secondary tile. The app uses the flip template, but the same technique can be used for cycle or iconic tiles, too. You could even create a mixture of secondary tiles that use all three of the tile templates, but this is probably not a good UI design. In general, it makes the most sense to create secondary tiles that use the same template as the app’s primary tile. Again, this will depend on the specifics of your app. For example, you can imagine an app such as Facebook might want an iconic tile for its primary tile, a cycle tile for any pinned albums, a flip tile for any pinned friends, and so on.

Of course, a realistic app would likely have a more sophisticated UI for triggering tile operations based on user action rather than just simple buttons. The typical scenario would be that the user opts to pin a tile to the Start screen that corresponds to some data item in your app. Then, the app would update the tile—typically without user interaction—to keep the tile fresh. Finally, deleting the tile would normally not be done in the app code; instead, it would be triggered by the user explicitly unpinning it from the Start screen.

All the interesting work is in the *Page2* button *Click* handlers. The *Click* handler for the create button initializes a *FlipTileData* object, and passes it to *ShellTile.Create*. This takes two additional parameters: the navigation *Uri* for the tile, and a *bool* that indicates whether this tile supports the wide format.

```
private void createTile_Click(object sender, RoutedEventArgs e)
{
    DateTime now = DateTime.Now;
    FlipTileData tileData = new FlipTileData
    {
        Title = "Sound Sunrise",
        Count = now.Second,
        BackgroundImage = new Uri("/Assets/Tiles/sunrise_336x336.png", UriKind.Relative),
        SmallBackgroundImage = new Uri("/Assets/Tiles/sunrise_159x159.png", UriKind.Relative),
        WideBackgroundImage = new Uri("/Assets/Tiles/sunrise_691x336.png", UriKind.Relative),
        BackTitle = "Sound Sunset",
        BackContent = now.ToLongTimeString(),
        BackBackgroundImage = new Uri("/Assets/Tiles/sunset_336x336.png", UriKind.Relative),
```

```

        WideBackContent = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, " +
            "sed do eiusmod tempor quo.",
        WideBackBackgroundImage = new Uri("/Assets/Tiles/sunset_691x336.png", UriKind.Relative),
    };
    ShellTile.Create(new Uri("/Page2.xaml?ID=SoundTile", UriKind.Relative), tileData, true);
    UpdateButtons();
}

```

A common pattern is to generate images dynamically according to some runtime context and then persist the images by using the *WriteableBitmap* class. The *System.Windows.Media.Imaging* namespace includes extension methods for this class to save and load JPG format images. The variation in the example that follows does just that. The custom *SaveTileImage* takes in a tile size; composes an image dynamically, with varying content according to the size of the tile; saves that image to isolated storage; and returns a corresponding *Uri*.

```

private Uri SaveTileImage(TileSize tileSize, int width, int height)
{
    string fileName = String.Format("/Shared/ShellContent/Dynamic{0}.jpg", tileSize);
    WriteableBitmap wb = new WriteableBitmap(width, height);
    TextBlock tb = new TextBlock()
    {
        Foreground = new SolidColorBrush(Colors.White),
        FontSize = (double)Resources["PhoneFontSizeExtraExtraLarge"]
    };

    switch (tileSize)
    {
        case TileSize.Small:
            tb.Text = "abc";
            break;
        case TileSize.Medium:
            tb.Text = "def ghi";
            break;
        case TileSize.Large:
            tb.Text = "jkl mno pqr stu vwx";
            break;
    }

    wb.Render(tb, new TranslateTransform() { X = 20, Y = height / 2 });
    wb.Invalidate();

    using (IsolatedStorageFile isoStore = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isoStore.FileExists(fileName))
        {
            isoStore.DeleteFile(fileName);
        }
        using (IsolatedStorageFileStream fileStream =
            new IsolatedStorageFileStream(fileName, FileMode.Create, isoStore))
        {
            wb.SaveJpeg(fileStream, wb.PixelWidth, wb.PixelHeight, 0, 100);
        }
    }
    return new Uri("isostore:" + fileName, UriKind.Absolute);
}

```



Note If you want to generate images on the fly, and you want to include transparency, you need to save your images in PNG format. However, this is not supported in the standard library. For a solution to this, you can consider using the *WriteableBitmapEx* third-party library, which is available on codeplex at <http://writeablebitmapex.codeplex.com/>.

In this sample app, *Page2* also has a *TextBlock* whose *Text* is set to a string that indicates whether the user navigated to this page via a pinned tile on the Start screen or via the *HyperlinkButton* on the *MainPage*. When you create a secondary tile, you can also specify the *NavigationUri* for the tile. This must include the page to which to navigate within this app, plus optionally a query string with whatever parameters you want. This sample app sets one *ID* parameter, which it uses subsequently to determine which tile this is. When you call the *ShellTile.Create* method, the tile is created with the specified properties and pinned to the Start screen on the phone. The Start screen is an app that is part of the system shell, so this action causes a navigation away from your app, which is therefore deactivated. The reason for this is to avoid spamming the Start screen: when you create a tile, the system makes it very obvious to the user that the tile has been created, and the user is shown where the tile is. She can then immediately interact with it, perhaps by moving it around, resizing it, or deleting it if she doesn't want it.

There are two ways to get to *Page2* in the app: through normal navigation via the hyperlink on the main page of the app, or via a pinned tile on the Start screen. So that you can determine which route was taken, you need to override the *OnNavigatedTo* method. This is where the *ID* parameter comes into play; the app can examine the query string to see which tile the user tapped to get to this page. This is also where you cache the *ShellTile* object. There's only one secondary tile in this app, so you can cache this as a *ShellTile* field in the class. If there were more tiles, it would make sense to use a collection, instead.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    String tmp;
    if (NavigationContext.QueryString.TryGetValue("ID", out tmp))
    {
        navigation.Text = String.Format("from Start ({0})", tmp);
    }
    else
    {
        navigation.Text = "from MainPage link";
    }

    tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains("ID=" + tmp));
}
```

In this example, if the query string indicates that the user arrived at *Page2* via a pinned tile, you simply extract the parameter value and display it in a *TextBlock*. In a more sophisticated app, you would use this identifier to govern some business logic in your solution.

Updating a tile's properties and deleting a tile are both very straightforward. To update a tile, you simply find that tile and invoke the *Update* method, passing in a replacement set of data by using an object of the appropriate *ShellTileData*-derived class (*FlipTileData*, *CycleTileData*, or *IconicTileData*), as before. You don't have to provide values for all the properties, because any properties for which you don't provide values simply retain their previous value. If you want to clear a value, you can provide an empty string or a *Uri* with an empty string, depending on the item to be cleared. To delete a tile, find the tile and invoke the *Delete* method, but remember that in a real app, you should normally leave tile deletion to the user and avoid doing this programmatically. Note that neither updating nor deleting need to send the user to the Start screen, so there is no navigation away from the app in these cases.

```
private void updateTile_Click(object sender, RoutedEventArgs e)
{
    DateTime now = DateTime.Now;
    FlipTileData tileData = new FlipTileData
    {
        Count = now.Second,
        BackContent = now.ToLongTimeString(),
    };
    tile.Update(tileData);
}

private void deleteTile_Click(object sender, RoutedEventArgs e)
{
    tile.Delete();
    tile = null;
}
```



Note One technique that you should *not* adopt is to force the user to pin a secondary tile to get live updates as if it were the primary tile. The user will pin the primary tile if that's what he wants; secondary tiles should be for additional entry points.

Pinning Tiles

The ability to pin local tiles to the Start screen becomes more interesting when the user can choose from multiple possible tiles to pin within an app. Consider a typical weather app. The user can mark individual locations as favorites and can then pin zero or more of these favorites to the Start screen. The screenshots in Figure 12-5 show the *PinTiles* solution from the sample code. This is based on the standard Visual Studio Databound Application project type.

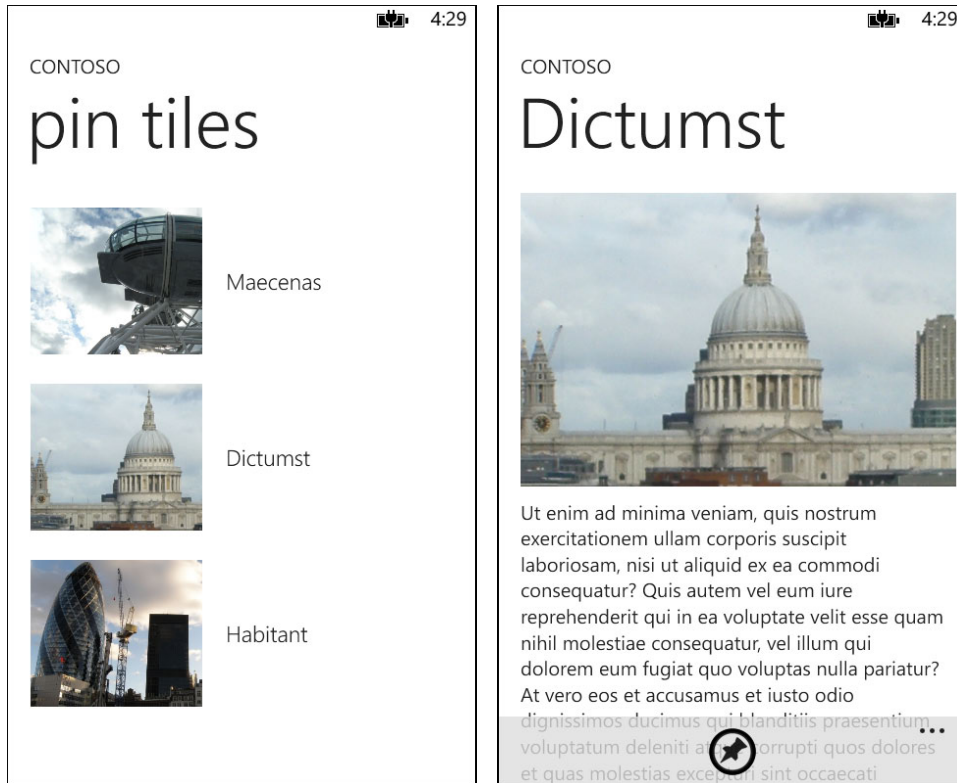


FIGURE 12-5 The *MainPage* (on the left) and *DetailsPage* (right) of the *PinTiles* sample app.

The idea here is that the app presents a list of items on the *MainPage*, and when the user taps one of these items, it navigates to the *DetailsPage*, which is data-bound to that item's viewmodel. In addition, the UI presents an app bar *Button* control that displays a "pin" image. When the user taps this control, the app creates a local tile and pins it to the Start screen. In this way, the user can tap multiple items and pin them all to Start, as shown in Figure 12-6. When the user taps one of the pinned tiles, this launches the app and navigates her to that page, with the corresponding data loaded.



FIGURE 12-6 Pinning multiple tiles from the same app.

Some apps in the Store adopt the practice of providing an “unpin” feature in addition to the pin feature. As you’ve seen from the previous example, the platform API does expose methods for programmatically deleting (and therefore, unpinning) secondary tiles. However, this is not strictly compliant with Windows Phone app guidelines. The preferred behavior is that the user experience (UX) is always very predictable and very simple. The user knows that she can always unpin any tile that she’s pinned to the Start screen; she knows that this is the standard approach for doing this. So, even though an app could provide an alternative mechanism for unpinning tiles, there’s really no need. That said, one scenario for which it does make sense is when you delete tiles that the user has implicitly requested to be deleted. For example, if a weather app has a list of cities that the user is following and she’s pinned some of these to Start, the app should delete the corresponding tile if the user removes a city from her favorites/watch list.

The viewmodel for each item is very simple: just a *Uri* for the image, and two strings.

```
public class ItemViewModel
{
    public Uri Photo { get; set; }
    public String Title { get; set; }
    public String Details { get; set; }
}
```

The *MainViewModel* is exposed as a static property of the *App* class, and some dummy data is loaded when this property is first accessed. See Chapter 4, “Data Binding and MVVM,” for details of this design (or examine the sample code). The *ListBox.SelectionChanged* handler in the *MainPage* navigates to the *DetailsPage* and then passes in a query string that includes an identifier for the selected item.

```
private void PhotoList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (PhotoList.SelectedIndex != -1)
    {
        String targetUrl = String.Format(
            "/DetailsPage.xaml?Title={0}",
            ((ItemViewModel)PhotoList.SelectedItem).Title);
        NavigationService.Navigate(new System.Uri(targetUrl, UriKind.Relative));
        PhotoList.SelectedIndex = -1;
    }
}
```

All the interesting work is in the *DetailsPage*. The XAML defines an *Image* control for the item photo and a *TextBlock* inside a *ScrollViewer* (to accomodate large amounts of text in the item’s *Details* property). The *Image* and the *TextBlock* are data-bound to the item properties. The app bar has one button that displays a “pin” image. This will be conditionally enabled depending on whether the user has already pinned this item to the Start screen.

```
<Grid x:Name="LayoutRoot" Background="Transparent" d:DataContext="{Binding Items[0]}">
...

    <ScrollViewer
        Grid.Row="1" Margin="12,0,12,0"
        VerticalScrollBarVisibility="Auto" ManipulationMode="Control">
        <StackPanel >
            <Image
                Height="300" Source="{Binding Photo}"
                Stretch="UniformToFill" Margin="12,0,0,0"/>
            <Grid Height="12"/>
            <TextBlock
                Text="{Binding Details}" TextWrapping="Wrap"
                Margin="{StaticResource PhoneHorizontalMargin}" />
        </StackPanel>
    </ScrollViewer>
</Grid>

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" Opacity="0.8">
        <shell:ApplicationBarIconButton
            x:Name="appBarPin" IconUri="/Images/Pin.png" Text="pin to start"
            Click="appBarPin_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

To apply this pinning behavior, you override the *OnNavigatedTo* method in the *DetailsPage*. First, you need to figure out to which item to data-bind, based on the query string parameters. In the process, you formulate a string that you can use later as the URI for this page. This is cached in a field object so that it is available across multiple methods. You also need to determine if there's an active tile for this item. If so, disable the app bar button; otherwise, you need to enable it.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    itemTitle = NavigationContext.QueryString["Title"];
    var pinnedItem = App.ViewModel.Items.FirstOrDefault(x => x.Title == itemTitle);
    if (pinnedItem != null)
    {
        DataContext = thisItem = pinnedItem;
    }

    thisPageUri = e.Uri.OriginalString;
    tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains(thisPageUri));

    appBarPin = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    if (tile != null)
    {
        appBarPin.IsEnabled = false;
    }
    else
    {
        appBarPin.IsEnabled = true;
    }
}
```



Note To handle the scenario in which one title is a substring of another title (and thus would cause a false match), it is useful to add some kind of sentinel value to the end of the URI string, such as "&end=here" or just a relatively unique string like "|~|".

In the *Click* handler for the pin button, create the tile. In this method, you can rely on the fact that you've already performed the search for the tile in the *OnNavigatedTo* override and established that it doesn't exist (otherwise, you wouldn't be in this *Click* handler). So, go ahead and create it now by using the current item's *Photo* property and *Title* property as the *BackgroundImage* and *Title* for the tile.

```
private void appBarPin_Click(object sender, EventArgs e)
{
    DateTime now = DateTime.Now;
    FlipTileData tileData = new FlipTileData
```



```

{
    Title = thisItem.Title,
    Count = now.Second,
    BackgroundImage = thisItem.Photo,
    SmallBackgroundImage = thisItem.Photo,
    WideBackgroundImage = thisItem.Photo,
    BackTitle = "Lorem Ipsum!",
    BackContent = now.ToLongTimeString(),
    BackBackgroundImage = new Uri("/Assets/FlipTileMediumBack.png", UriKind.Relative),
    WideBackContent = "Lorem ipsum dolor sit amet, consectetur adipisicing elit.",
    WideBackBackgroundImage = new Uri("/Assets/FlipTileLargeBack.png", UriKind.Relative),
};
ShellTile.Create(new Uri(thisPageUri, UriKind.Relative), tileData, true);
}

```

By doing this, the user can pin multiple tiles, with individual control over each tile and appropriate UI feedback (the pin button is conditionally enabled) so that it's clear what the pinned state of each item is. When he taps a pinned tile, the app launches and navigates to the item page specified in the tile's *NavigationUri*. This does not go through the *MainPage*; therefore, if he then taps the Back button, there are no more pages for this app in the navigation backstack, so the app terminates.

There is an alternative UX model whereby the user can always return to the main page—and therefore, to the rest of the app—regardless of whether he launched the app from Start in the normal way or from a pinned tile. This model uses a “home” button, but you should use it with care because it varies from the standard, expected behavior. Normally, the user model of a home button is not commonly employed, because it can result in a confusing navigation experience. However, the pinned tile technique gives the user two different ways to start the app. It can justify the decision, ensuring that he can always navigate from the individual item page back to the rest of the app.



Note The practice of providing a “home” feature, as demonstrated in this sample, is pushing the boundaries of Windows Phone app guidelines. A good rule of thumb is to use the built-in apps as your inspiration. For example, with the People Hub, you can pin individual people to the Start screen, but it does not provide a home button to return you to the all people list. The same is true of “music+videos”, and so on.

If you do want to implement this behavior, you can add a second app bar button to the *Details Page*, as shown in Figure 12-7. This technique is illustrated in the *PinTiles_Home* solution in the sample code.



FIGURE 12-7 You should consider carefully whether a home button is good or bad.

You implement the *Click* handler for the home button to navigate to the main page.

```
private void appBarHome_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(new System.Uri("/MainPage.xaml", UriKind.Relative));
}
```

This ensures that the user can always consistently navigate from an item page back to the main page. However, it now introduces a different problem. Consider the following scenario. The user navigates from the Start screen, through a pinned tile to an item page, and then navigates to the main page. This means that pressing Back from the main page will go back to the item page. This is not what the user normally anticipates: her expectation is that pressing Back from the app's main page always exits the app. Fortunately, it is very easy to fix this. All you need do is to ensure that the main page is always the top of the page stack for this app by overriding the *OnNavigatedTo* in the main page to clear the in-app page navigation stack, as shown in the following:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    while (NavigationService.CanGoBack)
    {
        NavigationService.RemoveBackEntry();
    }
}
```

In addition to cleaning up the backstack, this app also ensures that the home button is not available unless the user has navigated to the page via the corresponding pinned tile on the Start screen. Using normal navigation within the app, the home button is unnecessary. But, be aware that this might not be true for all apps. You can imagine an app that includes a home button to deal with a deep navigation stack. Enforcing this behavior helps to maintain the UX, where home navigation is a recognizable exception to the normal navigation behavior, and that it only applies in the specific scenario of a pinned tile. Although you can programmatically set the *IsEnabled* state of an *ApplicationBarIconButton*, this class does not expose a *Visibility* property, as do regular controls. Disabling the button is not really good enough; under normal navigation, you should not make this button available at all, not even in a disabled state. This means that you need to implement this additional button programmatically, not in XAML. To do this, you can update the *OnNavigatedTo* method. First, when creating the tile, enhance the page URI to include an additional parameter which indicates that the user navigated to this page from a pinned tile. Then, you can check if the current *NavigationContext* does include this parameter in the query string. If so, you can create the additional home button, and add it to the app bar.

```
private ApplicationBarIconButton appBarHome;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    itemTitle = NavigationContext.QueryString["Title"];
    var pinnedItem = App.ViewModel.Items.FirstOrDefault(x => x.Title == itemTitle);
    if (pinnedItem != null)
    {
        DataContext = thisItem = pinnedItem;
    }

    thisPageUri = String.Format("/DetailsPage.xaml?Title={0}&Nav=FromPinned", itemTitle);
    tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains(thisPageUri));

    appBarPin = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    if (tile != null)
    {
        appBarPin.IsEnabled = false;
    }
    else
    {
        appBarPin.IsEnabled = true;
    }

    // Did the user get here from a pinned tile?
    if (NavigationContext.QueryString.ContainsKey("Nav"))
    {
        appBarHome = new ApplicationBarIconButton();
        appBarHome.Text = "home";
        appBarHome.IconUri = new Uri("/Assets/Home.png", UriKind.Relative);
        appBarHome.Click += appBarHome_Click;
        ApplicationBar.Buttons.Add(appBarHome);
    }
}
```

Finally, keep in mind that this is one scenario in which it can be useful during debugging to change your *WMAppManifest.xml* file to have the app launched with a specific page and query string, as opposed to the default page. This is a debugging technique, and you must remember to remove the fake navigation before submitting your app to the marketplace. Notice that special characters—such as the ampersand (“&”) in the query string must be escaped as “&”—because they occur within the XML.

```
<Tasks>
  <!--<DefaultTask Name="_default" NavigationPage="MainPage.xaml" />-->
  <DefaultTask Name="_default" NavigationPage="DetailsPage.xaml?Title=Dictumst&Nav=FromPinned"/>
</Tasks>
```

Cross-Targeting Windows Phone 7

If you need to build an app that cross-targets both Windows Phone 7 and Windows Phone 8, you must restrict your code to using the lowest common denominator. In the context of tiles, this means using the old *TemplateType5* in the manifest, and the *StandardTileData* class in code, as shown in the code example that follows. You would also use the old overload of *ShellTile.Create*, which does not take the *bool* parameter for wide-format support. There is only one tile size in version 7, 173x173 pixels, but it is capable of flipping. The flip style in Windows Phone 8 evolved from this style.

```
<TemplateType5>
  <BackgroundImageURI IsRelative="true" IsResource="false">Background.png</BackgroundImageURI>
  <Count>0</Count>
  <Title>SecondaryTiles7</Title>
</TemplateType5>

...
private void createTile_Click(object sender, RoutedEventArgs e)
{
    StandardTileData tileData = new StandardTileData
    {
        BackgroundImage = new Uri("/Assets/sunrise_173x173.jpg", UriKind.Relative),
        Title = "Sound Sunrise",
        Count = 1,
        BackTitle = "Sound Sunset",
        BackContent = "Goodnight!",
        BackBackgroundImage = new Uri("/Assets/sunset_173x173.jpg", UriKind.Relative)
    };
    ShellTile.Create(new Uri("/Page2.xaml?ID=SoundTile", UriKind.Relative), tileData);
}
```

Another option you can consider is the “light-up” scenario, in which your Windows Phone 7 app uses reflection to discover whether it is running on Windows Phone 8, and therefore has new tile styles available to it. The code that follows (the *TileLightup* solution in the sample code) shows how this approach works. If the app is running on Windows Phone 8, it creates a *FlipTileData* object; otherwise, it creates a *StandardTileData* object. Some of the properties, such as the *Title* and *Count*, are common to both types, others are unique. A Windows Phone 7 project cannot use Windows Phone 8

types directly, so it uses reflection to determine the *Type* information for the *FlipTileData* class, and invokes its constructor and property setters indirectly.

```
private static Version wp8 = new Version(8, 0);
private static bool IsWp8 { get { return Environment.OSVersion.Version >= wp8; } }

private void createTile_Click(object sender, RoutedEventArgs e)
{
    DateTime now = DateTime.Now;
    int commonCount = now.Second;
    string commonTitle = "Sound Sunrise";
    string commonBackTitle = "Sound Sunset";
    string commonBackContent = now.ToLongTimeString();
    Uri commonTileId = new Uri("/Page2.xaml?ID=SoundTile", UriKind.Relative);

    if (IsWp8)
    {
        Type flipTileDataType = Type.GetType(
            "Microsoft.Phone.Shell.FlipTileData, Microsoft.Phone");
        Type shellTileType = typeof(ShellTile);

        var tileData = Activator.CreateInstance(flipTileDataType);
        SetProperty(tileData, "Title", commonTitle);
        SetProperty(tileData, "Count", commonCount);
        SetProperty(tileData, "BackTitle", commonBackTitle);
        SetProperty(tileData, "BackContent", commonBackContent);
        SetProperty(tileData, "BackgroundImage",
            new Uri("/Assets/sunrise_336x336.png", UriKind.Relative));
        SetProperty(tileData, "SmallBackgroundImage",
            new Uri("/Assets/sunrise_159x159.png", UriKind.Relative));
        SetProperty(tileData, "WideBackgroundImage",
            new Uri("/Assets/sunrise_691x336.png", UriKind.Relative));
        SetProperty(tileData, "BackBackgroundImage",
            new Uri("/Assets/sunset_336x336.png", UriKind.Relative));
        SetProperty(tileData, "WideBackBackgroundImage",
            new Uri("/Assets/sunset_691x336.png", UriKind.Relative));
        SetProperty(tileData, "WideBackContent", "Lorem ipsum dolor sit amet,
            consectetur adipisicing elit, sed do eiusmod tempor quo.");

        MethodInfo createMethod = shellTileType.GetMethod("Create",
            new Type[] { typeof(Uri), typeof(ShellTileData), typeof(bool) });
        createMethod.Invoke(null, new object[] { commonTileId, tileData, true });
    }
    else
    {
        StandardTileData tileData = new StandardTileData
        {
            Title = commonTitle,
            Count = commonCount,
            BackTitle = commonBackTitle,
            BackContent = commonBackContent,
            BackgroundImage =
                new Uri("/Assets/sunrise_173x173.jpg", UriKind.Relative),
            BackBackgroundImage =
                new Uri("/Assets/sunset_173x173.jpg", UriKind.Relative)
        };
    }
}
```

```

        ShellTile.Create(commonTileId, tileData);
    }

private void SetProperty(object instance, string name, object value)
{
    MethodInfo setMethod = instance.GetType().GetProperty(name).GetSetMethod();
    setMethod.Invoke(instance, new object[] { value });
}

```



Note Reflection is a potentially dangerous technique; therefore, you should use it with caution. The bottom line is that reflection against undocumented APIs is dangerous and should not be used. On the other hand, reflection on documented APIs is acceptable (although, of course, you forgo any sanity checks performed by the compiler). There are a few well-defined scenarios, such as tile light-up, for which the use of reflection is appropriate.

Push Notifications

Part of the delight users derive from Windows Phone is that apps keep themselves up-to-date. Not only can an app update its tiles dynamically, but it can also get updated data on demand from remote servers. This data can be used to update tiles, show toast notifications, and for any other purpose internal to the app. For remote updates, Windows Phone uses a push notification system, so named because the remote server pushes data to the phone only when it is updated rather than waiting for the app to pull the data from the server. Push notifications were introduced in Windows Phone 7, and there has been very little change in the feature set in the transition to Windows Phone 8.

The underlying philosophy of the push system is a *shoulder tap* mechanism. That is, you use a push notification as a signal to the phone/app that there is new data available, and potentially provide some information which hints at what it is. In some cases, the notification contains all the data required for the scenario. In other cases, the notification contains only a minimal amount of data, and the full data payload isn't delivered until the app pulls it, which is typically done via a web service call or a *WebClient* request. This is one reason why the size of push payloads is limited (the other reason, of course, is to minimize network data usage).

The Microsoft Push Notification Service (MPNS) handles many of the common server-side features of this model, including identifying the target devices, retry and queuing logic to allow for offline targets, and per-app registration for notifications. Your server sends new data as simple HTTP messages to the MPNS, specifying which phones should receive the messages. The MPNS then takes care of propagating the messages and pushing them to all the target devices. Each client app running on the device that wants to receive notifications talks to the MPNS to establish an identity for the device. Figure 12-8 illustrates the high-level workflow.



Note Don't be confused by the fact that the push server example happens to be a Windows Store app. Windows Store apps can use the Windows Push Notification System (WNS) for sending and receiving notifications. This borrows considerably from the MPNS that has been in use for Windows Phone for a couple of years, with a very similar model. However, the WNS and MPNS are separate systems; they cannot be cross-purposed. That is, while you can build a Windows Store app to send either WNS or MPNS notifications (or both), Windows Phone devices can only receive MPNS notifications, and Windows Store apps can only receive WNS notifications.

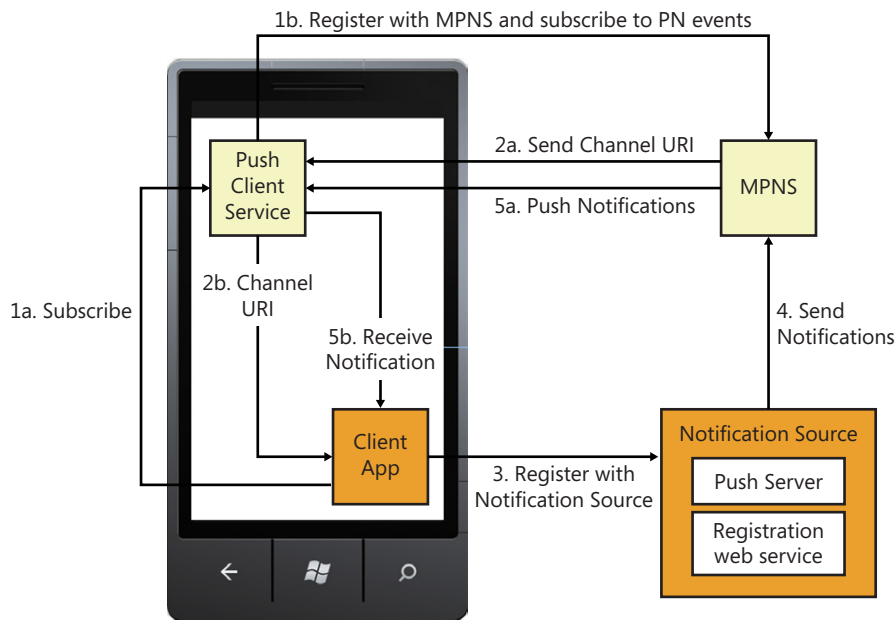


FIGURE 12-8 An overview of the Push Notification architecture.

The two darker boxes in Figure 12-8 represent pieces that you build; the two lighter boxes are supplied by Microsoft. The numbered items are described in more detail, as follows:

1. Your phone app initiates the communication by registering with the MPNS. This is a simple matter of using *HttpNotificationChannel.Open* (1a). Behind the scenes, this uses the Push Client service running on the device to communicate with the MPNS (1b).
2. The MPNS returns a channel URI back to the phone. This URI serves as a unique identifier for the phone; it will be used by the server application as the target URI for a web request. Again, the MPNS actually sends this to the Push Client service on the device (2a), which forwards it to your app (2b).

3. To register for the server's notifications, the phone must send this channel URI to the server application (the app that will send the notifications). The server application can be any kind of application that can make and receive web requests. The server application typically exposes a web service that the phone app can call in order to perform registration.
4. When it's ready to send a notification to a particular device, the server application makes an *HttpWebRequest* for the given channel URI (typically, it does this for multiple registered devices at the same time). This web request goes to the MPNS.
5. The MPNS then pushes the corresponding notification to the specified device or devices. Again, the MPNS actually sends this to the Push Client service on the device (5a), which forwards it to your app (5b).

Whenever the server application sends a notification to the MPNS, it receives a response that provides some information about the result, including the connection status of the target device and whether the notification was actively received or suppressed.

There are three types of push notification, which are described in Table 12-2. The payload for all types of notification must be no more than 3 KB, and additional constraints apply to each type. There's a limit of one push notification channel per app, and this channel will be used for all types of notification. The MPNS has a daily limit of 500 pushes per channel—this is per app/device combination, not 500 in total. Keep in mind that this limit doesn't apply if you create an authenticated push channel, as described later in this chapter. There is no limit to the number of push notification channels per device.

TABLE 12-2 Push Notification Types

Type	Description	Constraints	Typical Scenario
Tile	This is handled by the phone shell and rendered on the Start screen when the tile is pinned to Start. The display includes multiple customizable items, all specified in the tile notification received on the phone. The images must be either local to the phone app itself or specify a reachable HTTP URL.	The title can be any length, but only the first ~15 characters of the title will be displayed. Images are scaled to the tile size chosen by the user. They can be either JPG or PNG format. The Count is capped at 99. Any remote image must be ≤150 KB and must download in ≤30 seconds.	Status updates; for example, count of unread emails for an email client, current temperature for a weather app.
Toast	Include a title, a message body and a target page URI. If your app is not running or is obscured, the phone OS will display a popup toast at the top of the screen for 10 seconds, including both the title and the message. The user can tap the toast to launch the app. If your app is running and not obscured, there is no default display and it's up to your app to handle the message, as appropriate.	If you supply only a title, this is capped at ~40 characters. If you supply only a body, this is capped at ~47 characters. If you supply both, the combined total is capped at ~41 characters.	Breaking news, alerts, instant messaging apps.
Raw	No default visual display. With a raw push notification, you can send any arbitrary text or binary data (up to the overall 3 KB limit) to your app on the phone.	This can only be received when the app is running.	Arbitrary data for use in your app.

To build a solution that uses push notifications, you need two main pieces: the server-side application that generates and sends the notifications, and the client-side app that runs on the phone to receive and process incoming notifications. The client-side app is an optional piece because you might send only tile notifications that do not require client-side code to process them. Both server and client pieces are explored in the following sections.

Push Notification Server

The *PushSimpleServer* solution in the sample code is an example of a server that sends all three types of notification, as illustrated in Figure 12-9. The server is a simple Windows Store app that offers a user interface with which the user can enter suitable values for the various parts of the three notification types. The “server” in this context doesn’t mean a big computer in a data center, it just means an app or service that sends push messages to the phone.

There are two broad sets of functionality that you need to expose from the server: the code that generates and sends the notifications, and the code with which client apps can register to receive notifications. Although it is possible to incorporate both features into one app, in the real world these will most likely be separate server-side components. Building and testing push-based apps can sometimes be complicated by corporate firewalls and proxies; in particular, this affects web services. For this reason, the first version of the server application does not include a registration web service. Instead, when the client first registers with the MPNS and is given a channel URI, you can copy-and-paste this from the client debug session into the running server app (into the *target device uri* *TextBox*). Later, you will see how to layer on a more realistic registration service. Be aware that the emulator will be given a different channel URI each time you reboot it, and potentially at other times also.

Push Simple Server

raw

message

this is a raw message

push

tile

id (uri)

/DetailsPage.xaml?Title=Maeccnas

title

10:14:10

count

10

small background image

Assets/Rhodos_159x159.jpg

background image

Assets/Rhodos_336x336.jpg

wide background image

Assets/Rhodos_691x336.jpg

back title

flip side

back content

lorem ipsum

wide back content

dolor sit amet

back background image

Assets/Mountain_336x336.jpg

wide back background image

Assets/Mountain_691x336.jpg

push

target device uri

http://sn1.notify.live.net/throttledthirdparty/01.00/AAFWEHhBrPluR6ttBq9l6qUvAqAAAAADAQAA/AAQUZm52OkCMjg1OTg1OkZDMkUxREO

response

Raw: OK, Connected, Active, Received

Toast: OK, Connected, Active, Received

Tile: OK, Connected, Active, Suppressed

FIGURE 12-9 A simple Windows Store Push Notification server.

For some of the message elements—such as the raw message body, the tile title and count, and so on—the data is constructed entirely on the server and is independent of anything on the client. For others—such as the toast and tile target URI elements—the server data corresponds to some entity on the client. In this example, certain elements, notably the image URIs for the various tile backgrounds, the string passed from the server corresponds to resources known to exist in the client app. This will not always be the case, particularly for image paths, because the system does support remote image paths.

The app also implements a response list with which the server reports on status and responses from notifications that have been sent. The values in the response report are the response *StatusCode* and the values from the *X-DeviceConnectionStatus*, *X-SubscriptionStatus*, and *X-NotificationStatus* response headers.

In the server, the *MainPage* declares string templates for the toast and tile messages. Each type of notification is formatted as XML, with different elements and attributes for the different types of notification. This example sets up the XML payload by using simple text templates. This is useful as a learning exercise to understand the payload format and content. The templates use "{n}" placeholders for the variable data, to be completed by regular string formatting. Later on, you will see how to achieve this in a more robust manner.

```
const String toastTemplate =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\">" +
        "<wp:Toast>" +
            "<wp:Text1>{0}</wp:Text1>" +
            "<wp:Text2>{1}</wp:Text2>" +
            "<wp:Param>{2}</wp:Param>" +
            "</wp:Toast>" +
        "</wp:Notification>";

const String tileTemplate =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\" Version=\"2.0\">" +
        "<wp:Tile Id=\"{0}\" Template=\"FlipTile\">" +
            "<wp:SmallBackgroundImage>{1}</wp:SmallBackgroundImage>" +
            "<wp:WideBackgroundImage>{2}</wp:WideBackgroundImage>" +
            "<wp:WideBackBackgroundImage>{3}</wp:WideBackBackgroundImage>" +
            "<wp:WideBackContent>{4}</wp:WideBackContent>" +
            "<wp:BackgroundImage>{5}</wp:BackgroundImage>" +
            "<wp:Count>{6}</wp:Count>" +
            "<wp:Title>{7}</wp:Title>" +
            "<wp:BackBackgroundImage>{8}</wp:BackBackgroundImage>" +
            "<wp:BackTitle>{9}</wp:BackTitle>" +
            "<wp:BackContent>{10}</wp:BackContent>" +
        "</wp:Tile>" +
    "</wp:Notification>";
```

The *MainPage* constructor initializes the *TextBox* controls with some dummy data. Notice that the URI fields can include query strings, which can include name-value pairs. You can pass any name-value pairs that make sense in your client app, but the ampersand character (“&”) must be escaped by using the “&” entity.

```
toastUri.Text = "/DetailsPage.xaml?Title=Habitant&amp;Foo=Bar";
```

The app implements suitable button *Click* handlers to trigger sending each of the three notification types. Each handler invokes a custom *SendNotification* method, which has all the common code for sending a notification of any type. The parameters that you pass to this method distinguish the different types and therefore govern how the XML payload is ultimately composed. The payload must include the notification type—this is 1 for tiles, 2 for toasts, and 3 for raw messages. It is common to define an enum for these values, but the simple numeric values are retained here to make it clear that this is what the system uses under the hood. As you can see from the code following code, raw messages are simply sent directly, whereas toast and tile message data is composed by using the predefined template strings:

```
private void sendRaw_Click(object sender, RoutedEventArgs e)
{
    SendNotification(rawMessage.Text, 3);
}

private void sendToast_Click(object sender, RoutedEventArgs e)
{
    String message = String.Format(
        toastTemplate, toastTitle.Text, toastMessage.Text,
        toastUri.Text);
    SendNotification(message, 2);
}

private void sendTile_Click(object sender, RoutedEventArgs e)
{
    DateTime now = DateTime.Now;
    tileTitle.Text = now.ToString("hh:mm:ss");
    tileCount.Text = now.Second.ToString();
    String message = String.Format(
        tileTemplate, tileId.Text, tileSmallBackground.Text,
        tileWideBackground.Text, tileWideBackBackground.Text,
        tileWideBackContent.Text, tileBackground.Text,
        tileCount.Text, tileTitle.Text, tileBackBackground.Text,
        tileBackTitle.Text, tileBackContent.Text);
    SendNotification(message, 1);
}
```

The *SendNotification* method sets up an *HttpClient* for the notification and adds the required message headers. All messages must have a unique ID and must include the notification type. For toast and tile notifications, you also need to add the *X-WindowsPhone-Target* header; however, this is not used for raw notifications. The toast target specifier is “toast”, whereas the tile target specifier is “token” (tiles used to be called tokens, internally). Once you’ve composed the appropriate notification payload, you send the message to the target device via the *PostAsync* method, using the *await* keyword to wait for the return.

The return will be an *HttpResponseMessage*, in which much of the push-specific data will be in custom headers. The app extracts the notification type, status code, connection, subscription, and notification status header values so that they can be reported in the UI by adding them to the response *ListBox*.

```
private async void SendNotification(String message, short notificationClass)
{
    String responseText;
    if (message.Length > 3072)
    {
        responseText = String.Format("The message must be <= 3072 bytes: {0}", message);
    }
    else
    {
        HttpClient request = new HttpClient();

        // Add message headers.
        request.DefaultRequestHeaders.Add("X-MessageID", Guid.NewGuid().ToString());
        request.DefaultRequestHeaders.Add("X-NotificationClass", notificationClass.ToString());

        if (notificationClass == 1)
        {
            request.DefaultRequestHeaders.Add("X-WindowsPhone-Target", "token");
        }
        else if (notificationClass == 2)
        {
            request.DefaultRequestHeaders.Add("X-WindowsPhone-Target", "toast");
        }

        try
        {
            // Send the message, and wait for the response.
            HttpResponseMessage response = await request.PostAsync(
                targetDeviceUri.Text, new StringContent(message));

            IEnumerable<string> values;
            String connectionStatus = String.Empty;
            if (response.Headers.TryGetValues("X-DeviceConnectionStatus", out values))
            {
                connectionStatus = values.First();
            }
            String subscriptionStatus = String.Empty;
            if (response.Headers.TryGetValues("X-SubscriptionStatus", out values))
            {
                subscriptionStatus = values.First();
            }
            String notificationStatus = String.Empty;
            if (response.Headers.TryGetValues("X-NotificationStatus", out values))
            {
                notificationStatus = values.First();
            }
        }
    }
}
```

```

        responseText = String.Format("{0}: {1}, {2}, {3}, {4}",
            notificationClass == 1 ? "Tile" :
            notificationClass == 2 ? "Toast" : "Raw",
            response.StatusCode,
            connectionStatus, subscriptionStatus, notificationStatus);
    }
    catch (WebException ex)
    {
        responseText = ex.Message;
    }
}
responseList.Items.Add(responseText);
}

```

Push Notification Client

In the client, raw and toast notifications are handled by the app if the app is running and not obscured. If the app is not running, toast notifications are rendered by the platform shell as pop-up windows. The shell always handles tile notifications.

Figure 12-10 (left) shows the client app running. This is the *PushSimpleClient* solution in the sample code, which is an evolution of the *PinTiles* sample discussed earlier. Here, the notification status and any incoming toast notification is reported in a *ListBox* at the bottom of the main page. Contrast this with Figure 12-10 (right), which shows the Start screen when a toast notification is received. This displays the app's icon and the incoming toast *Title* and *Message* values. When the user taps the toast, this navigates to the target URI specified in the toast notification payload. Any additional parameters in the query string are passed in to the target page in the *NavigationContext*.

When the user taps the toast, she navigates to the specified page—in this example, to the *DetailsPage* with the *Habitant* item loaded. The fact that there is a pinned tile for the *DetailsPage* with the *Maecenas* item is irrelevant at this point. On the client side, the *DetailsPage* code is exactly the same as in the earlier *PinTiles* sample. The existing *OnNavigatedTo* override, which was designed to meet the needs of navigation via local tile, also meets the needs of navigation via URI-targeted toast notification, and, of course, navigation via a tile that might or might not have been updated via a push notification. The only update on the client side for this toast is to extract the additional query string values and display them in the UI, in the scenario in which the toast is received when the app is actually running.

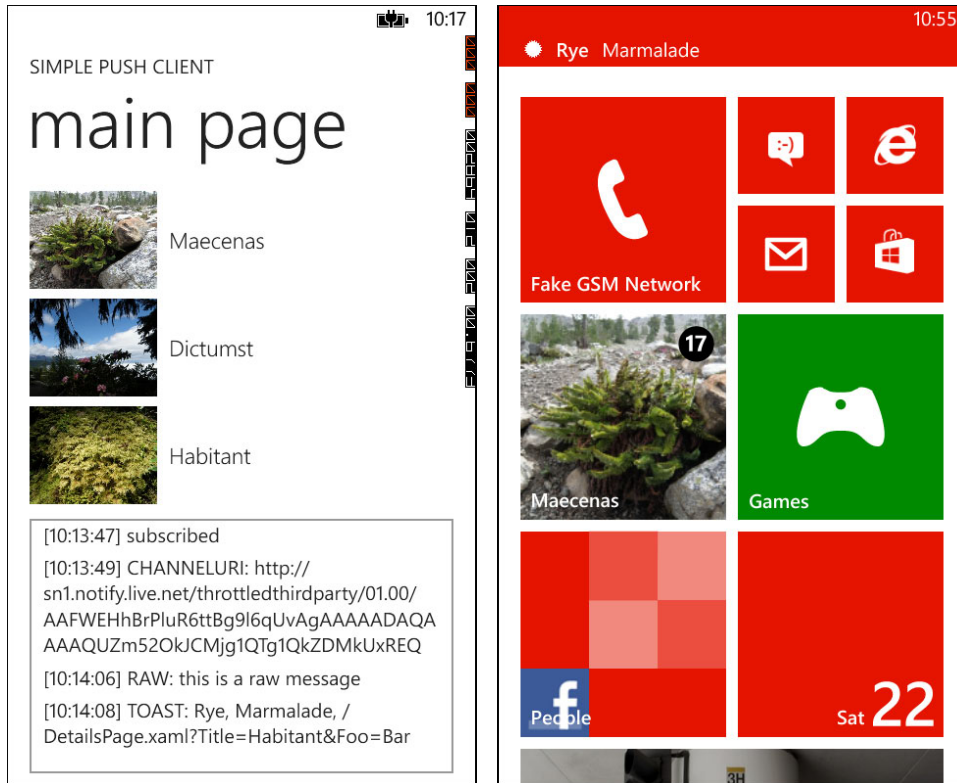


FIGURE 12-10 The client app receiving a toast notification while running (on the left), and the phone receiving a toast while the client app is not running (right).

Now, consider what happens when a tile notification is received. Figure 12-10 shows that the user has pinned a secondary tile for the *Maecenas* item. If a tile notification arrives that includes this corresponding tile ID—that is `/DetailsPage.xaml?Title=Maecenas`—the tile information will be used to update that specific tile. If the tile is not pinned, the notification is simply suppressed. If the server sends a tile notification and omits the *ID*, this will update the default (primary) tile—again, assuming that this tile is pinned to Start.

The client app needs to perform the following tasks:

1. Subscribe to receive notifications, via the Push Client service on the phone.
2. Bind toast and tile notifications to this app so that when these are received by the shell, the shell can associate them with the app.
3. When the MPNS sends this app a unique push client ID, the app would realistically then register this with a registration web service on the server. However, this first version of the client doesn't include remote registration; instead, it relies on copy-paste during debugging.
4. Process any incoming raw and toast notifications that arrive when the app is running.

Note that an app that uses push notifications must have the *ID_CAP_PUSH_NOTIFICATION* capability declared in its app manifest. The *MainPage* declares fields for the channel name (an arbitrary string), the *HttpNotificationChannel* for working with the MPNS, and an *ObservableCollection<T>* to hold all the message strings and status information. This collection is data-bound to the *ListBox* in the UI. The app also declares a simple *bool* which it uses to ensure that the notification events are only hooked up to event handlers once. This is important, because the hook-up is done in the *OnNavigatedTo* override, and this method, of course, can be invoked multiple times.

```
private String channelName = "Contoso Notification Channel";
private HttpNotificationChannel channel;
private Uri channelUri;
private bool isConnected;
private ObservableCollection<String> notifications;
public ObservableCollection<String> Notifications
{
    get { return notifications; }
    private set { }
}
```

The *OnNavigatedTo* override is a reasonable place to perform initialization. The first thing to do is to try to find an existing push channel. If none is found, go ahead and create a new one now. Behind the scenes, this triggers the Push Client component on the phone to request a device identifier (channel URI) from the MPNS. When the MPNS sends the channel URI back to the device, the Push Client component raises a *ChannelUriUpdated* event. So, this is the first push event for which the client must set up a handler. Two other events the client handles are the *ShellToastNotificationReceived* (self-explanatory) and the *HttpNotificationReceived* (which is raised when a raw notification is received). If the channel hasn't already been bound to tile and toast notifications, you can bind it now by using the *BindToShellTile* and *BindToShellToast* APIs. The app reports ongoing status and other messages in the UI via an *AddNotification* method that updates the data-bound *ListBox* on the UI thread.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    channel = HttpNotificationChannel.Find(channelName);
    if (channel == null)
    {
        channel = new HttpNotificationChannel(channelName);
        channel.ChannelUriUpdated += channel_ChannelUriUpdated;
        channel.Open();
    }

    if (!isConnected)
    {
        channel.ShellToastNotificationReceived += channel_ShellToastNotificationReceived;
        channel.HttpNotificationReceived += channel_HttpNotificationReceived;
        isConnected = true;
    }

    if (!channel.IsShellTileBound)
    {
        channel.BindToShellTile();
    }
}
```

```

        if (!channel.IsShellToastBound)
        {
            channel.BindToShellToast();
        }
        AddNotification("subscribed");
    }

    private void AddNotification(String message)
    {
        Dispatcher.BeginInvoke(() => Notifications.Add(
            String.Format("[{0:hh:mm:ss}] {1}", DateTime.Now, message)));
    }

```

You should allow for the possibility of a server restart while the client has a channel open. Of course, your client app won't know when this has happened, but you can mitigate it somewhat by always registering the channel during *OnNavigatedTo*. If you were to receive a *ChannelUriUpdate* event, you would realistically call the server app's web service in order to register this client, passing it the channel URI you've been given by the MPNS. This first version of the client simply prints this out to the output window in a Visual Studio debug session. From there, you can manually copy it and paste it into the Target Device Uri *TextBox* in the server app.

```

private void channel_ChannelUriUpdated(object sender, NotificationChannelUriEventArgs e)
{
    channelUri = e.ChannelUri;
    String message = String.Format("CHANNELURI: {0}", channelUri);
    Debug.WriteLine(e.ChannelUri);
    AddNotification(message);
}

```

When the app receives a push notification (raw or toast) event, it also displays this in the UI, appropriately dispatched to the UI thread. To retrieve the data, you dig into the *Collection* property on the *NotificationEventArgs* parameter that is passed into the event handler. Notification data is sent in this collection, which is a dictionary, so the elements are key-value pairs. In this app, there will only be one pair of data, although you will allow for it being a null element (in which case, you use an empty string).

```

private void channel_HttpNotificationReceived(object sender, HttpNotificationEventArgs e)
{
    byte[] bytes;
    using (Stream stream = e.Notification.Body)
    {
        bytes = new byte[stream.Length];
        stream.Read(bytes, 0, (int)stream.Length);
    }
    String rawMessage = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
    String message = String.Format("RAW: {0}", rawMessage);
    AddNotification(message);
}

```



```
private void channel_ShellToastNotificationReceived(object sender, NotificationEventArgs e)
{
    String title = e.Collection.Values.First();
    String toastMessage = e.Collection.Values.Skip(1).FirstOrDefault() ?? String.Empty;
    String toastUrl = e.Collection.Values.Skip(2).FirstOrDefault() ?? String.Empty;
    String message = String.Format("TOAST: {0}, {1}, {2}", title, toastMessage, toastUrl);
    AddNotification(message);
}
```

Registration Web Service

Of course, copying and pasting the client's channel URI from a debug session into the server app is only useful during development. In production, to support client registration on the server, you can expose a web service with appropriate *Register* (and *Unregister*) web methods. It is actually possible to host this web service within your push server application, but it is more likely to be hosted independently, perhaps by Internet Information Services (IIS), and potentially even on separate physical server computers. This raises the question of how to share information about registered devices between the web service and the push server. Typically this would be done via a shared database; on Windows Azure, this would be done via Azure table storage.

For test purposes, if you don't have a database server or a Windows Azure account, you can simulate a shared database by using a shared file, instead. This is the approach taken in the *Registration Service* web service in the sample code.

The web service itself is a straightforward Windows Communications Foundation (WCF) service that defines a simple *ServiceContract* named *IRegisterDeviceService* and implements it in a custom *RegisterDeviceService* class. Because the push server is a Windows Store app, it must adhere to constraints on filesystem access. The WCF service doesn't have the same restrictions, so for simplicity, the shared file is placed in the install folder for the push server—a location which is accessible to both components. Client phone apps will call *Register* to register their channel URI with this server app. The service loads the shared file (which simulates a more realistic database query), searches the list, adds the device's channel URI if not found, and then overwrites the file with the amended list, using a *DataContractJsonSerializer*. Similarly, client apps will call *Unregister* to unregister their channel URI with this server app, and in this case, the service searches the list, removes the channel URI if found, and then overwrites the file again.

```
[ServiceContract]
public interface IRegisterDeviceService
{
    [OperationContract]
    void Register(Uri deviceUri);

    [OperationContract]
    void Unregister(Uri deviceUri);
}

public class RegisterDeviceService : IRegisterDeviceService
{
```

```

private const string DeviceListFilePath =
    @"C:\temp\Samples12\PushSimpleServer_Registration\bin\Debug\AppX\deviceList.txt";

public void Register(Uri deviceUri)
{
    List<Uri> devices = LoadDeviceListFromFile();
    if (devices != null & !devices.Contains(deviceUri))
    {
        devices.Add(deviceUri);
        using (FileStream writeFile = File.Create(DeviceListFilePath))
        {
            DataContractJsonSerializer serializer =
                new DataContractJsonSerializer(typeof(List<Uri>));
            serializer.WriteObject(writeFile, devices);
        }
    }
}

public void Unregister(Uri deviceUri)
{
    List<Uri> devices = LoadDeviceListFromFile();
    if (devices != null & devices.Contains(deviceUri))
    {
        devices.Remove(deviceUri);
        using (FileStream writeFile = File.Create(DeviceListFilePath))
        {
            DataContractJsonSerializer serializer =
                new DataContractJsonSerializer(typeof(List<Uri>));
            serializer.WriteObject(writeFile, devices);
        }
    }
}

private List<Uri> LoadDeviceListFromFile()
{
    if (!File.Exists(DeviceListFilePath))
    {
        using (FileStream createFile = File.Create(DeviceListFilePath)){}
        return new List<Uri>();
    }
    using (FileStream readFile = File.OpenRead(DeviceListFilePath))
    {
        DataContractJsonSerializer serializer =
            new DataContractJsonSerializer(typeof(List<Uri>));
        return (List<Uri>)serializer.ReadObject(readFile);
    }
}
}

```



Note From Windows Vista onward, security has improved so that if you build and run a WCF service from Visual Studio, you might see this error (where “+” is a wildcard, and “8002” is the arbitrary port you specified for your service):

HTTP could not register URL http://+:8002/. Your process does not have access rights to this namespace (see <http://go.microsoft.com/fwlink/?LinkId=70353> for details).

This is because by default, every HTTP path is reserved for use by processes executed by the system administrator. If you are not running your WCF service as administrator, it will fail to start with an *AddressAccessDeniedException*. To mitigate this, you could run Visual Studio as administrator. Alternatively, you can open a command window (as administrator) and execute the *netsh* command to delegate permissions on your service’s HTTP path to your regular user account (or to Everyone); for example:

```
netsh http add urlacl url=http://+:8002/RegisterDevice user=Andrew
```

The push server app can now load the device list from the shared file. You can see this at work in the *PushSimpleServer_Registration* version of the app in the sample code. The device list itself is held in a *List<Uri>* field. The known path chosen for the shared file corresponds to the *InstalledLocation* property of the current app package. During development, this will be in the *bin\Debug\AppX* folder of the current project. If the device list file doesn’t exist, the app creates it now and returns an empty list. Otherwise, it converts the Windows Runtime (WinRT) input stream into a managed stream, and deserializes it into the list.

```
private List<Uri> deviceList;

private async void LoadDeviceListFromFile()
{
    //C:\temp\Samples12\PushSimpleServer_Registration\bin\Debug\AppX
    StorageFolder folder = Windows.ApplicationModel.Package.Current.InstalledLocation;
    string fileName = "deviceList.txt";
    StorageFile file = await folder.GetFilesAsync(fileName);

    if (file == null)
    {
        file = await folder.CreateFileAsync(fileName);
        deviceList = new List<Uri>();
    }
    else
    {
        IInputStream inputStream = await file.OpenReadAsync();
        Stream readStream = inputStream.AsStreamForRead();
        DataContractJsonSerializer serializer =
            new DataContractJsonSerializer(typeof(List<Uri>));
        deviceList = (List<Uri>)serializer.ReadObject(readStream);
    }
}
```

The app calls this custom *LoadDeviceListFromFile* method after setting up the *HttpClient* object but before invoking *PostAsync*; this is now done in a loop, where the app iterates through all the registered devices, sending the same notification to all of them. Of course, this is also where you would perform any target client filtering, perhaps sending some notifications only to certain devices and not to others, as your domain logic dictates.

```
HttpClient request = new HttpClient();
... add message headers: unchanged code omitted for brevity.
LoadDeviceListFromFile();
foreach (Uri deviceUri in deviceList)
{
    try
    {
        HttpResponseMessage response = await request.PostAsync(
            //targetDeviceUri.Text, new StringContent(message));
            deviceUri, new StringContent(message));

        ... process the response: unchanged code omitted for brevity.
    }
}
```

The client app can be updated to take advantage of the registration web service, as demonstrated in the *PushSimpleClient_Registration* version of the app in the sample code. First, you would add a service reference for the WCF service, in exactly the same way as for any other web service (as described in Chapter 7, “Web and Cloud”). You can then declare an instance of the web service client proxy class as a field in the *MainPage*.

```
private RegisterDeviceServiceClient serviceClient;
```

In the *OnNavigatedTo* override, you would instantiate this service proxy, and hook up a handler for the *RegisterCompleted* event that will be raised when the registration call completes. In this sample, these are implemented to report status in the UI.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    serviceClient = new RegisterDeviceServiceClient();
    serviceClient.RegisterCompleted += serviceClient_RegisterCompleted;
    serviceClient.UnregisterCompleted += serviceClient_UnregisterCompleted;
    ...unchanged code omitted for brevity.
}

private void serviceClient_RegisterCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error == null)
    {
        AddNotification("registered");
    }
    else
    {
        AddNotification(e.Error.Message);
    }
}
```

```
private void serviceClient_UnregisterCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error == null)
    {
        AddNotification("unregistered");
    }
    else
    {
        AddNotification(e.Error.Message);
    }
}
}
```

The client would typically make the registration call as soon as the channel URI is received from the MPNS, in the *ChannelUriUpdated* event handler. You could enhance this so that the app remembers its last URI (for example, by saving it in *AppSettings*), and if this differs from the latest one, send both an *UnRegister* and a fresh *Register* request.

```
private void channel_ChannelUriUpdated(object sender, NotificationChannelUriEventArgs e)
{
    channelUri = e.ChannelUri;
    String message = String.Format("CHANNELURI: {0}", channelUri);
    AddNotification(message);

    // Register the new device Uri with the server app.
    serviceClient.RegisterAsync(channelUri);
}
}
```

The preceding code illustrates all the core requirements for the client and server applications. However, there are a few additional features that you could incorporate to improve robustness and usability, as described in the following sections.

Additional Server Features

On the server side, you can use the following additional features offered by the push notification system:

- Batching intervals, which allow you to group your notifications into batches.
- The *XmlWriter* or *XDocument* classes, for building the notification payload, instead of using string templates.
- Enhanced notification response information, to implement richer message tracking and reporting.

These features are described in the following subsections.

Batching Intervals

The server currently uses strings 1, 2, 3 to identify the *X-NotificationClass*, and these must be added to the request headers. However, the values 1, 2, 3 really correspond to batching indicators, and there are nine possible values, organized into three categories, as shown in Table 12-3.

TABLE 12-3 Notification Batching Intervals

Value	Notification Type	Meaning
1	Tile	Send immediately
2	Toast	Send immediately
3	Raw	Send immediately
11	Tile	Batch and send within 450 seconds (7.5 minutes)
12	Toast	Batch and send within 450 seconds (7.5 minutes)
13	Raw	Batch and send within 450 seconds (7.5 minutes)
21	Tile	Batch and send within 900 seconds (15 minutes)
22	Toast	Batch and send within 900 seconds (15 minutes)
23	Raw	Batch and send within 900 seconds (15 minutes)

This makes it possible for the MPNS to batch notifications together, including from multiple apps. The primary purpose of this is to maintain an optimal balance of UX on the phone; sending notifications in batches improves battery performance because it makes maximum use of the network while it is up, rather than bringing it up for every single notification. The *PushMoreServer* solution in the sample code adds a *ComboBox* to the *MainPage*, with which the user can select one of the three batching intervals. This will be applied to all notifications until it is changed.

```
<ComboBox Name="batchList" Width="350" HorizontalAlignment="Left" FontSize="16">
    <ComboBoxItem Content="immedate" Tag="0"/>
    <ComboBoxItem Content="450 sec" Tag="10"/>
    <ComboBoxItem Content="900 sec" Tag="20"/>
</ComboBox>
```

The *SendNotification* method is updated to accomodate the batching interval. Specifically, change this line

```
request.DefaultRequestHeaders.Add("X-NotificationClass", notificationClass.ToString());
```

to this:

```
int batch = Int16.Parse(((ComboBoxItem)batchList.SelectedItem).Tag.ToString()) +
notificationClass;
request.DefaultRequestHeaders.Add("X-NotificationClass", batch.ToString());
```

Be aware that the MPNS does not provide a true end-to-end confirmation that the notification was delivered. In particular, if you batch up your notifications, you will still get an immediate response notification based on the last known state of the target device and app, even though the notification might not be sent until up to 15 minutes after the fact.

XML Payload

Building the XML payload from string templates is useful from a developer's perspective; it helps to make it obvious what the XML schema is and what a typical payload for each notification type looks like. However, it is a somewhat error-prone approach. For example, it is very fragile in the face of replacement values that contain reserved characters such as "<". A more robust approach is to construct the XML in code by using *XmlWriter* methods *WriteStartElement*, *WriteEndElement*, and so on. Alternatively, you can use the simpler *XDocument*.

Using *XDocument*, you can rewrite the cumbersome XML string-based code for toasts and tiles, as demonstrated in the example that follows. You can see this at work in the *PushMoreServer* solution in the sample code.

```
private static readonly XNamespace WpNs = "WPNotification";
private void sendToast_Click(object sender, RoutedEventArgs e)
{
    XDocument doc = new XDocument();
    doc.Add(
        new XElement(WpNs + "Notification",
            new XAttribute(XNamespace.Xmlns + "wp", WpNs.NamespaceName),
            new XElement(WpNs + "Toast",
                new XElement(WpNs + "Text1", toastTitle.Text),
                new XElement(WpNs + "Text2", toastMessage.Text),
                new XElement(WpNs + "Param", toastUri.Text))));
    String message = doc.ToString();
    SendNotification(message, 2);
}

private void sendTile_Click(object sender, RoutedEventArgs e)
{
    DateTime now = DateTime.Now;
    tileTitle.Text = now.ToString("hh:mm:ss");
    tileCount.Text = now.Second.ToString();
    XDocument doc = new XDocument();
    doc.Add(
        new XElement(WpNs + "Notification",
            new XAttribute(XNamespace.Xmlns + "wp", WpNs.NamespaceName),
            new XElement(WpNs + "Tile", new XAttribute(WpNs + "Id", tileId.Text),
                new XElement(WpNs + "SmallBackgroundImage", tileSmallBackground.Text),
                new XElement(WpNs + "WideBackgroundImage", tileWideBackground.Text),
                new XElement(WpNs + "WideBackBackgroundImage", tileWideBackBackground.Text),
                new XElement(WpNs + "WideBackContent", tileWideBackContent.Text),
                new XElement(WpNs + "BackgroundImage", tileBackground.Text),
                new XElement(WpNs + "Count", tileCount.Text),
```

```

        new XElement(WpNs + "Title", tileTitle.Text),
        new XElement(WpNs + "BackBackgroundImage", tileBackBackground.Text),
        new XElement(WpNs + "BackTitle", tileBackTitle.Text),
        new XElement(WpNs + "BackContent", tileBackContent.Text))));
String message = doc.ToString();
SendNotification(message, 1);
}

```

Response Information

The server app has logic to test for connected subscribers, so for each notification sent, the results are most likely to be *status code = OK*, *connection status = Connected*, or *subscription status = Active*. The notification status will be either *Received* or *Suppressed*. Raw notifications are received if the app is running; otherwise, they are suppressed. Tile notifications are received if the app is pinned to the start menu and is not running; otherwise, they are suppressed. Toast notifications are received whether the app is running or not, and the platform will show toast UI if the app is not in the foreground.

The server app reports the most useful notification response information in the UI. In a more sophisticated app, you might well want to track other information such as the message ID and time-stamp. Most of the potentially useful information is in the *Headers* property of the *HttpResponseMessage* object that you get back in the server after sending a notification (as listed in the example that follows). Be aware that only the items prefixed with "X-" are specific to push notifications and that the headers include some MPNS server information that is of no practical use to the developer.

```

X-DeviceConnectionStatus - Connected
X-NotificationStatus - Received
X-SubscriptionStatus - Active
X-MessageID - cb4f3a3b-4c2c-4363-8f5f-85458d11ed56
ActivityId - c56ab764-6d73-439e-b4a1-bbc27940c5d4
X-Server - SN1MPNSM019
Cache-Control - private
Date - Tue, 25 Sep 2012 00:05:14 GMT
Server - Microsoft-IIS/7.5
X-AspNet-Version - 4.0.30319
X-Powered-By - ASP.NET

```

Additional Client Features

On the client side, there are several enhancements that you should consider layering on top of the basic push features, including the following:

- Handling the special *ErrorOccurred* push notification event
- Providing a mechanism for the user to opt in or out of push notifications for your app
- Implementing a custom ViewModel for push settings

The *ErrorOccurred* Event

So far, the sample app has used a reasonable level of try/catch exception handling, including for standard HTTP web error codes. However, the MPNS also reports specific errors that you can consume in your app. To do this, in the *SubscribeToNotifications* method, add an event sink for the *ErrorOccurred* event, as shown in the following example (the *PushMoreClient* solution in the sample code):

```
if (channel == null)
{
    channel = new HttpNotificationChannel(channelName);
    channel.ChannelUriUpdated += channel_ChannelUriUpdated;
    channel.ErrorOccurred += channel_ErrorOccurred;
    channel.Open();
}
```

For testing purposes and for a simple implementation, the event handler could report the error—or rather, a suitably user-friendly version of the error message—to the screen. In a more sophisticated app, you would want to look at the *ErrorType* and *ErrorAdditionalData* properties and take the appropriate corrective action. For example, if you get a *ChannelOpenFailed* or *PayloadFormatError*, the channel is now useless, so you should clean it up and optionally recreate it. To support that technique, you should abstract all the notification subscription work from the *OnNavigatedTo* override out to a custom method (named *SubscribeToNotifications* in the example code that follows momentarily) and then invoke this method from *OnNavigatedTo*.

On the other hand, if you get bad data from the server or too many notifications in a short span of time, there's really not much you can do on the client, beyond reporting. If you get a *PowerLevelChanged* event, this is an informative warning that the server will stop sending tile and toast notifications. If phone power drops to critical level, MPNS will stop sending even raw notifications to the device to reduce power consumption.

```
private void channel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
{
    String description = String.Empty;
    switch (e.ErrorType)
    {
        case ChannelErrorType.ChannelOpenFailed:
        case ChannelErrorType.PayloadFormatError:
            channel.Close();
            channel.Dispose();
            channel = null;
            SubscribeToNotifications();
            description = "Channel closed and re-initialized.";
            break;
        case ChannelErrorType.MessageBadContent:
            description = "Bad data received from server.";
            break;
        case ChannelErrorType.NotificationRateTooHigh:
            description = "Too many notifications received.";
            break;
    }
}
```

```

        case ChannelErrorType.PowerLevelChanged:
            if (e.ErrorAdditionalData == (int)ChannelPowerLevel.LowPowerLevel)
            {
                description =
                    "No more toast or tile notifications will be "
                    + "received until power levels are restored.";
            }
            else if
                (e.ErrorAdditionalData ==
                 (int)ChannelPowerLevel.CriticalLowPowerLevel)
            {
                description =
                    "No notifications of any kind will be received"
                    + "until power levels are restored.";
            }
            break;
        case ChannelErrorType.Unknown:
            description = "unknown";
            break;
    }
    AddNotification(String.Format("ERROR: {0} - {1}", e.Message, description));
}

```

User Opt-In/Out

The Store certification requirements include two items that are specific to push notifications. The first time your app uses the *BindToShellToast* method, you must ask the user for explicit permission to receive toast notifications. You must also provide a UI mechanism with which the user can turn off toast notifications at any time later on. The reason for this is that toast notifications use the same alert mechanism as other system notifications, such as incoming Short Message Service (SMS) messages. These alerts are executed at idle-level priority; that is, they will be displayed immediately regardless of anything else the user is doing on the phone at the time, so long as the CPU is not at maximum utilization. It is beneficial for all toast notifications to use the same UI mechanism on the phone because it promotes consistency of UX. The downside is that the user might not consider your app's notifications to have the same priority as system alerts. In addition, notifications consume battery power, and even though the MPNS itself will throttle the rate at which notifications are sent, any potentially excessive use of battery power should also be under the user's control. The user must be given the choice on a per-app basis.

To accommodate this requirement, you need to prompt the user, typically with a *MessageBox*, and then persist the user's choice. You can enhance the client app with a couple of additional *bool* fields to record whether the user wants to allow toasts and whether you've already asked him once. These need to be included in the persistent app settings.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    bool push;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>("IsToastOk", out push))
    {
        isToastOk = push;
    }
}

```

```

        bool prompted;
        if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>("ToastPrompted", out
            prompted))
        {
            toastPrompted = prompted;
        }
        SubscribeToNotifications();
    }

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["IsToastOk"] = isToastOk;
    IsolatedStorageSettings.ApplicationSettings["ToastPrompted"] = toastPrompted;
    IsolatedStorageSettings.ApplicationSettings.Save();
}

```

Some apps only use toast notifications. Also, some apps require both toast notifications and either raw or tile notifications. This means that the user's choice about allowing toasts might determine whether your app uses notifications at all. However, it is more common to keep the different types of notification separate. In the following example (the *PushMoreClient* app in the sample code), you go ahead with raw and tile notifications, regardless, but only use toasts if the user has explicitly agreed to them.

```

private void SubscribeToNotifications()
{
    ...unchanged code omitted for brevity.

    if (!channel.IsShellToastBound)
    {
        if (!isToastOk && !toastPrompted)
        {
            MessageBoxResult pushPrompt =
                MessageBox.Show(
                    "Allow toast notifications for this application?",
                    "PushMoreClient", MessageBoxButton.OKCancel);
            toastPrompted = true;
            if (pushPrompt == MessageBoxResult.OK)
            {
                isToastOk = true;
                channel.BindToShellToast();
            }
        }
    }
}

```

Note that this doesn't cover the case in which *isToaskOk* is already *true*, but the channel URI has changed. In this case, you would also need to rebind.

Implementing a Push ViewModel

You are only required to ask the user about toast notifications once, but you are also required to offer a mechanism by which the user can change his decision at any later stage. This pretty much mandates a settings page of some kind. As soon as you implement a settings page, the limitations of the simple

client implementation you've been working with so far become more obvious, specifically because you now need to access connection information across at least two pages. The classic design solution here is to encapsulate all the connection information into a *ViewModel* class, and declare a public property of that type in the *App* class, where it will be accessible to all pages in the app.

The following is an example (the *PushViewModelClient* solution in the sample code) of the *App* class declaring a static *PushViewModel* property:

```
public partial class App : Application
{
    private static MainViewModel viewModel = null;
    public static MainViewModel ViewModel
    {
        get
        {
            lock (typeof(App))
            {
                if (viewModel == null)
                {
                    viewModel = new MainViewModel();
                    viewModel.LoadData();
                }
            }
            return viewModel;
        }
    }
}
```

All of the connection-related fields and properties are moved from the *MainPage* class to the *PushViewModel* class, along with all the server registration and notification subscription methods. Most of these methods are taken wholesale from the original implementation in *MainPage*.

```
public class PushViewModel : INotifyPropertyChanged
{
    private String channelName = "Contoso Notification Channel";
    private HttpNotificationChannel channel;
    private Uri channelUri;
    private ObservableCollection<String> notifications = new ObservableCollection<String>();
    public ObservableCollection<String> Notifications
    {
        get { return notifications; }
    }

    public void SubscribeToNotifications()
    {
        ...original code moved unchanged from MainPage class.
    }

    private void channel_ChannelUriUpdated(object sender, NotificationChannelUriEventArgs e)
    {
        ...original code moved unchanged from MainPage class.
    }
}
```

```

private void channel_HttpNotificationReceived(object sender, HttpNotificationEventArgs e)
{
    ...original code moved unchanged from MainPage class.
}

private void channel_ShellToastNotificationReceived(object sender, NotificationEventArgs e)
{
    ...original code moved unchanged from MainPage class.
}

private void channel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
{
    ...original code moved unchanged from MainPage class.
}
}

```

This viewmodel adopts the recommended pattern where changes to the critical data are persisted at the point where the change is made rather than waiting until some lifecycle event to load/save all the settings at once. In this particular example, the difference is negligible, but you should generally adopt this pattern by default, and it will certainly make a difference if the amount of data is large.

```

private bool isToastPrompted;
public bool IsToastPrompted
{
    get
    {
        bool prompted;
        if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
            ("IsToastPrompted", out prompted))
        {
            isToastPrompted = prompted;
        }
        return isToastPrompted;
    }
    set
    {
        if (value != isToastPrompted)
        {
            isToastPrompted = value;
            PropertyChangedEventHandler handler = PropertyChanged;
            if (null != handler)
            {
                handler(this, new PropertyChangedEventArgs("IsToastPrompted"));
            }
            IsolatedStorageSettings.ApplicationSettings["IsToastPrompted"] = isToastPrompted;
            IsolatedStorageSettings.ApplicationSettings.Save();
        }
    }
}

private bool isToastOk;

```

```

public bool IsToastOk
{
    get
    {
        bool toast;
        if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
            ("IsToastOk", out toast))
        {
            isToastOk = toast;
        }
        return isToastOk;
    }
    set
    {
        if (value != isToastOk)
        {
            isToastOk = value;
            PropertyChangedEventHandler handler = PropertyChanged;
            if (null != handler)
            {
                handler(this, new PropertyChangedEventArgs("IsToastOk"));
            }

            if (isToastOk)
            {
                if (!channel.IsShellToastBound)
                {
                    channel.BindToShellToast();
                    AddNotification("toasts bound");
                }
            }
            else
            {
                if (channel.IsShellToastBound)
                {
                    channel.UnbindToShellToast();
                    AddNotification("toasts unbound");
                }
            }
            IsolatedStorageSettings.ApplicationSettings["IsToastOk"] = isToastOk;
            IsolatedStorageSettings.ApplicationSettings.Save();
        }
    }
}

```

The notifications will come in on a non-UI thread, but you're data-binding the list of notifications to the UI. To avoid cross-thread exceptions, you need to marshal any updates to the notifications collection to the UI thread. The standard way to achieve this is to use the *Dispatcher* class. Every UI element—in fact, every type derived from *DependencyObject*—has a *Dispatcher* field of type *Dispatcher*. The *PushViewModel* class is not a UI element, but you can use the *Deployment.Current.Dispatcher*, which is always globally available in a phone app.

```
private void AddNotification(String message)
{
    String formattedMessage =
        String.Format("{0:hh:mm:ss}] {1}", DateTime.Now, message);
    if (Deployment.Current.Dispatcher.CheckAccess())
    {
        Notifications.Add(formattedMessage);
    }
    else
    {
        Deployment.Current.Dispatcher.BeginInvoke(
            () => Notifications.Add(formattedMessage));
    }
}
```



Note The *CheckAccess* method on the *Dispatcher* class doesn't show up in IntelliSense or AutoComplete in Visual Studio. This is because it is marked with the *[EditorBrowsable(EditorBrowsableState.Never)]* attribute. The original reason for this is because it was thought that the method would only be used in advanced scenarios. So, although it is part of the public API, it is not advertised in the tools. This is a historical hangover, and not really relevant any more. Certainly, there is no problem in your using this method.

In the *MainPage* class, the *ListBox* is data-bound to the notifications collection, as before, but in this version, the notifications collection is exposed as a property of the *PushViewModel*, which itself is a property of the *App* class. The *MainPage* class also implements an app bar button and wires its *Click* event to navigate to the new *SettingsPage*. The only work left for the *OnNavigatedTo* override to do is to invoke the *SubscribeToNotifications* method on the *PushViewModel*.

```
public MainPage()
{
    InitializeComponent();
    BuildLocalizedApplicationBar();
    PhotoList.ItemsSource = App.ViewModel.Items;
    //notifications = new ObservableCollection<String>();
    //messageList.ItemsSource = Notifications;
    messageList.ItemsSource = App.Push.Notifications;
}

private void BuildLocalizedApplicationBar()
{
    ApplicationBar = new ApplicationBar();
    ApplicationBarIconButton appBarButton = new ApplicationBarIconButton(
        new Uri("/Assets/settings.png", UriKind.Relative));
    appBarButton.Text = AppResources.AppBarButtonText;
    appBarButton.Click += appBarButton_Click;
    ApplicationBar.Buttons.Add(appBarButton);
}
```

```
private void appBarButton_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(new Uri("/SettingsPage.xaml", UriKind.Relative));
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    App.Push.SubscribeToNotifications();
}
```

The *SettingsPage* itself is trivial; it uses the *ToggleSwitch* from the Microsoft Silverlight Toolkit so that the user can turn toast notifications on or off. The *ToggleSwitch* is data-bound to the *IsToastOk* property on the *PushViewModel*.

```
<StackPanel x:Name="LayoutRoot" Background="Transparent" Margin="{StaticResource
PhoneHorizontalMargin}">
    <toolkit:ToggleSwitch
        Header="Allow toast notifications" IsChecked="{Binding IsToastOk, Mode=TwoWay}"
        FontSize="{StaticResource PhoneFontSizeLarge}"/>
    <TextBlock
        Style="{StaticResource PhoneTextTitle3Style}" TextWrapping="Wrap"
        Text="When a toast notification is received, it consumes additional battery power, and
may be distracting." />
</StackPanel>

...
public SettingsPage()
{
    InitializeComponent();
    DataContext = App.Push;
}
```

For an even richer UX, you could provide two settings: one for toggling toasts on/off, and another or toggling all notifications on/off. This won't be useful in all apps, but if your app uses toasts plus tiles and/or raw notifications, and if it can function correctly with only some or none of the notification features, a finer-grained settings option might make sense.

Push Notification Security

The communications between MPNS and the Push Client component on the phone are secured via Secure Sockets Layer (SSL). This is set up by Microsoft. However, there is no default security for communications between the phone and your push web service, so you'd have to implement your own. The general recommendation is the same as for any web service that can exchange sensitive data: you should protect it with SSL. See Chapter 7 for more details on SSL.

Setting up SSL authentication for your web service is no different for a push web service than for any other web service. An additional benefit of securing your service is that this eliminates the daily

limit on the number of push notifications that you can send. Recall that, by default, unsecured web services are throttled at a rate of 500 notifications per app, per device, per day. To enable this feature, you must upload a TLS (SSL) certificate to the Store. The key-usage value of this certificate must be set to include client authentication; the Root Certificate Authority (CA) for the certificate must be one of the CAs that are trusted on the Windows Phone platform (these are listed in the documentation in the Store).

After you have submitted the certificate to the Store, you can associate any subsequently submitted apps with this certificate. This option is made available during the app submission process. There is obviously a gap between setting up your secured web service and submitting your certificate on the one hand, and having a submitted app approved and published in the Store on the other hand. To bridge this gap, Microsoft flags your web service as authenticated for a period of four months. Time constraint is removed when your app is successfully published to the Store.

To use the authenticated channel from your phone's client app, when creating the *HttpNotification Channel*, set the service name to the Common Name (CN) in the certificate. Keep in mind that you cannot use a self-signed certificate for this purpose.

If you do authenticate your push web service in this way, you can also take advantage of a call-back registration feature. This feature enables the MPNS to callback on your registered URI when it determines that it cannot deliver a message to the device as a result of the device being in an inactive state.

It's also worth keeping in mind that the push service does not guarantee message delivery, so you should not use it in scenarios for which failure to deliver one or more messages has serious consequences.

Summary

In this chapter, you saw how you can support the users personalization of her phone by enabling her to pin tiles to the Start screen. You can keep these fresh and relevant by using local and remote updates, and you can choose between three style templates (flip, iconic, and cycle) that each support the three tile sizes.

On top of that, you saw how the Windows Phone push notification system provides a way for you to build apps that can easily keep information on the phone up to date from remote servers. This involves a server-side application for generating notifications, a server-side web service for clients to register with, and a client-side app for receiving the incoming notifications and rendering the data appropriately on the phone. The three different notification types (raw, tile, and toast) each provide different data and behavior, appropriate for different use cases. Raw notifications are entirely under your control; tile and toast notifications integrate seamlessly with the standard phone experience.

Location and Maps

Location and maps are supported in Windows Phone by two sets of APIs: the location APIs and the map APIs. Each of these also comes in two flavors: version 7 and version 8. If you want to build an app that targets both version 7 and version 8 (or indeed both Windows Phone and desktop Windows or web Silverlight), you should use the version 7 APIs, because those work on both versions of the platform. On the other hand, if you want to target only version 8—taking advantage of its enhanced functionality—it is recommended that you use the version 8 APIs. This chapter examines all approaches, starting with the version 7 APIs to build a range of map and location-based apps, and then showing how you can build similar apps by using the version 8 APIs. The testing and performance tips at the end of the chapter apply equally to both versions.

Architecture

Under the hood (in both versions), location information is gathered from a variety of sources, including both on-device hardware and drivers and via web service calls to cloud-based location services. A key difference in Windows Phone 8 is that both maps and location are provided by the same underlying Nokia maps platform. As well as providing a cleaner component stack that makes it a lot easier for you to work with, this also provides significantly enhanced features and improved performance. Figure 16-1 presents the architecture of the two stacks.

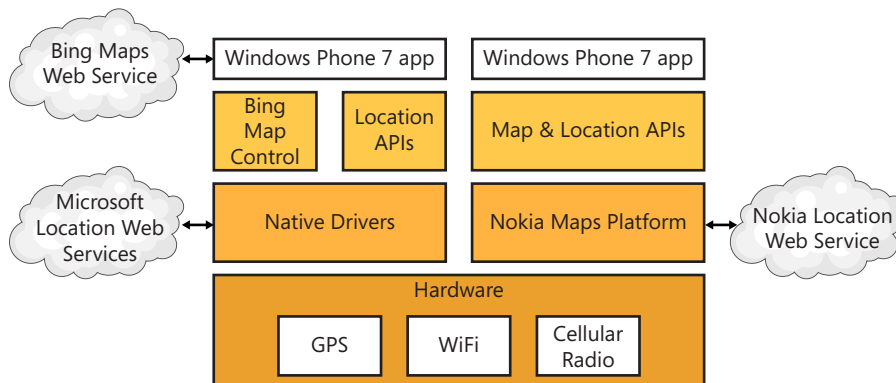


FIGURE 16-1 The support for location and maps uses two different stacks for Windows Phone version 7 and version 8.

The location data surfaced by the platform APIs can be sourced from any of the underlying sensors (GPS, Wi-Fi, cellular radio). With version 7, in which data is sourced via Wi-Fi, the on-device location service also uses the Microsoft location web service to help resolve positioning. The version 7 location APIs wrap the underlying native operating system (OS) feature that determines the best data source to feed to the API layer, depending on which source(s) are available and whether the app requires default or high accuracy. A version 7 app might also need to set up its own client to the Bing Maps web services, for example, for geocode information. In version 8, by contrast, the Nokia maps platform includes full geocoding, routing, and navigation capabilities, supplemented by a Nokia location web service where required. The version 8 APIs are Windows Runtime (WinRT) APIs, which is where the commonality with Windows 8 comes in. A version 8 app does not need to set up proxies to any web services, because all of this is handled at a lower level in the maps platform.

The most significant benefit of the Nokia maps platform is that it supports offline caching and preload of map data. On top of that, it also supports offline routing, which is unique among smartphones. All of this means that the phone will make far fewer calls to the server to retrieve data, and this in turn saves on network usage, which can significantly reduce the cost to users who have tariffed or pay-as-you-go data plans. The Nokia maps platform provides other benefits, too. For example, it knows about features in the landscape around the user, such as tunnels, and can predict where in the tunnel the user is, based on the speed he was going when he entered it. The Nokia maps platform also applies filtering to the raw sensor data when matching locations with maps so that it can track driving along a road and not register the location as being off the road, even when the sensors are struggling to provide accurate data.

Determining the Current Location (version 7)

Windows Phone 7 exposes a *GeoCoordinateWatcher* class, which provides a simple API for determining the user's current location. Figure 16-2 illustrates an app (the *SimpleGeoWatcher* solution in the sample code) that uses location data, reporting each location event as it occurs. You can see how this could easily form the basis of a "run-tracker" or "trail-tracking" app.

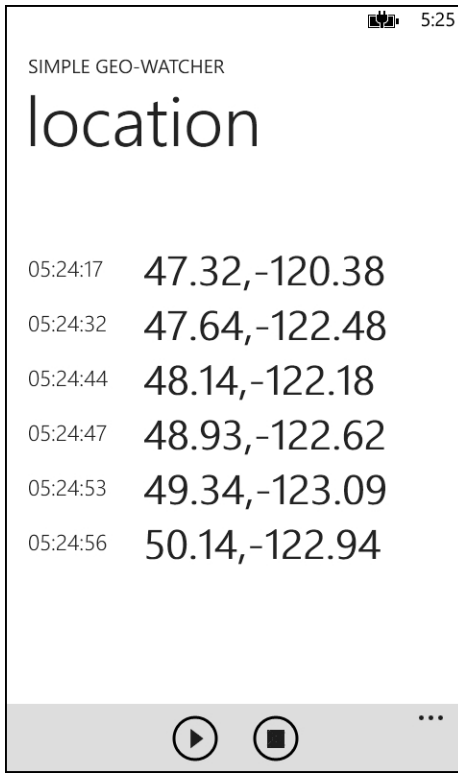


FIGURE 16-2 The *GeoCoordinateWatcher* class provides longitude and latitude readings.

In a version 7 project, you need to add a reference to the *System.Device.dll* (this is added automatically in a version 8 project). Then, to use location functionality, the app declares a *GeoCoordinateWatcher* object and instantiates it in the *OnNavigatedTo* override. The constructor is your only opportunity to set the required accuracy. You can subsequently retrieve this value from the *Desired Accuracy* property, which is read-only. You also declare a collection of *GeoPosition* objects, which will be stored each time you get a location event. This is enabled by hooking up the *PositionChanged* event.

```
private GeoCoordinateWatcher watcher;
public ObservableCollection<GeoPosition<GeoCoordinate>> Positions;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    InitializeCollection();

    if (watcher == null)
    {
        watcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High);
        watcher.PositionChanged += watcher_PositionChanged;
    }
}
```

In the *PositionChanged* event handler, extract the *Position* value and store it in the collection. The *Position* property is of type *GeoPosition<T>*, which includes both a *DateTimeOffset* and a position of type *<T>* (in this case, a *GeoCoordinate* value). The *GeoCoordinate* type represents a geographical location with latitude and longitude coordinates. Notice that the *PositionChanged* events come in on the user interface (UI) thread, so there's no need to use *Dispatcher.BeginInvoke* when updating the UI. This is a convenience to you, but it is inconsistent with the general pattern of event handlers and has caused confusion. You should normally assume that non-UI events will arrive at the app on a non-UI thread. As you will see later in this chapter, this inconsistency was removed in the Windows Phone 8 location APIs.



Note *GeoPosition<T>* will always in fact be *GeoPosition<GeoCoordinate>*. Given that, you might be wondering why the API doesn't simply use *GeoCoordinate* directly instead of wrapping it with the seemingly redundant *GeoPosition<T>* type. The *GeoPosition* type is designed to be generic in order to support potential future expansion. However, this is actually contrary to the internal API design guidelines; this is just an anomaly in the API surface that slipped through.

```
private void watcher_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    Positions.Add(e.Position);
}
```

The only other thing you need to do is to *Start* and *Stop* the *GeoCoordinateWatcher* at appropriate times. You only start it under user control, in the *Click* handler for the corresponding button. However, you stop it either when the user asks to stop or, as a good housekeeping technique, in the *OnNavigatedFrom* override. To start the watcher, you can, in theory, use either the *Start* method or the *TryStart* method. With *TryStart*, you can specify a timeout such that if the service doesn't start within that time, the method returns false. However, you should probably never use *TryStart* in this context, because this will block the UI thread, and blocking the UI thread is always considered bad practice. In fact, even the *Start* method can block the UI for up to several hundred milliseconds, but it will return relatively quickly, while the service is still starting up asynchronously. You should also consider listening to the *StatusChanged* event, which among other things, can tell you when the watcher has started.

```
private void startButton_Click(object sender, EventArgs e)
{
    watcher.Start();
}
```

```
private void stopButton_Click(object sender, EventArgs e)
{
    watcher.Stop();
}
```

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["Positions"] = Positions;
    watcher.Stop();
    watcher = null;
}
```



Note This example also persists the position changes in *ApplicationSettings*. This is reasonable if the number of positions is small, but it can rapidly become unreasonable as the number of positions increases. In a more realistic app, if you do want to persist location change data, you should consider doing so in isolated storage files, not in *ApplicationSettings*.

Before the location app will work, you need to ensure that your app manifest (*WMAppManifest.xml*) includes the *ID_CAP_LOCATION* in the *Capabilities* section.

```
<Capabilities>
...
<Capability Name="ID_CAP_LOCATION" />
</Capabilities>
```

In Windows Phone 7, if you don't add this capability, when you attempt to start a *GeoCoordinate Watcher*, you'll receive a status error. In Windows Phone 8, if you don't add this capability, the app will throw an "UnauthorizedAccessException" at the point where you try to create the *GeoCoordinate Watcher*. In Microsoft Visual Studio 2012, you can edit the manifest by using the convenient graphical manifest editor, as shown in Figure 16-3.

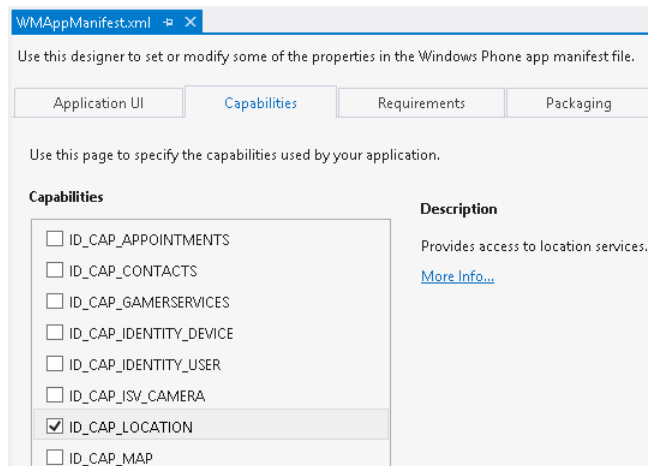


FIGURE 16-3 You can use the manifest editor in Visual Studio 2012 to configure your app's capabilities.

As well as location, the *GeoCoordinate* type provides additional values, including *Speed* and *Altitude*. Keep in mind that, in addition to the *PositionChanged* events raised on the *GeoCoordinate* *Watcher*, you should also handle the *StatusChanged* events. These events provide information about the location service, such as whether the user has disabled location on the device, and error conditions.



Note The location APIs also provide a *CivicAddressResolver* class. This class exposes a *Resolve* method that is intended to resolve a given *GeoCoordinate* to a civic address. However, the *Resolve* method is not implemented, so it will always return an empty address.

Bing Maps (version 7)

The Bing Maps service exposes a range of APIs for use in a wide variety of app types. These break down into three categories: the Bing *Map* control, the Bing Maps web services, and Bing-related Launchers, all of which are described in the following sections.

The Bing *Map* Control

To use Bing Maps and Bing services, you need a Microsoft Account (formerly known as a Windows Live ID) and a Bing Maps developer account, both of which are free and easy to obtain. Go to <http://binged.it/eGZfqT> for terms and conditions. To get started, go to <http://www.bingmapsportal.com>, associate an account with your Microsoft Account ID, and then select the option to create keys. A Bing Maps key is required for all apps that use the Bing Maps APIs: this is a 64-character string that identifies your app, which your app passes to the Bing Maps servers with each request. You can create multiple keys, and for each one, you supply an arbitrary app name and URL (these don't have to bear any relation to your real app name/URL, but you should use names that you will recognize as being associated with the real app).

For simplicity, you can paste the key into your app code, typically as a field in the *App* class. You can see an example of this in the *SimpleBingMaps* solution in the sample code. Be aware, however, that this is not a secure approach to use for a production app. For a more secure approach, it is common to store the key on a web server, not in the app, and have the app make a web service call on startup to retrieve the key.

```
internal const string BingMapsAccountId = "<< YOUR BING KEY >>";
```

You need to incorporate this key with a *CredentialsProvider*, and for this, you need to add a reference to the version 7 *Microsoft.Phone.Controls.Maps.dll* (not the *Microsoft.Phone.Maps.dll*, which is for version 8 projects), which you will find in the installation folder for the tools. This is typically in `%ProgramFiles(x86)%\Microsoft SDKs\Windows Phone\7.1\Libraries\Silverlight\`. You can then set up an *ApplicationIdCredentialsProvider* in your *MainPage* class, using this key. *ApplicationIdCredentialsProvider* implements *INotifyPropertyChanged*, so you can use it for data binding. This is useful for cases in which you want to download the key at runtime from your own web service.

```
private readonly CredentialsProvider _credentialsProvider =
    new ApplicationIdCredentialsProvider(App.BingMapsAccountId);
public CredentialsProvider CredentialsProvider
{
    get { return _credentialsProvider; }
}
```

Next, you need to put a *Map* control onto your page. In Visual Studio 2010, the Bing *Map* control is available by default in the toolbox. However, Visual Studio 2012 is geared toward the new map APIs, and for a version 8 project, the toolbox will show the new map control, not the Bing *Map* control. If you're creating a version 7 project in Visual Studio 2012, you won't see either the Bing *Map* control or the new *Map* control in the toolbox. Although you could add the Bing *Map* control to the toolbox, you will not be able to add the control to your page in the designer. To remedy this, you must edit your XAML manually. First, add a namespace reference for the *Microsoft.Phone.Controls.Maps.dll*, as shown here:

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

Next, declare an instance of the Bing *Map* control somewhere in your page, perhaps in the default *Grid* element. Data-bind the control to the *CredentialsProvider*, such as done in the following:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <maps:Map x:Name="map1" CredentialsProvider="{Binding CredentialsProvider}"/>
</Grid>
```

These few simple steps will give you map functionality in your app.

The *BingMapsLocation* solution in the sample code demonstrates how you can combine the *GeoCoordinateWatcher* with the Bing *Map* control. In the project, add a reference to the *Microsoft.Phone.Controls.Maps.dll* and set up a *CredentialsProvider*, as just described. As with the earlier location sample, if this is a version 7 project, you must also add a reference to *System.Device.dll*. In the XAML for your main page, add a namespace reference for the *Microsoft.Phone.Controls.Maps.dll* and then declare a map object on your page, as shown earlier.

In the *MainPage* class, declare a *GeoCoordinateWatcher* field for retrieving location sensor data. In *OnNavigatedTo*, you instantiate and start the watcher and hook up a handler for the *PositionChanged* events. You stop the watcher in *OnNavigatedFrom*.

```
private GeoCoordinateWatcher watcher;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (watcher == null)
    {
        watcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High);
        watcher.PositionChanged += watcher_PositionChanged;
    }
    watcher.Start();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
}
```



```

    if (watcher != null)
    {
        watcher.Stop();
        watcher = null;
    }
}

```

In the *PositionChanged* event handler, set the map position from the incoming *GeoCoordinate* parameter. At the same time, set the *ZoomLevel* to an arbitrary value of 16 (the valid range for the *ZoomLevel* property is 1.0 to 21.0), just to make it more obvious that the position locator is working as expected.

```

private void watcher_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    map1.Center = e.Position.Location;
    map1.ZoomLevel = 16;
}

```

One area in which the Bing maps provide better programmability than the new maps API is when you want to layer arbitrary graphics and text onto the surface of the map. The simplest implementation of this in Bing maps is the pushpin feature. Figure 16-4 shows a screenshot of the *BingPushpins* solution in the sample code, in which the user has tapped the map a few times. Each time, the app inserts a pushpin on top of the map and labels it with the selected coordinates.

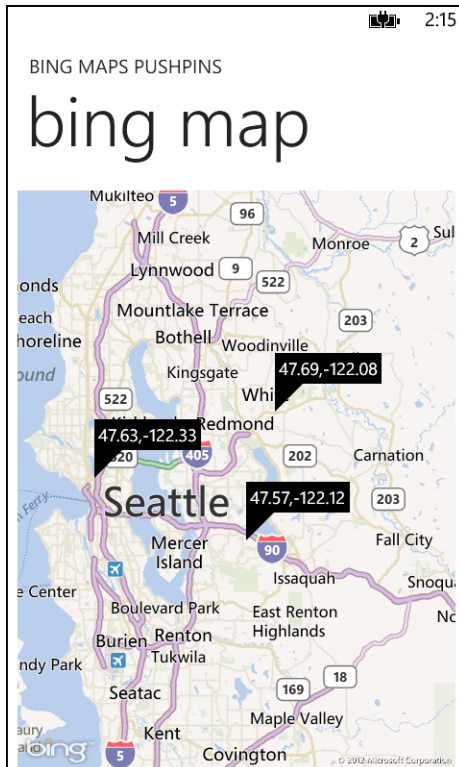


FIGURE 16-4 You can easily insert pushpins on top of the Bing *Map* control.

The code required to achieve this is very simple. The *MainPage* declares a *MapLayer* field; this layer will be used for all the pushpins. The *Tap* gesture for the *Map* control is wired up in XAML to an event handler. The handler first extracts the *Point* location of the tap gesture relative to the *Map* control and then invokes the *Map.ViewportPointToLocation* conversion method to convert this *Point* to a *GeoCoordinate*. The app uses this coordinate value for both the *Pushpin* location and its text label.

A map can have multiple layers or none at all. In this sample, even one layer is actually superfluous; you could simply add the *Pushpin* objects directly to the map, as follows:

```
private MapLayer layer;

private void map_Tap(object sender, System.Windows.Input.GestureEventArgs e)
{
    GeoCoordinate coord = map.ViewportPointToLocation(e.GetPosition(map));
    Pushpin pin = new Pushpin();
    pin.Location = coord;
    pin.Content = string.Format("{0:0.00},{1:0.00}", coord.Latitude, coord.Longitude);

    if (layer == null)
    {
        layer = new MapLayer();
        map.Children.Add(layer);
    }

    layer.Children.Add(pin);
}
```

Bing Maps Web Services

The Bing Maps Simple Object Access Protocol (SOAP) services are listed in Table 16-1.

TABLE 16-1 Bing Maps SOAP Services

Namespace	URL
GeocodeService	http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc?wsdl
SearchService	http://dev.virtualearth.net/webservices/v1/searchservice/searchservice.svc?wsdl
ImageryService	http://dev.virtualearth.net/webservices/v1/imageryservice/imageryservice.svc?wsdl
RouteService	http://dev.virtualearth.net/webservices/v1/routeservice/routeservice.svc?wsdl

The following example (the *TestGeocodeService* solution in the sample code), which is shown in Figure 16-5, uses the *GeocodeService* to find the geocode (latitude,longitude) for a given street address.

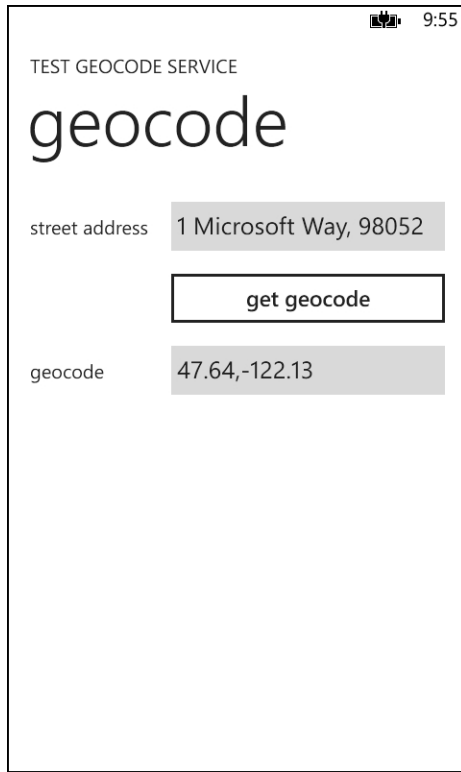


FIGURE 16-5 The *GeocodeService* is the most commonly used Bing Maps web service.

There are, in fact, two different ways that you can use the Bing Maps web services: using the traditional SOAP approach, or using the newer REST approach. Both approaches are described here. The sample code includes two corresponding variants: *TestGeocodeService_SOAP* and *TestGeocodeService_REST*. For more information on SOAP and REST, see Chapter 7, “Web and Cloud.”

For the SOAP approach, you need to generate a client-side proxy for the web service. To do this, add a service reference to the *GeocodeService* (see Figure 16-6). Click the Advanced button, set the Collection Type to System.Array (or IList), and then clear the Reuse Types In Referenced Assemblies check box; otherwise, the wizard generates a blank service reference. You don’t want the types from the assemblies referenced by the service because they’ll be full Microsoft .NET Common Language Runtime (CLR) and not compatible with the phone. The same applies to the collection classes.

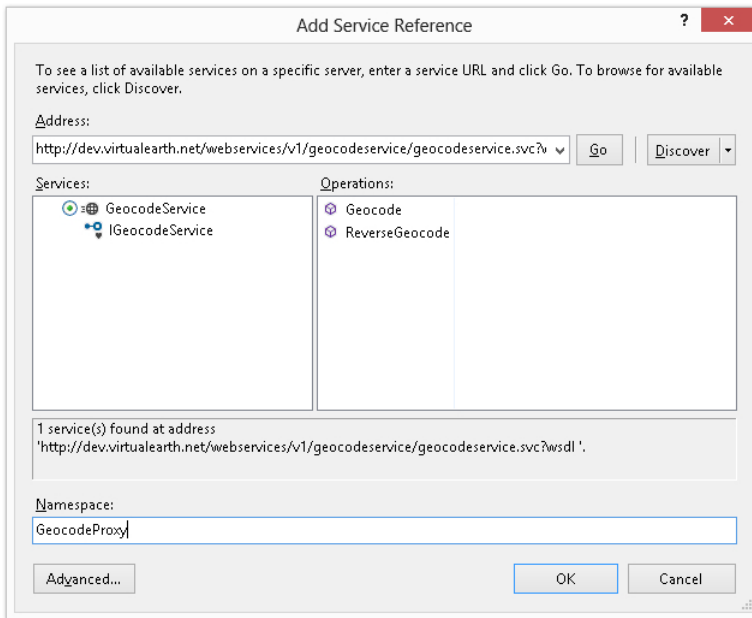


FIGURE 16-6 Use the Add Service Reference dialog box to generate proxy code for the Geocode service.

In addition to the client proxy code, this will also generate a client-side service config file, typically named *ServiceReferences.ClientConfig*. This will include declarations such as the basic HTTP binding, which you will need to reference later in your code.

```
<basicHttpBinding>
  <binding name="BasicHttpBinding_IGeocodeService" maxBufferSize="2147483647"
    maxReceivedMessageSize="2147483647">
    <security mode="None" />
  </binding>
</basicHttpBinding>
```

In the main page, you implement the button *Click* handler by creating a *GeocodeRequest*, handling its *GeocodeCompleted* event and invoking the asynchronous web service call *GeocodeAsync*.

```
private const string BingMapsAccountId = "<< YOUR BING KEY >>";

private void getGeocode_Click(object sender, RoutedEventArgs e)
{
    if (!String.IsNullOrEmpty(streetText.Text))
    {
        GeocodeRequest request = new GeocodeRequest();
        request.Credentials = new Credentials();
        request.Credentials.ApplicationId = BingMapsAccountId;
        request.Query = streetText.Text;
```

```

        GeocodeServiceClient geocodeService =
            new GeocodeServiceClient("BasicHttpBinding_IGeocodeService");
        geocodeService.GeocodeCompleted += geocodeService_GeocodeCompleted;
        geocodeService.GeocodeAsync(request);
    }
}

```

When you receive the *GeocodeCompleted* event, extract the *Latitude* and *Longitude* values from the results and then set them into the last *TextBox*.

```

private void geocodeService_GeocodeCompleted(
    object sender, GeocodeCompletedEventArgs e)
{
    GeocodeResponse response = e.Result;
    if (response.Results.Length > 0)
    {
        geocodeText.Text = String.Format("{0:F2},{1:F2}",
            response.Results[0].Locations[0].Latitude,
            response.Results[0].Locations[0].Longitude);
    }
    else
    {
        geocodeText.Text = "not found";
    }
}

```

For the alternative REST approach, you do not need to generate a client-side web service proxy at all. Instead, you can simply use the generic *WebClient* type to invoke the REST service and then parse the resulting XML (or JSON) data. Instead of the strongly-typed client-side proxy, you pass a simple query string. This version of the sample app sets up a template string with two placeholders: one for the user-supplied street address, and one for the developer Bing Maps account ID. Take note of the *o=xml* parameter in the query string. This specifies that the output (return data) should be in XML format. If you omit this parameter, the default output format is JSON. See Chapter 7 for more details on JSON.

The *Click* handler wires up the *DownloadStringCompleted* event to an inline delegate (you could alternatively use a traditional event handler method, if you prefer). This delegate parses the returned XML, extracts the *Latitude* and *Longitude* elements, and then composes these into a string to update the UI. The delegate will be invoked when the call to *DownloadStringAsync* returns.

```

private const string BingMapsAccountId = "<< YOUR BING KEY >>";
private const string geoCodeTemplate =
    @"http://dev.virtualearth.net/REST/v1/Locations/{0}?o=xml&key={1}";
private static XNamespace BingRestNs =
    @"http://schemas.microsoft.com/search/local/ws/rest/v1";

```

```

private void getGeocode_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += (o, r) =>
    {
        XDocument doc = XDocument.Parse(r.Result);
        double lat = double.Parse(
            (from entry in doc.Descendants(BingRestNs + "Latitude")
             select entry.Value).FirstOrDefault());
        double lon = double.Parse(
            (from entry in doc.Descendants(BingRestNs + "Longitude")
             select entry.Value).FirstOrDefault());
        Dispatcher.BeginInvoke(() => geocodeText.Text =
            String.Format("{0:0.00},{1:0.00}", lat, lon));
    };

    client.DownloadStringAsync(new Uri(String.Format(
        geoCodeTemplate, streetText.Text, BingMapsAccountId,
        UriKind.RelativeOrAbsolute)));
}

```

Bing Maps Launchers

Windows Phone 7 offers two Launchers that wrap the underlying Bing maps capabilities, as described in Table 16-2. These Launchers offer a simplified programming model; however, this simplification costs the loss of some of the power of the Bing Maps API. You do not need a Bing Maps development account to use the *BingMapsTask* and *BingMapsDirectionsTask*. You therefore also do not need to set up an *ApplicationIdCredentialsProvider*.

TABLE 16-2 Bing Maps Launchers

Task	Description
BingMapsDirectionsTask	Launches the Bing Maps app, specifying a starting and/or ending location, for which driving or walking directions are displayed.
BingMapsTask	Launches the Bing Maps app centered at the specified or current location.

Figure 16-7 demonstrates the new *BingMapsDirectionsTask*. You can see this at work in the *TestBingTask* solution in the sample code. The app offers one *Button* control; the *Click* handler simply creates a *BingMapsDirectionsTask* and sets the destination location.

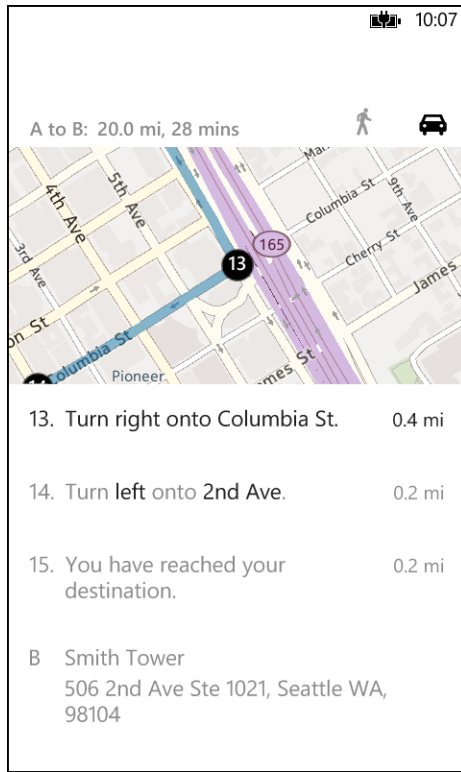


FIGURE 16-7 Using the BingMapsDirectionsTask Launcher.

When you instantiate the *BingMapsDirectionsTask*, you can specify a label and/or *GeoCoordinates* for the target location (the end point). If you supply a location, the label is used as a descriptive label. If you don't supply a location, the label is used as a search string. You can optionally also specify a location start point, but if you don't, the current location is used as the default starting point. Finally, invoke the *Show* method to execute the launcher.

```
private void getDirections_Click(object sender, RoutedEventArgs e)
{
    BingMapsDirectionsTask bingTask = new BingMapsDirectionsTask();
    bingTask.End = new LabeledMapLocation("Smith Tower, Seattle", null);
    bingTask.Show();
}
```

Getting Location (version 8)

The new Windows Phone 8 maps APIs and the new location APIs form a more seamless whole. They are replacements for the old Bing Maps APIs and the old version 7 location APIs. To use the new APIs, you do not need a Bing account and you do not use the Bing web services. The new APIs are more powerful than the old ones in some cases. They use the underlying Nokia maps platform layer, which

is faster and more accurate. It is also easier to develop against the new APIs. You should use the new location APIs unless you're explicitly targeting both Windows Phone 7 and Windows Phone 8. The *Geolocator* class is the Windows Phone 8 equivalent of the *GeoCoordinateWatcher* class that was used in version 7 projects. There are two ways with which you can use *Geolocator*:

- In "ad hoc" mode, wherein you invoke the *GetGeopositionAsync* method on demand
- In "continuous" mode, wherein you handle the *PositionChanged* events.

The *SimpleGeoLocator* app in the sample code shows how to use the "ad hoc" approach, as shown in Figure 16-8.

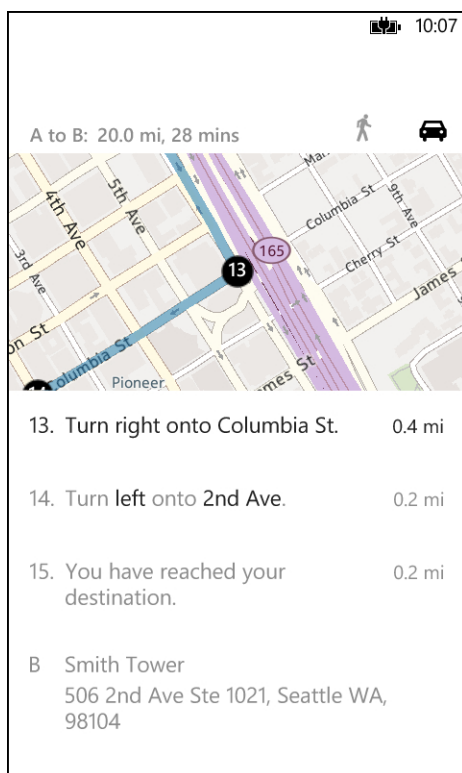


FIGURE 16-8 You can use the *Geolocator* class in ad hoc mode.

All the work is done in the *Click* handler for a custom App Bar button. This instantiates a *Geolocator* object, and sets the *DesiredAccuracy* property. You could alternatively set the *DesiredAccuracyInMeters* property if you want more precise control over this. Next, the app fetches the current position by using the *GetGeopositionAsync* method. As its name implies, this is an asynchronous method, so the code waits on the return from this by using the *await* keyword. Keep in mind that you can only invoke asynchronous methods in this way if the calling method itself is declared as *async*. Using the *await* keyword in this way also has the effect that the results are returned on the calling thread. In this example, this is the UI thread, so no additional thread marshaling is required here.

When the location data is retrieved, the app sets the various location properties into UI elements. Unlike the *GeoCoordinateWatcher*, the *Geolocator* always returns location data on a non-UI thread. For this reason, you must always use *Dispatcher.BeginInvoke* to marshal the UI property setters back to the UI thread.

It is also important to handle exceptions when using the *Geolocator*. In particular, you must handle the case in which the user has disabled location services on his phone; in this case, an *UnauthorizedAccessException* is thrown. Finally, don't forget that the *ID_CAP_LOCATION* capability must be added to the manifest.



Note There is a quirk in the location simulator (which is part of the Windows Phone emulator). If you accept the default value for *DesiredAccuracy* (or if you set it to *PositionAccuracy.Default* instead of *PositionAccuracy.High*), when you test the ad hoc *Geolocator* approach, the *GetGeopositionAsync* call fetches the last location set in the simulator before you start, and does not fetch any updated location. However, for real use on a real device, *PositionAccuracy.Default* is mostly what you want; it provides a good level of accuracy while still keeping battery consumption down. If you do set *PositionAccuracy.High* for the purposes of testing in the emulator, you should remember to reset it to *PositionAccuracy.Default* before shipping your app.

GetGeopositionAsync has an overload that takes two parameters: the *maximumAge* of the data to retrieve, and a *timeout* value. When you invoke this method, it gathers old data up to the point where the *maximumAge* is exceeded; Thus, if you want to test rapid changes in location, you should set *maximumAge* to a low value (in seconds). This sample creates a fresh *Geolocator* each time the user taps the refresh button. You could instead set up a *Geolocator* as a class field, initialized once, and reuse it for each user request.

```
private async void refreshButton_Click(object sender, EventArgs e)
{
    try
    {
        Geolocator locator = new Geolocator();
        locator.DesiredAccuracy = PositionAccuracy.High;
        Geoposition position = await locator.GetGeopositionAsync();

        timestamp.Text = position.Coordinate.Timestamp.ToString("hh:mm:ss");
        latitude.Text = position.Coordinate.Latitude.ToString("0.00");
        longitude.Text = position.Coordinate.Longitude.ToString("0.00");
        speed.Text = String.Format("{0:0.00}", position.Coordinate.Speed);
        altitude.Text = String.Format("{0:0.00}", position.Coordinate.Altitude);
    }
    catch (UnauthorizedAccessException)
    {
        MessageBox.Show("location capability is disabled");
    }
}
```

```

    catch (Exception)
    {
        MessageBox.Show("unknown error");
    }
}

```

The ad hoc approach is often all you need (wherein you fetch location data only when it is actually required in the app). In other scenarios, you want the app to be fed location data continuously over a period of time. The classic example is a “run-tracker” app. In this scenario, you handle the *Position Changed* events. Unlike the *GeoCoordinateWatcher*, the *Geolocator* class does not expose *Start* and *Stop* methods; instead, the location events are raised as soon as you hook up the *PositionChanged* event handler, and stop being raised when you unhook the handler. The *ContinuousLocation* solution in the sample code demonstrates this approach, which is presented in Figure 16-9.

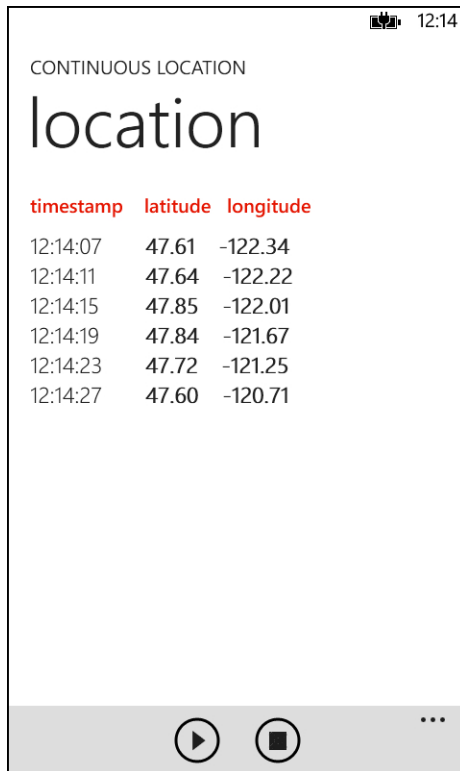


FIGURE 16-9 You can handle *PositionChanged* events on the *Geolocator* class.

All of the interesting code is in the *MainPage* class. This sets up a *Geolocator* field and a collection of *Geocoordinate* values. The *Positions* collection is set as the *ItemsSource* for a *ListBox* in the UI. When the user navigates to the page, the app creates a *Geolocator* object and sets the *DesiredAccuracy* and *MovementThreshold* properties. Note that you must set either the *MovementThreshold* property (in meters) or the *ReportInterval* property (in seconds) before hooking up the *PositionChanged* event,

or an exception will be thrown. It is also a good idea to unhook the event handler and clean up the object in the *OnNavigatedFrom* override. The *OnNavigatedTo* also enables the start button and disables the stop button.

```
private Geolocator locator;
public ObservableCollection<Geocoordinate> Positions;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (locator == null)
    {
        locator = new Geolocator();
        locator.DesiredAccuracy = PositionAccuracy.High;
        locator.MovementThreshold = 50;
        //locator.ReportInterval = 1;
    }
    stopButton.IsEnabled = false;
    startButton.IsEnabled = true;
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    locator.PositionChanged -= locator_PositionChanged;
    locator = null;
}
```

The user can tap the start button to begin receiving *PositionChanged* events and then tap the stop button to discontinue receiving the events. The *Click* event handlers for these buttons also enable or disable the buttons as appropriate. This ensures that the user can't, for example, tap the start button twice and end up with two *PositionChanged* event handlers. When the last event handler is unregistered, the events are no longer raised. When an event is received, the app extracts the *Geoposition* data and adds it to the *Positions* collection. This is a dynamic data-bound collection.

```
private void startButton_Click(object sender, EventArgs e)
{
    if (locator != null)
    {
        locator.PositionChanged += locator_PositionChanged;
        startButton.IsEnabled = false;
        stopButton.IsEnabled = true;
    }
}

private void stopButton_Click(object sender, EventArgs e)
{
    if (locator != null)
    {
        locator.PositionChanged -= locator_PositionChanged;
        stopButton.IsEnabled = false;
        startButton.IsEnabled = true;
    }
}
```

```
private void locator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    Dispatcher.BeginInvoke(() => { Positions.Add(args.Position.Coordinate); });
}
```

The *MainPage* XAML declares a *ListBox* with an *ItemTemplate* set to include three *TextBlock* controls, suitably data-bound and formatted.

```
<TextBlock
    Grid.Column="0" Text="{Binding Timestamp, StringFormat='hh:mm:ss'}"
    Style="{StaticResource PhoneTextTitle3Style}"/>
<TextBlock
    Grid.Column="1" Text="{Binding Latitude, StringFormat='0.00'}"
    Style="{StaticResource PhoneTextTitle3Style}" FontWeight="Bold"/>
<TextBlock
    Grid.Column="2" Text="{Binding Longitude, StringFormat='0.00'}"
    Style="{StaticResource PhoneTextTitle3Style}" FontWeight="Bold"/>
```

Maps API (version 8)

Windows Phone 8 introduces a new set of maps APIs that can be used in version 8 projects in place of the old Bing Maps APIs. This includes a new *Map* control, support classes such as the *MapRoute* and *MapLayer*, and Launchers such as the *MapsTask* and *MapsDirectionsTask*. The new map system in version 8 is faster than the old system; consumes less network bandwidth; is more accurate; has more intelligent error-correction and noise-filtering; and, in most scenarios, is also easier to develop against. However, some things are easier with the “old” Bing maps API, specifically, adding items such as pushpins to the surface of a map.

The Map Control

In a Windows Phone 8 project, you can drag a *Map* control from the toolbox onto the XAML design surface in your app. This will generate the XML namespace for the *Map* control (*Microsoft.Phone.Maps.Controls*). You must add the corresponding *ID_CAP_MAP* capability to your app manifest manually. The *MapDemo* solution in the sample code illustrates some of the features of the *Map* control by giving the user the ability to set the control properties, as shown in Figure 16-10.

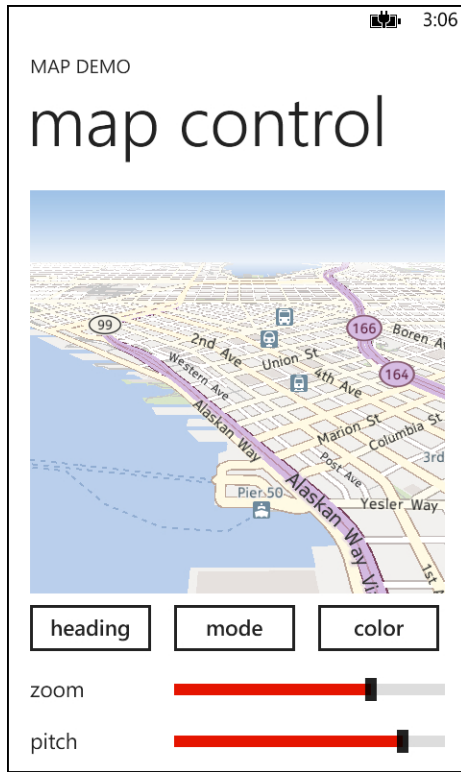


FIGURE 16-10 The new *Map* control exposes several programmable properties.

In this example, the *Map* control is initialized in the *OnNavigatedTo* override. It is also manually synchronized with the two *Slider* controls. Each public property of the *Map* control, including *ZoomLevel* and *Pitch*, is also exposed as a *DependencyProperty*, so the app could just as easily data-bind these. The user can move the *Slider* controls to adjust the *ZoomLevel* and *Pitch* properties on the map. *ZoomLevel* is self-explanatory. *Pitch* affects the user's perspective view of the map. A *Pitch* value of 0 (zero) results in a flat view (the valid range is 0 to 75). Increasing the *Pitch* has the effect of rotating the map about the X axis so that the top of the map appears further away, and the bottom appears closer.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    map.Center = new System.Device.Location.GeoCoordinate(47.6066, -122.3386);
    map.ZoomLevel = zoomSlider.Value = 15;
    map.Pitch = pitchSlider.Value = 0;
}
private void zoomSlider_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)
{
    if (map != null)
    {
        map.ZoomLevel = ((Slider)sender).Value;
    }
}
```

```
private void pitchSlider_ValueChanged(object sender, RoutedEventArgs<double> e)
{
    if (map != null)
    {
        {
            map.Pitch = ((Slider)sender).Value;
        }
    }
}
```

In a similar manner, the three *Button* controls provide the means for the user to modify the *Heading*, *MapCartographicMode* and *ColorMode* of the map. The *Heading* property governs the orientation of the map. With a *Heading* value of 0 or 360, north is at the top of the page. As the user increments the *Heading* property, the map orientation rotates in a counter-clockwise direction about the Y axis. So, with a *Heading* value of 90, the top of the page is due east. A value of 180 would put the top of the page at due south. Due west at the top would require a value of 270. There are four discrete *MapCartographicMode* values, which are illustrated in Figure 16-11. Similarly, there are two *MapColorMode* properties.

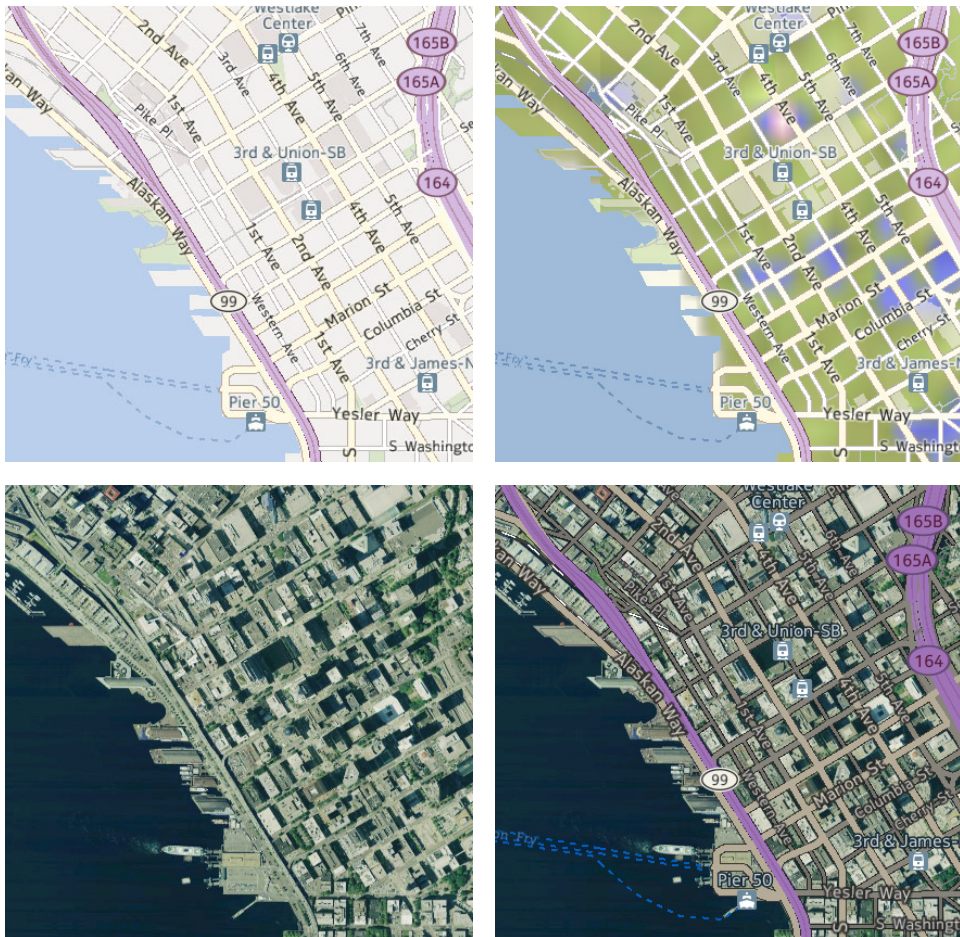


FIGURE 16-11 There are four *MapCartographicMode* values: *Road*, *Terrain*, *Aerial*, and *Hybrid*, as viewed from left to right.

```

private void headingButton_Click(object sender, RoutedEventArgs e)
{
    map.Heading = (map.Heading + 10) % 360;
}

private void mapModeButton_Click(object sender, RoutedEventArgs e)
{
    switch (map.CartographicMode)
    {
        case MapCartographicMode.Aerial:
            map.CartographicMode = MapCartographicMode.Hybrid; break;
        case MapCartographicMode.Hybrid:
            map.CartographicMode = MapCartographicMode.Road; break;
        case MapCartographicMode.Road:
            map.CartographicMode = MapCartographicMode.Terrain; break;
        case MapCartographicMode.Terrain:
            map.CartographicMode = MapCartographicMode.Aerial; break;
    }
}

private void colorModeButton_Click(object sender, RoutedEventArgs e)
{
    switch (map.ColorMode)
    {
        case MapColorMode.Dark:
            map.ColorMode = MapColorMode.Light; break;
        case MapColorMode.Light:
            map.ColorMode = MapColorMode.Dark; break;
    }
}

```

An alternative to setting individual properties on the *Map* control is to call the *SetView* method. This method has 10 overloads, and you can change multiple properties at once. Figure 16-12 shows the *MapDemo_SetView* solution in the sample code, which demonstrates this technique.

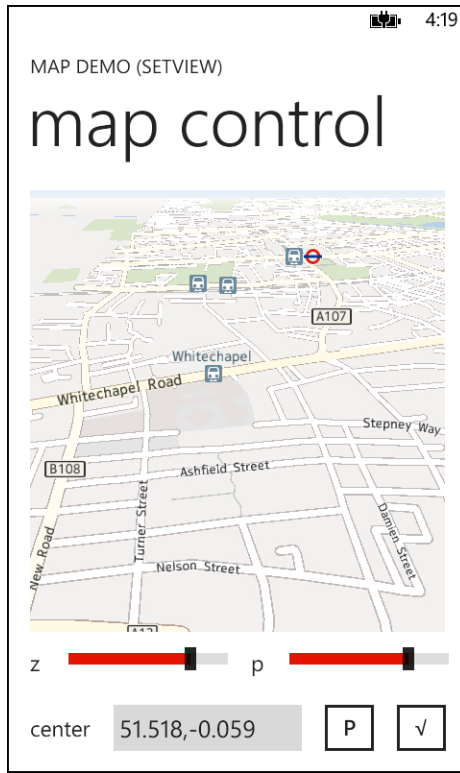


FIGURE 16-12 Using the *SetView* method, you can change multiple map properties at once.

The two *Slider* controls govern the *ZoomLevel* and *Pitch* as before, but there are no *ValueChanged* handlers for the controls. Instead, the *Click* handler for the *updateButton* picks up the current value of each *Slider* and passes it to the *SetView* method. The *updateButton* is labelled with a tick mark (✓). The *animationButton* changes its label from "P" to "L" to "N", and is backed by a *Click* handler that cycles through the three possible *MapAnimationKind* values; the *updateButton* uses the cached *animationKind* field when it invokes *SetView*, along with a *GeoCoordinate* value for the new map *Center* value. The *MapAnimationKind* governs the visual animation that is used to transition to the new view. From the user's perspective, the effect is as follows:

- **MapAnimationKind.None** The old view snaps to the new view, with no visual animation.
- **MapAnimationKind.Linear** The user moves in a linear path from the old view to the new view, with no change in perspective.
- **MapAnimationKind.Parabolic** The user's view zooms out from the old view to a mid-point and then zooms in to the new view, describing a parabolic "flight path."


```

private MapAnimationKind animationKind = MapAnimationKind.None;

private void animationButton_Click(object sender, RoutedEventArgs e)
{
    switch (animationKind)
    {
        case MapAnimationKind.None:
            animationKind = MapAnimationKind.Linear;
            animationButton.Content = "L"; break;
        case MapAnimationKind.Linear:
            animationKind = MapAnimationKind.Parabolic;
            animationButton.Content = "P"; break;
        case MapAnimationKind.Parabolic:
            animationKind = MapAnimationKind.None;
            animationButton.Content = "N"; break;
    }
}

private void updateButton_Click(object sender, RoutedEventArgs e)
{
    string[] coordinates = centerText.Text.Split(',');
    GeoCoordinate geoCoord = new GeoCoordinate(double.Parse(coordinates[0]), double.
Parse(coordinates[1]));
    map.SetView(geoCoord, zoomSlider.Value, 0, pitchSlider.Value, animationKind);
}

```

Some of the properties of the *Map* control cannot be set in the *SetView* overloads. These include the *LandmarksEnabled* and *PedestrianFeaturesEnabled* flags. These flags govern whether recognized landmarks (such as prominent buildings and other structures) and pedestrian features (such as walkways, stairs, and pedestrian-only plazas) are shown on the map. A related feature is the map overlay capability, whereby you can add layers on top of the map. You can put any arbitrary UI elements in a layer and then add that layer to the collection of layers that the *Map* control maintains. You first saw this technique in the *BingPushpins* sample app. The new map APIs support the same conceptual behavior of layering on text and graphics onto the map. However, the new APIs do not support *Pushpin* controls; thus, they can be slightly more cumbersome to work with in this context. The layer and overlay APIs are demonstrated in the *MapLayerDemo* app in the sample code, as shown in Figure 16-13.

In this app, there are two App Bar buttons: one for toggling *LandmarksEnabled*, and the other for toggling *PedestrianFeaturesEnabled*. The code for the *Click* handlers is trivial.

```

private void landmarksButton_Click(object sender, EventArgs e)
{
    map.LandmarksEnabled = !map.LandmarksEnabled;
}

private void pedestrianButton_Click(object sender, EventArgs e)
{
    map.PedestrianFeaturesEnabled = !map.PedestrianFeaturesEnabled;
}

```

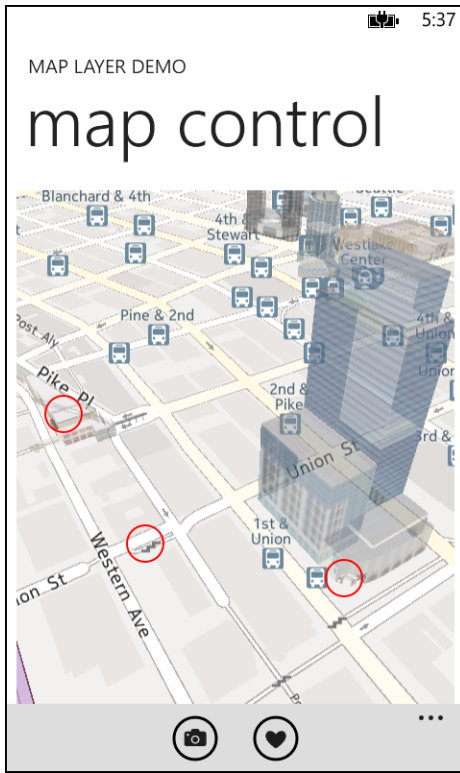


FIGURE 16-13 You can overlay multiple layers onto a map.

The overlay code is a little more complicated. The idea here is that when the user taps the map, the app will create a red circle and overlay this on the map at the point where the user tapped. A *Tap* event handler is wired up in the XAML for the map. This handler creates an *Ellipse*, and adds this to a new *MapOverlay* object. To set the correct position, you first extract the position of the *Tap* event relative to the *Map* control. Next, you offset it by half the width/height of the *Ellipse* so that the point is centered around the center point of the *Ellipse* itself. Then, invoke the *ConvertViewpointPointToGeoCoordinate* method on the *Map* control and set the resulting *GeoCoordinate* into the *MapOverlay* object. Finally, add the *MapOverlay* to a *MapLayer* object and then add that object to the *Layers* collection on the *Map* control. This example only needs one *MapLayer*, so this is set up as a class field, initialized on first use. In other scenarios, you might want more than one *MapLayer*.

```
private MapLayer layer;

private void map_Tap(object sender, GestureEventArgs e)
{
    Ellipse ellipse = new Ellipse();
    ellipse.Width = 40;
    ellipse.Height = 40;
    ellipse.Stroke = new SolidColorBrush(Colors.Red);
    ellipse.StrokeThickness = 2;
```

```

MapOverlay overlay = new MapOverlay();
overlay.Content = ellipse;
Point point = e.GetPosition(map);
point.X -= 20;
point.Y -= 20;
GeoCoordinate coordinate = map.ConvertViewportPointToGeoCoordinate(point);
overlay.GeoCoordinate = coordinate;

if (layer == null)
{
    layer = new MapLayer();
    map.Layers.Add(layer);
}
layer.Add(overlay);
}

```

Route and Directions

If you want to provide route information and directions in your app, you have two choices. The *MapsDirectionsTask* is a simple, self-contained way to get directions, whereas the *GeocodeQuery*, *RouteQuery*, and *MapRoute* APIs offer more flexibility and power, at the cost of greater developer complexity. The *MapDirections* app in the sample code illustrates this more flexible approach, as depicted in Figure 16-14. Note that this app needs both *ID_CAP_MAP* and *ID_CAP_LOCATION*.

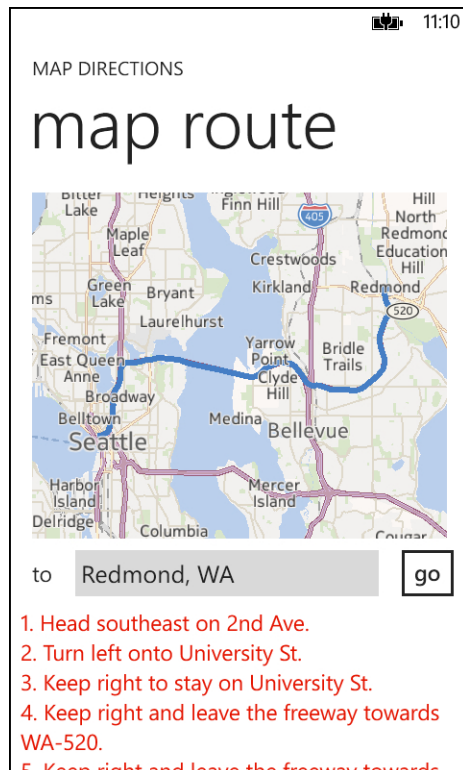


FIGURE 16-14 You can use the *RouteQuery* and *MapRoute* APIs to get route directions.

The XAML for the *MainPage* defines a *Map* control, a *TextBox* for the user to enter a destination place, and a *Button* to begin fetching the route information. Below that is a *ListBox* that will be data-bound to the route information when it is returned. The code for the page defines fields for a collection of *GeoCoordinate* values, which is used for the coordinates of the start and finish locations. The page also defines a *GeocodeQuery* for getting the coordinates for a textual finish location, and a *RouteQuery* for getting the route information. When the user taps the *getRoute* button, the *Click* handler first gets the current location by using the *GetGeopositionAsync* method of the *Geolocator* class. Next, the user's requested finish location is fed into a *GeocodeQuery* to get the corresponding coordinates. This is done asynchronously; it raises a *QueryCompleted* event when finished.

```
private List<GeoCoordinate> coordinates = new List<GeoCoordinate>();
private GeocodeQuery codeQuery;
private RouteQuery routeQuery;

private async void getRoute_Click(object sender, RoutedEventArgs e)
{
    try
    {
        Geolocator locator = new Geolocator();
        locator.DesiredAccuracy = PositionAccuracy.High;
        Geoposition position = await locator.GetGeopositionAsync();

        GeoCoordinate coordinate = new GeoCoordinate(
            position.Coordinate.Latitude, position.Coordinate.Longitude);
        coordinates.Add(coordinate);

        codeQuery = new GeocodeQuery();
        codeQuery.SearchTerm = targetText.Text;
        codeQuery.GeoCoordinate = coordinate;
        codeQuery.QueryCompleted += codeQuery_QueryCompleted;
        codeQuery.QueryAsync();
    }
    catch (UnauthorizedAccessException)
    {
        MessageBox.Show("location is disabled");
    }
}
```

The handler for the *QueryCompleted* event for the *GeocodeQuery* adds both the start and finish coordinates to a *RouteQuery*, hooks up the *QueryCompleted* event for that query, and then invokes the query.

```
private void codeQuery_QueryCompleted(
    object sender, QueryCompletedEventArgs<IList<MapLocation>> e)
{
    if (e.Error != null)
    {
        return;
    }
}
```

```

routeQuery = new RouteQuery();
coordinates.Add(e.Result[0].GeoCoordinate);
routeQuery.Waypoints = coordinates;
routeQuery.QueryCompleted += routeQuery_QueryCompleted;
routeQuery.QueryAsync();
}

```

The handler for the *QueryCompleted* event on the *RouteQuery* first extracts the *Route* object from the event arguments, converts it to a *MapRoute* object, and then adds it to the *Map* control. This results in the route being drawn on the map. Next, the handler iterates the collection of route instructions and adds each one to a list, which is then data-bound to the *ListBox* at the bottom of the page.

```

private void routeQuery_QueryCompleted(
    object sender, QueryCompletedEventArgs<Route> e)
{
    if (e.Error != null)
    {
        return;
    }

    Route route = e.Result;
    MapRoute mapRoute = new MapRoute(route);
    map.AddRoute(mapRoute);

    List<string> list = new List<string>();
    foreach (RouteLeg leg in route.Legs)
    {
        for (int i = 0; i < leg.Maneuvers.Count; i++)
        {
            RouteManeuver maneuver = leg.Maneuvers[i];
            list.Add(String.Format("{0}. {1}", i+1, maneuver.InstructionText));
        }
    }
    routeList.ItemsSource = list;
}

```

This approach is useful if you need to do further computations on the route information or if you're using it for other purposes than display. If all you want is to provide a simple set of directions, it is easier to use the *MapsDirectionsTask*. This also provides a suitable UI for you, as described in the Maps Launchers section that follows.



Note Some of the types in the new maps and location APIs are defined in the *System.Device.Location* namespace, whereas some are defined in *Windows.Devices.Geolocation*. This can be a little confusing, especially, for instance, the *GeoCoordinate* class (defined in *System.Device.Location*) and the (similar, but different) *Geocoordinate* class (defined in *Windows.Devices.Geolocation*). In Windows Phone location-based apps, it is common to use a combination of the *System.Device.Location.GeoCoordinate* class along with the *Windows.Devices.Geolocation.Geolocator* class. In practice, you're unlikely to use the *Windows.Devices.Geolocation* type, unless you're cross-targeting with Windows 8.

As an aside, if you want to streamline your code a little, you could write simple wrappers for the asynchronous *GeocodeQuery* and *RouteQuery* calls instead of setting up traditional callbacks. The following code shows how you can achieve this, via extension methods:

```
public static class GeocodeQueryExtensions
{
    public static Task<IList<MapLocation>> GeocodeAsync(this GeocodeQuery query)
    {
        TaskCompletionSource<IList<MapLocation>> source =
            new TaskCompletionSource<IList<MapLocation>>();
        query.QueryCompleted += (sender, e) =>
        {
            if (e.Error != null)
            {
                source.TrySetException(e.Error);
            }
            else
            {
                source.SetResult(e.Result);
            }
        };
        query.QueryAsync();
        return source.Task;
    }
}

public static class RouteQueryExtensions
{
    public static Task<Route> RouteAsync(this RouteQuery query)
    {
        TaskCompletionSource<Route> source =
            new TaskCompletionSource<Route>();
        query.QueryCompleted += (sender, e) =>
        {
            if (e.Error != null)
            {
                source.TrySetException(e.Error);
            }
            else
            {
                source.SetResult(e.Result);
            }
        };
        query.QueryAsync();
        return source.Task;
    }
}
```

You could then invoke these extension methods and await their return, as demonstrated in the code that follows. This eliminates the need for the *QueryCompleted* event handlers for both queries.

```
private async void getRoute_Click(object sender, RoutedEventArgs e)
{
    ... unchanged code omitted for brevity
    codeQuery = new GeocodeQuery();
```

```

codeQuery.SearchTerm = targetText.Text;
codeQuery.GeoCoordinate = coordinate;

//codeQuery.QueryCompleted += codeQuery_QueryCompleted;
//codeQuery.QueryAsync();
IList<MapLocation> loc = await codeQuery.GeocodeAsync();
coordinates.Add(loc[0].GeoCoordinate);
routeQuery = new RouteQuery();
routeQuery.Waypoints = coordinates;

//routeQuery.QueryCompleted += routeQuery_QueryCompleted;
//routeQuery.QueryAsync();
Route route = await routeQuery.RouteAsync();
MapRoute mapRoute = new MapRoute(route);
map.AddRoute(mapRoute);

List<string> list = new List<string>();
foreach (RouteLeg leg in route.Legs)
{
    for (int i = 0; i < leg.Maneuvers.Count; i++)
    {
        RouteManeuver maneuver = leg.Maneuvers[i];
        list.Add(String.Format("{0}. {1}", i + 1, maneuver.InstructionText));
    }
}
routeList.ItemsSource = list;
}

```

Maps Launchers

The new maps APIs in Windows Phone 8 include four Launchers, which are described in Table 16-3. The *MapsTask* and *MapsDirectionsTask* are the modern equivalent of the “old” *BingMapsTask* and *BingMapsDirectionsTask*. In fact, under the hood, these two sets of APIs are the same, they just have two different names.

TABLE 16-3 Windows Phone 8 Maps Launchers

Task	Description
<i>MapsTask</i>	Searches around the current or specified location for items that match the specified search text. This is a synonym for the <i>BingMapsTask</i> .
<i>MapsDirectionsTask</i>	Provides a map route and directions list, given the destination location (which can be in coordinate or textual form), and optionally a starting location. This is a synonym for the <i>BingMapsDirectionsTask</i> .
<i>MapDownloaderTask</i>	Takes the user through a series of narrowing choices to download selected maps to the phone so that later map-based apps can use these maps for faster rendering.
<i>MapUpdaterTask</i>	Takes the user’s current set of locally-downloaded maps and then fetches any updates that might be available for them.

The *TestMapsTasks* solution in the sample code illustrates the use of all four of these Launchers (invoked by four buttons), as shown in Figure 16-15.

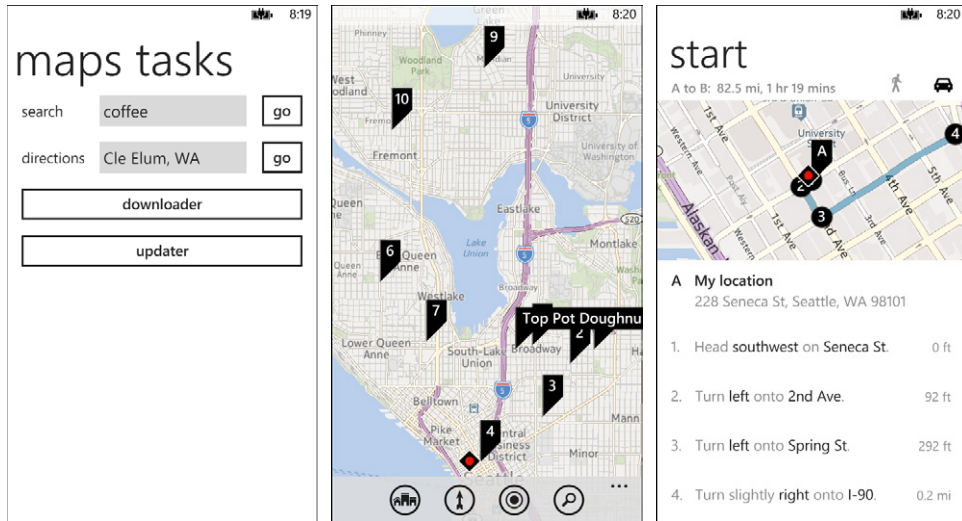


FIGURE 16-15 The *TestMapsTasks* solution (on the left), the *MapsTask* Launcher (center), and the *MapsDirectionsTask* Launcher (right).

The code is very simple; all the interesting work is in the *Click* event handlers for the four *Button* controls. The *searchButton* invokes the *MapsTask*, the *directionsButton* invokes the *MapsDirectionsTask*, the *downloadButton* invokes the *MapDownloaderTask*, and the *updateButton* invokes the *MapUpdaterTask*.

```
private void searchButton_Click(object sender, RoutedEventArgs e)
{
    MapsTask mapsTask = new MapsTask();
    mapsTask.SearchTerm = searchText.Text;
    mapsTask.ZoomLevel = 15;
    mapsTask.Show();
}

private void directionsButton_Click(object sender, RoutedEventArgs e)
{
    MapsDirectionsTask directionsTask = new MapsDirectionsTask();
    directionsTask.End = new LabeledMapLocation(directionsText.Text, null);
    directionsTask.Show();
}

private void downloadButton_Click(object sender, RoutedEventArgs e)
{
    MapDownloaderTask downloaderTask = new MapDownloaderTask();
    downloaderTask.Show();
}

private void updateButton_Click(object sender, RoutedEventArgs e)
{
    MapUpdaterTask updaterTask = new MapUpdaterTask();
    updaterTask.Show();
}
```


Continuous Background Execution (version 8)

The memory and processor constraints on a mobile device mean that there is only ever one app running in the foreground, with access to the full screen. As described in Chapter 2, “App Model and Navigation,” when the user navigates away from an app, that app is suspended and might also be tombstoned. Then, there is a set of background-specific task types that are designed to run only in the background, as discussed in Chapter 8, “Background Agents.” Between the “fully-foreground” app and the “fully-background” agent lies the Continuous Background Execution (CBE) app, which can run either in the foreground or in the background.

The canonical use case for this app type is the “run-tracker.” The idea is that the app starts in the foreground, but then when the user navigates forward away from the app, it continues to execute in the background. Of course, if the user navigates backward out of the app, it is closed as normal. Taking the run-tracker case as an example, the user might start the app, and perhaps configure some settings in readiness for starting his run. Then, when he starts his run, he might navigate forward to a media-player app so that he can listen to music while he’s running. Meanwhile, the run-tracker app continues to track his running progress (tracking location in real time, distance covered, average speed, altitude gain, and so on). The app might also provide intermittent feedback to the user in the form of toasts or audible prompts. If the user taps a toast, this can take him back into the app.

Although you could achieve something close to the required behavior in Windows Phone 7, this required you to handle the screen lock events; it was not very elegant, and also it didn’t give you any way to continue executing if the user started another foreground app. Behind the scenes, the system imposes constraints. The reason for these constraints is that if an app is allowed to continue to run while it is not in the foreground, there is a risk that it will consume resources to an extent that might impact the foreground app and therefore the user’s experience of the phone. The resource management model in Windows Phone 8 explicitly supports CBE apps by carefully balancing the competing requirements of the foreground app and the CBE app so that they don’t conflict.



Note CBE apps are listed in the background task control panel on the phone, in the list of “apps that might run in the background.” Because background execution potentially consumes more battery power, the user always has the option to block any or all of these apps from running.

If you want to build a location-tracking CBE app, there are four requirements:

- Your app manifest must specify that the app requires background execution. Without this, your app would be subject to the normal activation/deactivation model and will not be allowed to execute in the background.
- Your *App* class must handle the *RunningInBackground* event. This event is raised at the point when the user navigates forward away from your app, thereby putting your app into the background.

Technically, you can do whatever you like in this handler (or nothing), but what you're supposed to do is to ensure that you trim your operations so that while you're running in the background, you perform only the minimal critical work that is required. At this point, of course, you can no longer access the screen, so there's no point trying to update your UI, for example.

The Windows Phone 8 SDK documentation includes a detailed list of which APIs are permitted during background execution. It is in your own interests to minimize your background work, because behind the scenes, CBE resources are allocated opportunistically. This means that under normal circumstances, you can continue to execute in the background, so long as you don't consume resources excessively. However, for the case in which the phone is under extreme resource pressure (which is more likely on a low-memory device), priority is generally given to the foreground app (and to VoIP), whereas background tasks—including CBE—have a lower priority, and can even be starved of resources altogether. Minimizing your work in the background helps to prevent overall resource pressure.

- The app must be able to handle the relaunch case. The typical scenario is when your app is running in the background but surfaces toasts periodically. When the user taps a toast, this relaunched your app. In the case of CBE apps, the system relaunched the app by resuming the currently running instance, complete with its internal page navigation backstack from the previous user session. In this scenario, you might want to clear the backstack, to avoid confusing the user. See Chapter 2 for more details on the app lifecycle, resume model, and managing the backstack.
- Your app must be actively tracking location.

The *SimpleCbe* solution in the sample code demonstrates the behavior of a CBE app, which is demonstrated in Figure 16-16.

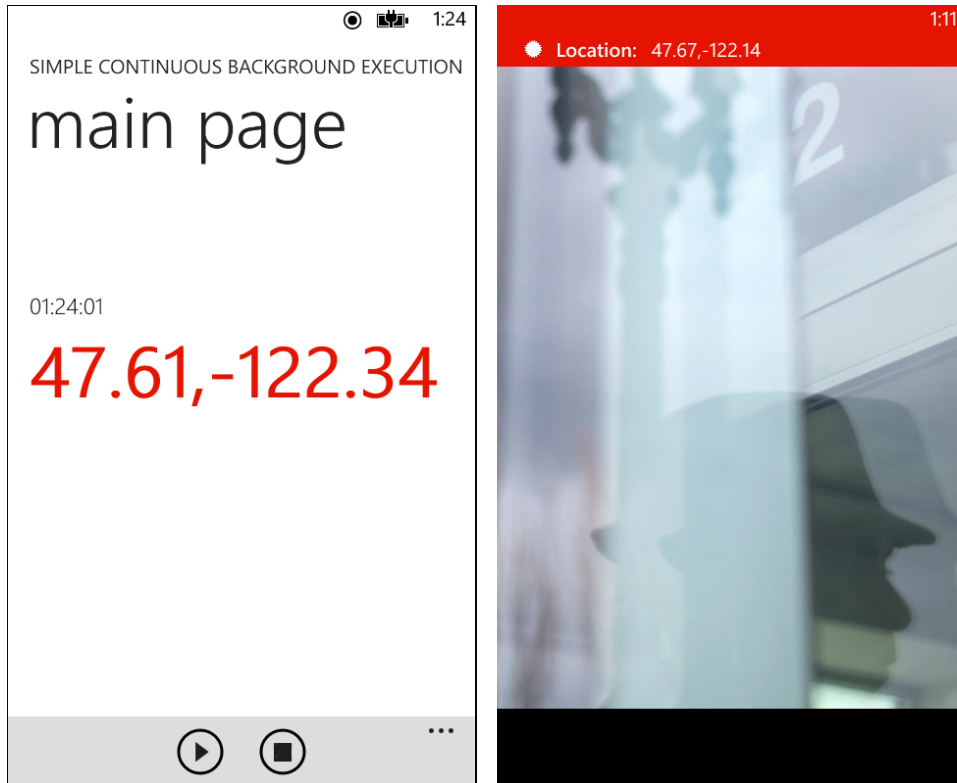


FIGURE 16-16 A CBE app has a normal UI in the foreground (on the left), and uses toasts in the background (right).

For the first requirement, the app manifest must include two items. The app requires *ID_CAP_LOCATION* as normal, because it will be using location features. It might also require *ID_CAP_MAP* if it uses maps. In addition, it must register as a background-capable app, and this is done by adding the *BackgroundExecution* element, as shown in the code that follows. This is a child node of the *DefaultTask* element. Currently, there is only one supported *ExecutionType*, and its *Name* must be set to "LocationTracking". Keep in mind that although you can configure the *ID_CAP_LOCATION* setting by using the manifest designer, there is no designer support for the *BackgroundExecution* element, so you must edit the app manifest manually to add this.

```
<DefaultTask Name="_default" NavigationPage="MainPage.xaml">
  <BackgroundExecution>
    <ExecutionType Name="LocationTracking"/>
  </BackgroundExecution>
</DefaultTask>
```

The second requirement is that the *App* class must register to handle the *RunningInBackground* event. You do this in the *App.xaml* file, where all the other app-level event handlers are registered.

```
<shell:PhoneApplicationService
    Launching="Application_Launching" Closing="Application_Closing"
    Activated="Application_Activated" Deactivated="Application_Deactivated"
    RunningInBackground="Application_RunningInBackground"/>
```

If you use Visual Studio's IntelliSense and AutoComplete feature to register for this event, a stub method will be added to your *App.xaml.cs* file for you. In the sample app, the *App* class has just two custom data members. First, it exposes a *Geolocator* property, suitably initialized in the property getter. This will be used for getting location data. Second, it defines a *bool* property for tracking whether the app is currently running in the background. This is set to *true* in the *RunningInBackground* event handler, and to *false* in the *Activated* event handler. Because this app is registered to run in the background, when the user navigates forward away from the app, the system sends it a *RunningInBackground* event instead of the usual *Deactivated* event.

```
private static Geolocator locator;

public static Geolocator Locator
{
    get
    {
        lock (typeof(App))
        {
            if (locator == null)
            {
                locator = new Geolocator();
                locator.DesiredAccuracy = PositionAccuracy.High;
                locator.MovementThreshold = 50;
            }
        }
        return locator;
    }
}

public static bool IsRunningInBackground { get; private set; }

private void PhoneApplicationService_RunningInBackground(
    object sender, RunningInBackgroundEventArgs e)
{
    IsRunningInBackground = true;
}

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    IsRunningInBackground = false;
}
```

In a CBE app, it's also a good idea to take a closer look at the *Deactivated* event. Recall that the system raises this event for the app when it is deactivating it. The *DeactivatedEventArgs* supplied with this event exposes a *Reason* property, which gives you some insight as to why the app is being terminated.

```
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    switch (e.Reason)
    {
        case DeactivationReason.ApplicationAction:
            Debug.WriteLine("Application_Deactivated: the app did something to cause
termination.");
            break;
        case DeactivationReason.PowerSavingModeOn:
            Debug.WriteLine("Application_Deactivated: power-saver mode is on, and the power went
below the threshold for running CBE.");
            break;
        case DeactivationReason.ResourcesUnavailable:
            Debug.WriteLine("Application_Deactivated: the phone does not have enough resources
(memory, CPU) to allow CBE to continue.");
            break;
        case DeactivationReason.UserAction:
            Debug.WriteLine("Application_Deactivated: the user did something to terminate the
app.");
            break;
    }
}
```



Note If some condition arises at runtime for which you actually want to deactivate the app and not continue to run in the background, you can stop the location data events by unhooking your *PositionChanged* event handler in the *RunningInBackground* handler, and then possibly hook it up again in the *Activated* event.

In the *MainPage* class, the two App Bar buttons are implemented to register or unregister the event handler for the *PositionChanged* event on the *Geolocator* property in the *App* class. This handler does one of two things: if the app is running in the foreground, it extracts the latest coordinate data and sets it into the UI; if the app is not running in the foreground, it composes a toast from the coordinate data and shows that, instead.

```
private void startButton_Click(object sender, EventArgs e)
{
    App.Locator.PositionChanged += Locator_PositionChanged;
}

private void stopButton_Click(object sender, EventArgs e)
{
    App.Locator.PositionChanged -= Locator_PositionChanged;
}
```

```

private void Locator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    Geocoordinate coord = args.Position.Coordinate;
    string coordText = String.Format("{0:0.00},{1:0.00}", coord.Latitude, coord.Longitude);
    if (!App.IsRunningInBackground)
    {
        Dispatcher.BeginInvoke(() =>
        {
            timeText.Text = coord.Timestamp.ToString("hh:mm:ss");
            locationText.Text = coordText;
        });
    }
    else
    {
        ShellToast toast = new ShellToast();
        toast.Content = coordText;
        toast.Title = "Location: ";
        toast.NavigationUri = new Uri("/MainPage.xaml", UriKind.Relative);
        toast.Show();
    }
}

```

The third requirement is less obvious. In the app so far, all the UI work is done in one page: the *MainPage*. However, consider the *SimpleCbe_MultiPage* solution in the sample code. This is a variation on the app in which the location UI is not done on the *MainPage*, but instead on a second page named *LocationPage*. The implication of this is that when the app is running in the background and sends a toast on which the user then taps, it takes her to *LocationPage* not to *MainPage*.

When a background-enabled app is relaunched from the App List or from a toast, the system navigates to the page on the top of the stack (that is, the last page the user was on before she navigated away from the app) by using *NavigationMode.Reset*. Next, the system navigates to the page specified in the *NavigationUri* in the toast by using *NavigationMode.New*. Recall from Chapter 2 that the system maintains a stack of pages to which the user has navigated within the app. It is up to the app developer to decide whether to unload the page stack. The Visual Studio template has been updated to clear the page stack, under the assumption that this is what most people will want to do. If you don't want this behavior, you can remove or modify the code from the hidden region of *App.xaml.cs* (highlighted in bold in the example that follows), specifically, the *CheckForResetNavigation* and *ClearBackStackAfterReset* methods.

In the *Navigated* event handler, the app checks to see if the *NavigationMode* of the navigation is *Reset*. This will be the case if the user has relaunched the app when it is already running in the background. In this case, the app goes on to clear the backstack. Exactly what you do here depends on your business logic; for example, instead of clearing the backstack, you might want to cancel the incoming navigation and redirect to another page.

```

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

```

```

RootFrame = new PhoneApplicationFrame();
RootFrame.Navigated += CompleteInitializePhoneApplication;
RootFrame.NavigationFailed += RootFrame_NavigationFailed;

RootFrame.Navigated += CheckForResetNavigation;

phoneApplicationInitialized = true;
}

private void CheckForResetNavigation(object sender, NavigationEventArgs e)
{
    if (e.NavigationMode == NavigationMode.Reset)
        RootFrame.Navigated += ClearBackStackAfterReset;
}

private void ClearBackStackAfterReset(object sender, NavigationEventArgs e)
{
    RootFrame.Navigated -= ClearBackStackAfterReset;
    if (e.NavigationMode != NavigationMode.New && e.NavigationMode != NavigationMode.Refresh)
        return;
    while (RootFrame.RemoveBackEntry() != null) { ; }
}

```



Note You can take advantage of the CBE model in an app that doesn't track location, if you want. The reason this might be useful is if you want the "resume" behavior, where your app is relaunched with its previous backstack. In Windows Phone 7, this behavior was only available to in-box apps, but there are scenarios in which it is valid for any app. To achieve "resume" behavior, you need to mark your app *LocationTracking*, even though you don't actually track location. When an app is marked *LocationTracking* and you actively track location, the system will allow your app to run in the background and also apply the Fast App Resume (FAR) policy. If you don't actually track location or if another location-tracking app is started, the system will suspend your app as normal. The difference is that when the user taps on one of your tiles, the system resumes the app (as opposed to launching it from scratch), provided it is still in the backstack and was not tombstoned.

Testing Location in the Simulator

To test your location-based app, you can use the Visual Studio location sensor simulator. This is available from the Location tab in the Additional Tools window in the emulator. To use this, you start your app and start the location-tracking feature. In the simulator, toggle the Live button on. This generates location information for your current location. You can then click anywhere on the map to generate further locations. This is illustrated in Figure 16-17. You can also use the simulator to record a chain of locations, and play these back at any time. You can even save a chain of locations to a file so that it persists beyond the life of an emulator session and can be loaded in a later session.

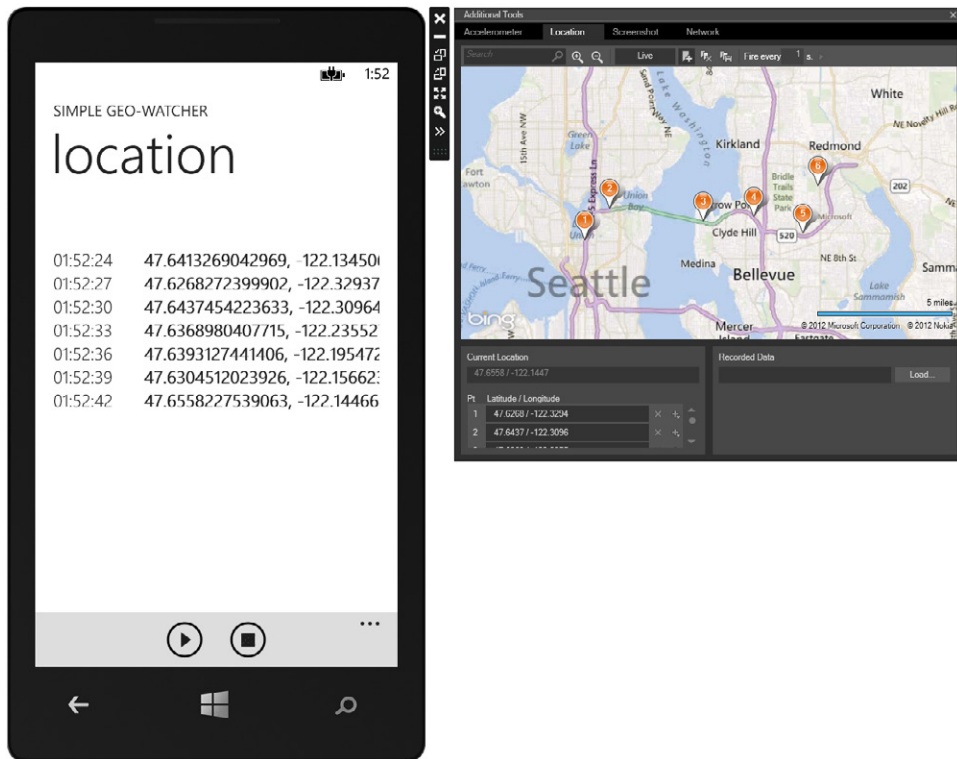


FIGURE 16-17 You can use the location sensor simulator to test your location-based app.

Location Best Practices

As noted in Chapter 6, “Sensors,” all sensors on the phone use power, and that includes the sensors used in location services. The location sensors can consume a lot of power (reducing battery life) because of the way they fetch and report data very frequently, and because it involves keeping the GPS radio on. You should therefore consider very carefully how you want to use classes such as the *GeoCoordinateWatcher* and the *Geolocator*. You should avoid running them without notifying the user, and you should always give the user control over starting and stopping them. When the user is running a location tracking app, the phone shell displays a corresponding “bullseye” icon in the system tray. In some apps, the use of these features is implicit and obvious from the nature of the app, but it’s still a good idea to inform the user that you’re doing this, and preferably to ask his permission at least once, in advance. You should also turn them off when not needed and remove all event handlers. Consider using the default accuracy setting, which is less accurate but consumes less power. For example, use the default setting if you only need a less specific location, such as the general city area for things like weather or basic personalization.

You are also encouraged to set the *MovementThreshold* property on the *GeoCoordinateWatcher* or *Geolocator* class. This is the distance in meters that the phone must move, relative to the last *PositionChanged* event, before the location provider raises another *PositionChanged* event. To optimize battery life, the recommended setting is above 20. An appropriate setting depends on the nature of your app: if you're tracking position changes for someone walking, a smaller setting might be more appropriate. On the other hand, if you're tracking movements of a car, a much higher value will probably be more useful (unless you're building a turn-by-turn feature). Setting *MovementThreshold* to a higher value will save some CPU time because the app platform does not report all values from the sensor, but it does not prevent the sensors from retrieving the data. So, although this does save on battery consumption, it doesn't save very much.

You can use the Reactive Extensions for location data in exactly the same way as for accelerometer data, discussed in Chapter 6. This way, you can sample the data stream and apply a filter of some kind. However, this also does not prevent the underlying system from sourcing the sensor data, nor does it prevent the app platform from propagating the data changed events to your app, so again, there are no battery savings with this approach.

Summary

You have an array of choices for building location-aware apps, with or without integrated maps. Your choice of the version 7 Bing-related APIs or the new enhanced version 8 APIs largely depends on whether you're cross-targeting version 7 and version 8, or cross-targeting Windows Phone 8 and Windows 8. The programming model and supported techniques are broadly similar, even though the specific classes are different. Windows Phone 8 also provides the ability to continue executing in the background. As with all sensor-based apps, you should carefully consider the battery impact of getting location data, especially if you're building a CBE app.

Speech

Mobile phones have always been on the cutting edge of new user interface (UI) paradigms. With limited screen real estate and the need of users to complete tasks quickly while on the run, solutions that might have worked well on a desktop computer must be rethought or thrown out entirely. Until recently, however, most of the innovation in mobile phone interaction focused on tactile input—ways for users to control the phone by using their fingers, whether via a physical keyboard, a stylus pen, or a capacitive touch screen. The advent of speech as a means of interacting with modern smartphones is exciting not just for the existing tasks it makes easier (for example, saying, “Call Terry Adams at home,” to initiate a phone call) but for its potential to open up an entire new genre of apps and experiences on the phone. Of course, in some ways it’s ironic that the latest craze in phones is the ability to talk into them, but as we’ll show in this chapter, the possibilities of a modern speech platform are far beyond anything of which Alexander Graham Bell could have dreamt.

When it comes to speech in Windows Phone 8 apps, there are three topics of interest. First, you can register voice commands, which can then be invoked by the user from the Global Speech Experience (GSE); for example, “Movies, Show What’s Playing Now.” Second, you can use APIs for Speech Recognition (SR) to accept speech input within the apps themselves, including both command-and-control and free-form dictation. Finally, you can use the phone’s Text-to-Speech (TTS) framework to read out text as audio. The latter two features can also be combined to provide a responsive spoken dialogue similar to what is available in the built-in Messaging app. (“Message from Kim Abercrombie.” You can say, “Read it” or “Ignore.”)

Voice Commands

In Windows Phone 7.x, users were able to perform a number of specific tasks directly from the GSE, the speech dialog that is launched when the user presses and holds the hardware Start button. These tasks included making phone calls, sending text messages, and issuing Bing search queries. Users could also use the speech functionality to *open* Store apps, but nothing more. That is, you could say, “Open Weather,” but not, “Open Weather and show me the forecast for Phoenix.” In Windows Phone 8, developers can provide a rich set of voice commands with which users can launch Store apps to perform specific tasks with all of the necessary context provided by speech. This section will describe how to register the actions that your app supports and how to handle being launched from speech.

The GSE

Because all of the user scenarios in this section originate from the GSE, we should start by reviewing the user experience (UX) for that component so that you understand how users will trigger your voice commands. This will help you to determine how best to expose your app to users through this UI.

As in Windows Phone 7, users can activate the GSE by pressing and holding the hardware Start button. When they do so, they see a screen that looks similar to Figure 17-1.

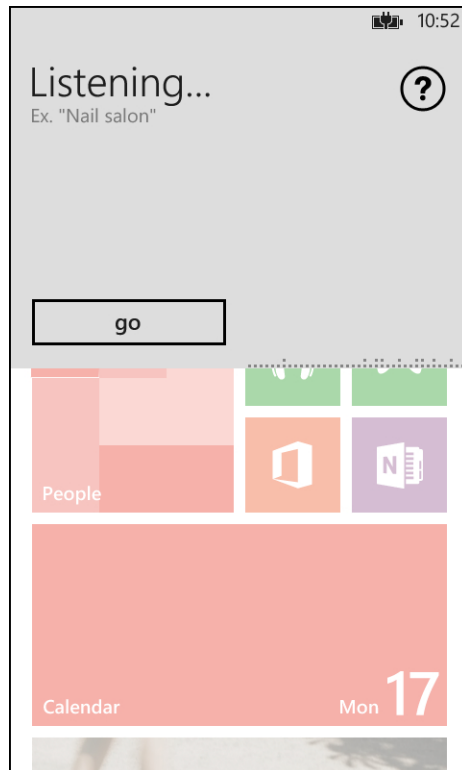


FIGURE 17-1 Users activate the GSE by pressing and holding the hardware Start button.

After the GSE displays and says that it's "Listening...," users are free to make verbal requests into the phone's microphone. The GSE is sensitive enough to know when they have completed their requests, at which point it will begin trying to match it to a valid action on the phone (or perform a Bing search).

Although most users know what they want to do when launching the GSE, a small help button is provided to show what else can be done with speech. Tapping the Help icon in the upper-right of the GSE brings up a full-screen display titled "What Can I Say?" This provides a list of valid voice

commands. In Windows Phone 8, this screen includes a new pivot for Apps that lists Windows Phone Store apps that support voice commands, as shown in Figure 17-2. Tapping one of the apps provides a list of commands supported by that app.



FIGURE 17-2 The GSE's Apps pivot shows which apps support voice commands.

You will probably notice that throughout the experience, there is a considerable emphasis on providing example commands that can help users to build up a repertoire of tasks that they can complete by voice. As a developer, when you define voice commands, you also have an opportunity to provide such examples. Ensure that you take advantage of this chance to expose some of your most valuable commands.



Tip If you have a microphone set up on your computer, you can use the Windows Phone Emulator to perform all the same search commands that you can on a device. You can also use the F2 key to activate the GSE rather than pressing and holding the emulator's Start button.

Building a Simple Voice Commands App

Now that you understand the user entry point for voice commands, it's time to build an app that can expose them. The *ToDoList* solution in the sample code demonstrates some of the capabilities of the speech API. It contains two pages, one for listing existing items and one for adding new items.

Figure 17-3 shows the basic UI.

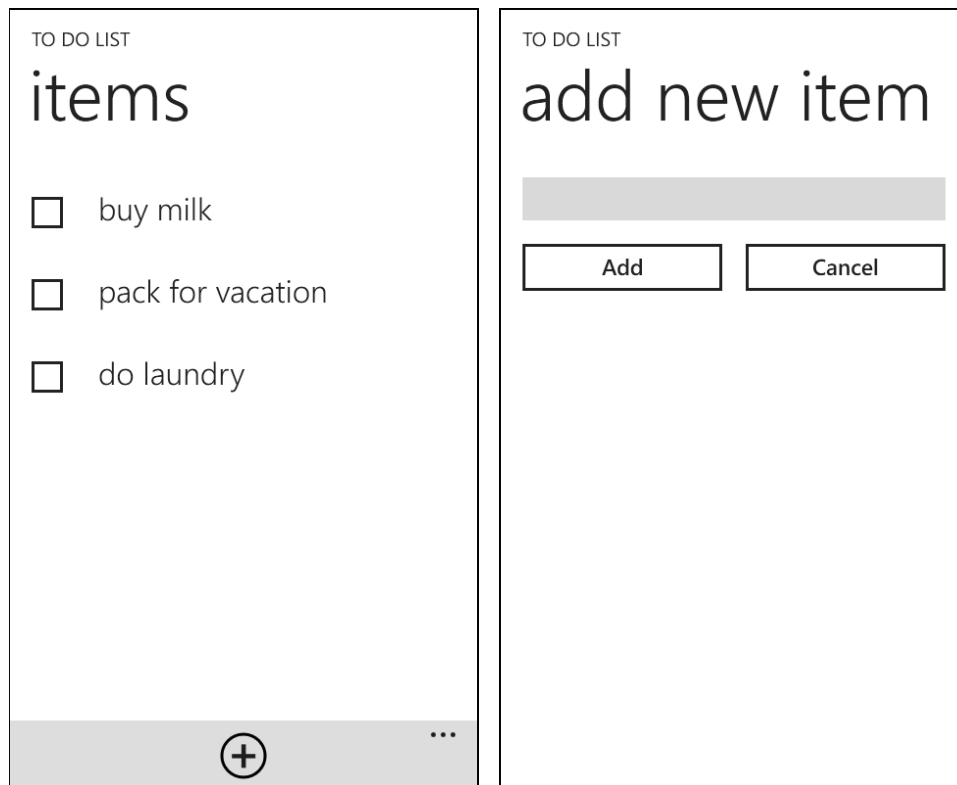


FIGURE 17-3 With the *ToDoList* solution, users can manage a simple list of to-do items.

Registering Voice Commands

To list the voice commands that your app supports, you first need to create a Voice Command Definition (VCD) file. The VCD file describes the format of the command that is expected, what the device should say back to you when it recognizes the command, and which page in your app should be invoked to handle it. Microsoft Visual Studio Express 2012 for Windows Phone includes a default template for VCD files, making it easy to add one to your project. Simply right-click your project, and then in the options menu that opens, click Add New Item, and then click Voice Command Definition from the list of available items, as shown in Figure 17-4.

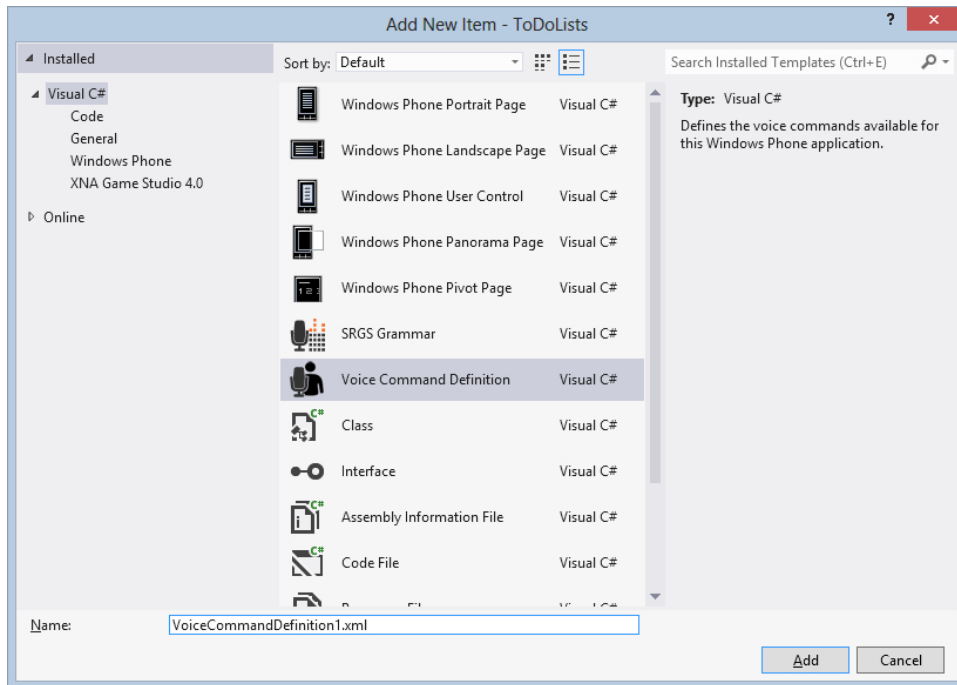


FIGURE 17-4 The Add New Item dialog box presents a default template for VCD files.

The default template includes sample commands for a game, including “play new game.” For the *ToDoList* solution, we will start by defining a simple voice command which can create a new item. The full VCD file required to enable this command is shown in the following:

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

<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="en-US">
    <CommandPrefix>To Do List</CommandPrefix>
    <Example>create new item</Example>

    <Command Name="CreateNewItem">
      <Example>create new item</Example>
      <ListenFor>create [a] new item</ListenFor>
      <ListenFor>add [an] item</ListenFor>
      <Feedback>creating a new item... </Feedback>
      <Navigate Target="/NewListItemPage.xaml" />
    </Command>
  </CommandSet>
</VoiceCommands>
```

VCD files offer a concise and powerful way to describe the voice commands that your app can support, so it is worth understanding what all of the elements mean. Table 17-1 describes the basic schema for VCD files.

TABLE 17-1 Voice Commands Schema

Element	Attribute	Description
<i>VoiceCommands</i>		Top level element for all voice commands. This can contain many <i>CommandSet</i> children.
	<i>Xmlns</i>	Always http://schemas.microsoft.com/voicecommands/1.0 .
<i>CommandSet</i>		Contains all voice commands for a given culture (for example, "en-us").
	<i>xml:lang</i>	An IETF-style culture code, such as "en-us" or "de-de." When this value matches the current culture of the phone (configurable in Settings), the speech service tries to match the given commands; otherwise, they will be ignored. Keep in mind that command sets are not shared across culture codes, even when the language is the same. Thus, if you want to have the same set of voice commands supported for "en-us" and "en-gb," you need to create two copies of the <i>CommandSet</i> , one for each code.
	<i>Name</i>	An optional name to be used as an index into the list of installed command sets from code.
<i>CommandPrefix</i>		The pronunciation of app names doesn't always match up with their phonetic parts. Using this optional element, you can spell out an alternative name for your app that matches the way it sounds.
Example		A single example voice command for this <i>CommandSet</i> . This example is used in several places: In the apps pivot of the GSE On the "What can I say?" page for the app On some error screens On the "Listening..." screen—the system will cycle through examples from built-in apps and those installed from the Windows Phone Store.
<i>Command</i>		The container for a single voice command.
	<i>Name</i>	This value is used to specify to your app which command it was launched to handle, so it must be unique within the <i>CommandSet</i> .
Example		An example phrase for this specific command. These example phrases will be shown in the "Did You Know?" screen for the app.
<i>ListenFor</i>		The text for which the speech recognizer should listen when trying to match what users are saying to a specific command. You can have up to 10 <i>ListenFor</i> elements for each <i>Command</i> , which should make it possible for you to account for the different ways that a user might try to invoke the command. In the simple <i>ToDoList</i> solution that you just saw, we included two <i>ListenFor</i> elements: "create a new item" "add an item" You can denote optional words with square brackets, as we did with the indefinite articles "a" and "an" in the preceding commands. By making "a"/"an" optional, the speech recognizer will not only listen for these phrases, but also for: "create new item" "add item" By thoughtful definition of your <i>ListenFor</i> elements, you can maximize the likelihood that users will speak a command that will be correctly matched with a task that your app supports. <i>ListenFor</i> elements also support the ability to include lists of potential words within the phrase, known as <i>PhraseLists</i> , which you can even update dynamically. We'll review <i>PhraseLists</i> in detail later in the chapter.

Element	Attribute	Description
<i>Feedback</i>		Where the <i>ListenFor</i> element describes what the system should listen for in what users are saying, the <i>Feedback</i> element describes what the system should say back to users when the command has been recognized. This serves to inform users whether the voice command that they uttered was correctly recognized by the system so that they can determine whether to follow through with launching the app or cancel it and try again.
<i>Navigate</i>		The element that defines where in your app the system should send users if this voice command is recognized.
	<i>Target</i>	A relative URI to the page in your app that will handle the invocation when this command is matched. You can also include custom query parameters here. Technically, this attribute is optional as the system will launch the main page of your app if nothing is specified. This gives you the opportunity to use a custom <i>URIMapper</i> to do more granular parsing of the incoming command before determining which page will handle it.

There are actually a few more elements in the VCD schema, offering an interesting set of dynamic scenarios. You will look at those shortly.

Initializing the VCD File

Now that you've registered the voice commands that you want to support in your VCD file, there's just one more step before they will be recognized by the system through the GSE: initialization.

When your app runs for the first time, use the *InstallCommandSetsFromFileAsync* WinPRT method from the *Windows.Phone.Speech.VoiceCommands.VoiceCommandService* class to pass in your VCD file for processing. For the *ToDoList* solution, we named the VCD file *ToDoListVoiceCommands.xml*, so the full command would be the following:

```
await VoiceCommandService.InstallCommandSetsFromFileAsync
    (new Uri("ms-appx:///ToDoListVoiceCommands.xml"));
```

Note the use of the "ms-appx" URI scheme prior to the filename. This scheme is shorthand for the app's install directory, which generally matches the file structure of the original project and the final XAP package. Because our VCD file resides at the root of the package, nothing else is required. If you choose to store your VCD files in a subdirectory of your project, you will need to include the entire relative path after "ms-appx:///".

Once this line of code is executed, your voice commands will be registered with the speech service and ready to be recognized. To be sure, you can do the following:

1. Activate the GSE by pressing and holding the hardware Start button.
2. Tap the Help button (the question mark icon in the upper-right corner).
3. Swipe over to the Apps pivot.

You should see your app listed, along with the top-level example you provided for your *CommandSet*. If you tap your app name, you will see the list of voice commands that have been registered, which so far is just one, as shown in Figure 17-5.

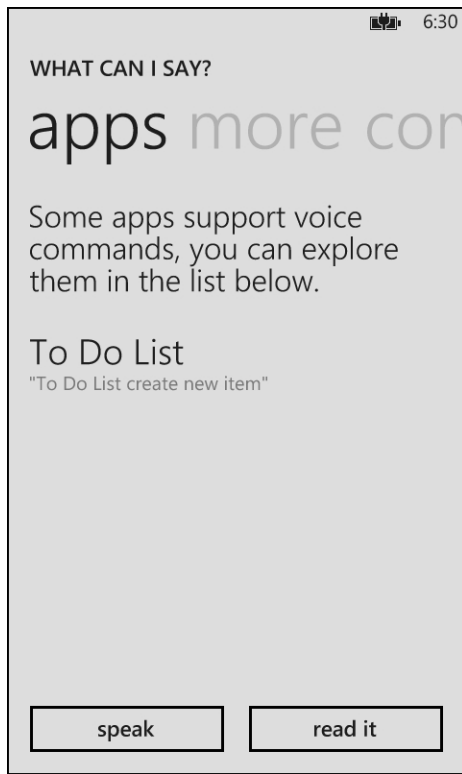


FIGURE 17-5 After initializing your VCD, the GSE lists your supported commands.

Handling Invocation for Simple Voice Commands

For simple voice commands, such as the “create new item” command that you just created, there’s relatively little for your app to do upon invocation. Assuming that you’ve set the appropriate target page on the *Navigate* element in your VCD file, the system will just launch your app and direct the user to the correct page.

However, although you might never notice it for simple commands, there is a significant amount of metadata about the initial voice recognition coming into your app through this invocation. This will become more interesting when you get to dynamic commands, but first, you should understand what’s happening in the basic case now so that layering on the dynamic commands will be an incremental addition.

Recall from Chapter 2, “App Model and Navigation,” that the Windows Phone navigation model is highly analogous to the web in that it is made up of a series of pages, and those pages pass data to one another through query string parameters. This is true not only between pages within an app, but also between system components and your app. In the case of the simple *ToDoList* solution and the “create new list” voice command, the part of the URI that’s interesting looks like this:

```
/NewListItemPage.xaml?
voiceCommandName=CreateNewItem&reco=To%20Do%20List%20create%20new%20item
```

This query string has two potentially useful pieces of information:

- **voiceCommandName** This contains the name of the *Command* that was matched by the speech recognizer.
- **reco** This contains the full text of what the speech recognizer matched, including the app name.

Note that the value of *reco* includes a set of spaces that have been replaced with their percent-encoded representation of “%20”. All URI values passed to your app from the system are similarly percent-encoded.

You will add more query string parameters as you delve into more advanced scenarios in the next section. For now, all you need to know is that this is how all data will be fed from the speech service into your app.

Dealing with Deep Links

Directly invoking app functionality via voice commands is an example of an app “deep link,” whereby an app exposes a page to which the system can navigate directly rather than requiring users to start from the main page and find their way there. The most common place you will encounter deep links is in the form of secondary Start tiles, when an app creates a tile to launch directly to a specific page or piece of content, such as a particular city in a weather app.

Adding deep links to your app might require you to rethink how you move between pages. Consider the *ToDoList* solution in this example. It contains a *MainPage*, which includes a button in the App Bar to add a new item. Tapping that button navigates the user forward to *NewListItemPage.xaml*. Prior to adding deep links to the app, there is exactly one way reach *NewListItemPage*: from *MainPage*. In other words, you can safely assume that any time users find themselves on *NewListItemPage*, there must be an instance of *MainPage* right behind it on the backstack and a simple way to leave *NewListPage* when the user is finished adding the list would be to call *NavigationService.GoBack()*. Once you provide a deep link to that page from speech or elsewhere, however, that will no longer work; in fact, *NavigationService.GoBack()* will throw an exception because you cannot programmatically navigate back out of your app.



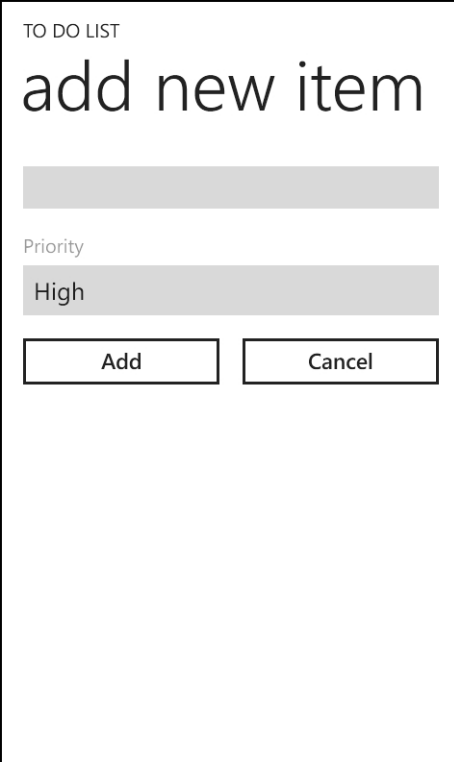
Note Although the absence of deep links in your app might guarantee that users navigated to a particular page from the main page, you cannot assume much else about the state of that main page when navigating to a secondary page. As is pointed out in Chapter 2, if the app has been resumed from a tombstoned state, the main page object will not have been instantiated yet. In general, you should not depend on a particular user flow through your app for ensuring that it is in a given state.

There are several solutions to this. Assuming that you want to keep *NewListPage* as an entirely separate page in your app (rather than, for example, making it a full screen *Popup* over your *MainPage*), the best approach is to perform a forward navigation via *NavigationService.Navigate(Uri)* to the *MainPage*. Of course, because you probably do not want to keep that *NewListPage* instance sitting around after returning to the *MainPage*, you should use *NavigationService.RemoveBackEntry()* to clean it up. See Chapter 2 for more details on backstack management.

Adding Flexibility with Labels

Now that you have a basic understanding of how to register voice commands and how those commands will be received in your app, it's time to make things a little more interesting.

Those of us who are compulsive creators of to-do lists know that it isn't enough to have a list; you must a *prioritized* list! You will create a new property on the list item to contain its priority, which will be one of three states: high, normal, or low. Users can set that priority when creating a new item. Figure 17-6 shows the new UI for the *NewListItemPage*.



TO DO LIST

add new item

Priority

High

Add Cancel

FIGURE 17-6 Lists can now be assigned a priority.



Note The Priority field here is set by using the *ListPicker* control, which is included in the Silverlight Toolkit for Windows Phone.

There are two ways that you can add this priority field to the VCD file. You could create new commands called *CreateNewHighPriorityItem*, *CreateNewNormalPriorityItem*, and *CreateNewLowPriorityItem*, but that would culminate in a significant amount of duplication when all you really want is to add a parameter to the existing *CreateNewItem* command; enter the *PhraseList*.

With a *PhraseList*, you can specify a set of acceptable values for a given part of a command and then receive the chosen value when your app is invoked. Phrase lists are defined per *CommandSet* and then referenced within the individual commands. The following updated VCD file illustrates the addition of the priority field to the *CreateNewItem* command:

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

<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="en-US" Name="todoVoiceCommands">
    <CommandPrefix>To Do List</CommandPrefix>
    <Example> create new high priority item </Example>

    <Command Name="CreateNewList">
      <Example> add a high priority item </Example>
      <ListenFor> create [a] new {priority} priority item</ListenFor>
      <ListenFor> add [a] {priority} priority item</ListenFor>
      <Feedback> Creating a new {priority} priority item... </Feedback>
      <Navigate Target="NewListItemPage.xaml" />
    </Command>
    <PhraseList Label="priority">
      <Item>high</Item>
      <Item>normal</Item>
      <Item>low</Item>
    </PhraseList>
  </CommandSet>
</VoiceCommands>
```

Now, the user can include any of the specified priority strings in his voice commands; for example, "create a new high-priority item." Note the addition of the priority label to the *Feedback* element. The GSE inserts the matched value for priority here when reading back to the user what it heard.

Of course, what you actually want to do is use the priority provided by the user through speech to automatically set the priority in the *ListPicker* on the *NewListItemPage*. Recall that all information coming into the app from the system is in the form of query string parameters. As you saw earlier, the most basic commands yield query string parameters for the name of the voice command that was matched and the full string that the speech service recognized. With phrase lists, the app receives an

additional parameter for each of the phrase lists in the matched command, with the value that the speech service recognized for that phrase list. In the case of the user command "create a new low priority item," the query string looks like this:

```
/NewListItemPage.xaml
?voiceCommandName=CreateNewItem
&reco=To%20Do%20List%20create%20a%20new%20low%20priority%20item
&priority=low%20priority
```

The app can then read that parameter in its *OnNavigatedTo* event and set the *ListPicker* appropriately, as shown in the following:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    string voiceCommand;
    bool launchedByVoice =
        NavigationContext.QueryString.TryGetValue
            ("voiceCommandName", out voiceCommand);

    if (launchedByVoice && voiceCommand == "CreateNewItem")
    {
        string priority;
        NavigationContext.QueryString.TryGetValue("priority", out priority);

        SetPriorityListPicker(priority);
    }
}

private void SetPriorityListPicker(string priority)
{
    /**
     * ListPicker priorities are defined in this order:
     * High: 0
     * Normal: 1
     * Low :2
     */
    switch (priority)
    {
        case "high ":
            priorityListPicker.SelectedIndex = 0;
            break;
        case "normal ":
            priorityListPicker.SelectedIndex = 1;
            break;
        case "low":
            priorityListPicker.SelectedIndex = 2;
            break;
        default:
            priorityListPicker.SelectedIndex = 1;
            break;
    }
}
```

There is actually one small catch here. *PhraseList* tokens are not optional, nor can they contain empty values. If you include a *PhraseList* token in your *Command*, all of its values must be non-empty and the user must utter one of them in order for the *Command* to be matched. In the previous *CreateNewItem* example, this means that the addition of a priority *PhraseList* made the basic command, “create new item,” no longer recognizable. Thus, if you want to have a basic version of a command as well as one that takes parameters, you need to create separate *Command* elements in your VCD file.



Note The full version of the *ToDoList* project shown in the code sample creates one voice command for non-prioritized items (*CreateNewItem*) and one for prioritized items (*CreateNewPrioritizedItem*).

Updating *PhraseLists* at Runtime

Adding *PhraseLists* to a VCD file is a good way to avoid creating individual *Commands* for every possible input value, but because the VCD file is a static part of the app package, it can only be updated when the user updates the app from the store and cannot contain user-specific values. If you want to support voice commands based on content that the user actually created within the app, you will need to update your *PhraseLists* at runtime.



Note Technically, you can also create new VCD files at runtime by passing a file in your local storage to *InstallCommandSetsFromFileAsync*. However, because entirely new commands will generally require new handler code in your app, this should be rare.

For the to-do list app, the obvious candidates for dynamic commands are the to-do items themselves. This gives you the opportunity to issue a voice command to check off items as you complete them. You will add a new *Command*, called *MarkItemComplete*, which follows the same pattern as those shown in the previous section, and is listed here:

```
<Command Name="MarkItemComplete">
  <Example>mark 'buy milk' complete</Example>
  <ListenFor>mark {todoItem} complete</ListenFor>
  <ListenFor>{todoItem} is complete</ListenFor>
  <ListenFor>{todoItem} is done</ListenFor>
  <Feedback>marking "{todoItem}" complete</Feedback>
  <Navigate Target="MainPage.xaml" />
</Command>
```



Note The use of quotes around *{todoItem}* in the *Feedback* element is simply to demarcate the item itself in the feedback string shown in GSE, such that it shows something like “marking “buy milk” complete.”

The definition of the *PhraseList* you're naming *todoItem* is slightly different, however. Because you cannot know what users will add to their list at runtime, you cannot include any items in the static VCD file. Instead, it is defined as an empty set.

```
<PhraseList Label="todoItem">
</PhraseList>
```

Now, you need to add the code to update the *todoItem PhraseList* with the items the user has entered. The natural place to do this is immediately after the user has added an item to the list, so you will add a small helper method called *UpdateTodoPhraseList* to the code-behind of *NewListItem Page*, which you can then call from the *addButton* click event handler.

```
private static async void UpdateTodoPhraseList()
{
    VoiceCommandSet voiceCommands =
        VoiceCommandService.InstalledCommandSets["todoVoiceCommands"];
    await voiceCommands.UpdatePhraseListAsync
        ("todoItem", App.ListItems.Select(todoItem => todoItem.ItemName));
}
```



Note You can dynamically update any *PhraseList*, even those that do contain items in the VCD file. This is useful if you have a set of default values for a phrase that you want to update from a web service or based on user input.

The first line retrieves the appropriate *CommandSet* from the speech service by using its *Name* attribute. The second line updates the *todoItem PhraseList* that you defined earlier in the VCD file by using an *IEnumerable* collection of strings. In your case, those strings are the item names. Notice that *UpdatePhraseListAsync* performs a complete replacement of the existing list of items in the *PhraseList*. Even if you only want to add one or two items to an existing *PhraseList*, you must remember to include the existing items in the call to *UpdatePhraseListAsync*.



Note You are limited to a maximum of 2000 items across all *PhraseLists* in a given *CommandSet*.

Speech Recognition in Apps

The same core technology that enables voice commands can also be used to recognize speech within apps, with a great degree of flexibility in how the UX is presented and how speech is matched. There are two dimensions for which you will need to decide upon the level of customization that you want to provide: the speech recognition UI to show, and the grammar against which to match. You will

begin by showing looking at an example in which the app offloads virtually all of the work to the built-in UI and speech recognizer, and then proceed to show ways that you can customize each of these. Keep in mind that the dimensions are independent; you can use the highly customized speech recognizer with the built-in UI, or vice-versa.

Simple Recognition with Built-In UX

The easiest way to add speech recognition is to use the provided system UI and Microsoft's cloud-based speech service for converting a user's utterance to real text. This makes sense for scenarios in which you want to allow the user to enter free-form text, such as entering text into a *TextBox*. You will use this to provide a way for users to enter the name of a new to-do list item by using speech.

In Windows Phone, the standard UI cue for speech input being available in an app is a microphone icon, so you will add a microphone button next to your item name *TextBox*, as shown in Figure 17-7.



Note We've mimicked the UI of the speech-enabled *TextBox* from the Bing app by layering the microphone's *Button* on top of the *TextBox*. We've done this by laying out the core content of the page as a grid with three rows and two columns. The *TextBox* spans both columns of the first row, whereas the microphone's *Button* shares the second column. It's important to ensure that the *Button* is defined *after* the *TextBox* in the XAML; otherwise, the *TextBox* will sit above it and the *Button* will not be clickable. It's also a good idea to set the *Padding* attribute for the right side of the *Textbox*. This will ensure that even if the user types a long string, the characters will not run into the microphone icon. We've built all of this UI directly into the page to keep the sample code in one place; in a real app, it would be better to split this out into a separate *UserControl*.



Note The standard Windows Phone microphone icon is available in the set of icons installed with the Windows Phone SDK and can be found at %PROGRAMFILES%\Microsoft SDKs\Windows Phone\v8.0\icons\Dark\microphone.png (substitute Light for Dark to get the appropriate image for the light theme).

When the user taps that microphone, you want to launch the system-provided speech recognition UI. That UI is contained in a single class, *Windows.Phone.Speech.Recognition.SpeechRecognizerUI*. The UI associated with this class is very similar to the GSE discussed in the previous section, but it runs in the context of an app and allows for some customization of UI elements. In particular, you can set the top-level label for the control ("Listening..." in the GSE) and the example text to something that is appropriate for your app.

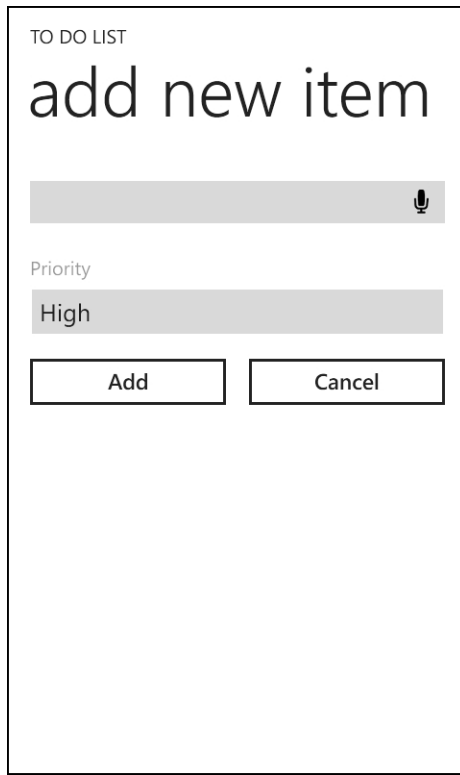


FIGURE 17-7 A microphone icon is the standard UI cue that indicates speech input is available.



Note The fact that the speech recognizer UI is now running in the context of the app is a subtle distinction from the perspective of the users (they might not even notice), but it makes a difference to the security model. Because the speech input is now being delivered directly to the app, it must include the *ID_CAP_MICROPHONE* capability, in addition to *ID_CAP_SPEECH_RECOGNITION*.

You will create a small helper method called *ReceiveItemNameBySpeech*, which you will then call from the click event handler for the microphone button.

```
private async void ReceiveItemNameBySpeech()
{
    SpeechRecognizerUI todoItemNameRecognizer = new SpeechRecognizerUI();
    todoItemNameRecognizer.Settings.ListenText = "Add your item...";
    todoItemNameRecognizer.Settings.ExampleText = "eg. buy milk";

    SpeechRecognitionUIResult result =
        await todoItemNameRecognizer.RecognizeWithUIAsync();
    newListItemNameTextBox.Text = result.RecognitionResult.Text;
}
```

As you can see, the *SpeechRecognizerUI* class is straightforward. The *ListenText* and *ExampleText* properties set two pieces of text in the built-in speech UI control, whereas *ShowConfirmation* and *ReadoutEnabled* determine how the control behaves after the speech has been recognized, namely whether the control shows and/or reads what it recognized before passing it back to the app. The string that the speech recognizer matches is available from the *SpeechRecognitionResult* object that is returned from *RecognizeWithUIAsync*.

Customizing the Recognizer UI

For most speech input scenarios, the built-in UI that is invoked by *SpeechRecognizerUI* is sufficient. Indeed, because it mimics the look-and-feel of the GSE, it can be preferable to custom UI because it is familiar and intuitive to the user. In some cases, however, it will make sense to display custom UI or no explicit listening UI at all. The *ToDoList_CustomSpeechRecognizerUI* solution shows an example of custom speech recognition UI.



Note Although it is possible to perform speech recognition without any explicit “Listening...” UI, you should always ensure that your experience makes it clear to the user when the device is listening, either through voice prompts or other UI cues.

You will repurpose the microphone icon added in the previous section to act as a subtle UI cue that speech recognition is enabled. When the user clicks the microphone, it will start rotating and continue doing so until the recognizer has stopped listening for input. See the full sample code for the *ToDoList_CustomSpeechRecognizerUI* solution for the definition of the animation storyboard, which we’ve named *RotatingMicrophone*.

You will create a new method called *ReceiveItemNameBySpeechWithCustomUI*, which will replace *ReplaceltemNameBySpeech* from the *ToDoList* solution. The new method is shown in the code that follows. Note that the microphone button is disabled while speech is being recognized; this ensures that the user does not inadvertently stack up recognition requests. It was not necessary when using the built-in UI earlier, because that UI obscures the foreground app and makes it non-interactive automatically.

```
private async void ReceiveItemNameBySpeechWithCustomUI()
{
    microphoneButton.IsEnabled = false;
    RotatingMicrophone.Begin();

    using (SpeechRecognizer recognizer = new SpeechRecognizer())
    {
        SpeechRecognitionResult result = await recognizer.RecognizeAsync();
    }
}
```

```

        if (result.TextConfidence != SpeechRecognitionConfidence.Rejected)
        {
            newListItemNameTextBox.Text = result.Text;
        }
    }

    RotatingMicrophone.Stop();

    microphoneButton.IsEnabled = true;
}

```



Note There is a small chance that *RecognizeAsync* could throw an exception in the preceding code sample. Before the user provides speech input to the system for the first time, she must accept the speech privacy policy. Because it's unlikely that a user will use speech for the first time ever inside your app, we've chosen not to clutter the code with an additional try/catch block here. The full project that accompanies this chapter does show the handler code, however. It's also worth noting that *RecognizeAsyncWithUI* handles the acceptance of the privacy statement on the app's behalf.

Adding Custom Grammars

If you've tried both voice commands and simple speech recognition as you've seen in the chapter thus far, you probably noticed that the recognition accuracy is much higher for voice commands. The reason for this is simple: with voice commands, the speech recognizer has a relatively small number of possible inputs against which to match—perhaps a few hundred—and each valid command must match a specific format (an app name, followed by one of the handful of commands defined for that app). With open-ended speech, there are potentially thousands of valid matches for a given utterance and very little in guaranteed structure, making it much more difficult to match it accurately.

A good analog for this is a foreign language that you might have studied but never mastered. If you stick to checking into hotels, ordering food in restaurants, and buying postcards, your high school French will probably be good enough for a week in Paris, because each of those interactions is structured enough to limit the number of things you can expect to hear from your Parisian interlocutor. The cashier at the Louvre gift shop is much more likely to tell you how much you owe and ask whether you want a bag than to start expounding on the subtleties of Sartre, so you can hone in on those phrases and more easily decipher them.

In the case of apps, the way to provide this refinement is to provide the speech recognizer with grammars. There are two simple ways to add grammars to the speech recognizer, and one more advanced way. You will review the simpler methods first.

Adding Predefined Grammars

The first option is to take advantage of Microsoft's predefined grammars, of which there are two supported in Windows Phone 8: dictation and web search. The dictation grammar is the same one used by the built-in messaging app and is therefore optimized for very short messages (150 to 200

characters), which are returned as complete sentences; that is, starting with a capital letter and ending with a period. The web search grammar is also used by a built-in app (in this case, Bing) and is usually a better option for short utterances that don't map to sentences. Both of these grammars rely on cloud-based processing, so they require a network connection in order to work, something you should keep in mind when designing your workflow and considering error cases. You will also need to ensure that your app has the *ID_CAP_NETWORKING* capability; even though the network traffic is abstracted away by the speech framework, it is ultimately still happening on behalf of your app. The following code shows the *ReceiveItemNameBySpeech* method from the previous section updated to include loading the *WebSearch* grammar via *AddGrammarFromPredefinedType*:

```
private async Task ReceiveItemNameBySpeechWithCustomUI()
{
    RotatingMicrophone.Begin();

    using (SpeechRecognizer recognizer = new SpeechRecognizer())
    {
        recognizer.Grammars.AddGrammarFromPredefinedType
            ("websearch", SpeechPredefinedGrammar.WebSearch);
        SpeechRecognitionResult result = await recognizer.RecognizeAsync();

        if (result.TextConfidence != SpeechRecognitionConfidence.Rejected)
        {
            newListItemNameTextBox.Text = result.Text;
        }
    }

    RotatingMicrophone.Stop();
}
```

The first parameter to *AddGrammarFromPredefinedType* (and the other *AddGrammar* APIs we will discuss) is a grammar name. You can use this to reference the individual grammars that you've added to your recognizer later. You will understand the value of this after reviewing the second type of simple grammar.

Adding Simple Lists

Using the predefined grammars should help to improve the accuracy of recognition for free-form speech that falls into either the dictation or web search category, but most scenarios for speech in apps will be far more specialized, with commands and content that are specific to that app and the portion of its UI with which the user is currently interacting. For those cases, you can achieve significantly higher degrees of accuracy by further constraining the grammars against which the speech recognizer is matching.

In your *NewListItemPage*, the obvious candidate for a highly constrained grammar is the list of priority options. This is a simple list—just three one-word options—but it's one that is specific to this scenario within the app, so the recognizer will benefit greatly from the constraint. In this case, you will use the *AddGrammarFromList* API which allows for the creation of simple grammars from any *IEnumerable* collection of *string*. You encapsulate the recognition in a new helper method named

ListenForPriority, which will be triggered after the user adds an item name through speech by tapping the microphone icon. You again use a subtle visual cue to indicate when the device is listening for speech input. In this case, you will show a pulsating border in the device theme color around the *ListPicker* (the animation, named *BorderAnimator*, is also available in the sample code for this chapter).

```
private async Task ListenForPriority()
{
    priorityListPicker.BorderBrush =
        App.Current.Resources["PhoneAccentBrush"] as SolidColorBrush;

    BorderAnimator.Begin();

    using (SpeechRecognizer priorityRecognizer = new SpeechRecognizer())
    {
        priorityRecognizer.Grammars
            .AddGrammarFromList("priority", new string[] { "high", "normal", "low" });
        SpeechRecognitionResult result = await priorityRecognizer.RecognizeAsync();

        if (result.TextConfidence != SpeechRecognitionConfidence.Rejected)
        {
            SetPriorityListPicker(result.Text);
        }
    }

    BorderAnimator.Stop();
    priorityListPicker.BorderBrush = new SolidColorBrush(Colors.Transparent);
}
```

Assuming that the *SpeechRecognizer* does not reject the result, you pass the result to the *SetPriorityListPicker* helper method, which was shown earlier in this chapter.

Notice again that the API for adding a new grammar (in this case, *AddGrammarFromList*) takes a name to identify it within the recognizer which, as mentioned earlier, makes it possible for you to reference the grammar later. This is important because this gives you the ability to easily enable and disable grammars as necessary on your *SpeechRecognizer* instance, which is faster and more efficient than constantly creating new *SpeechRecognizer* objects, particularly if you are using the same grammars multiple times.

Why might you want to disable a grammar? Remember that the speech recognizer is ultimately performing a probabilistic pattern matching operation when it tries to convert the user's speech input into something that the app can understand. The likelihood of it being able to correctly understand the input is inversely correlated with the number of options it has to choose from.

To illustrate this, imagine an app for a camping store that offers speech as a way of searching for products. They might have a grammar which contains those products and includes entries for (among other things) "tent" and "boots." If the user chooses boots, the app might want to further narrow his search by asking him to provide a shoe size, in which case it will require a grammar for the valid sizes, which will naturally include the word "ten." If the app is using a common *SpeechRecognizer* instance and keeps all grammars active at the same time, that recognizer will need to discern the difference between "ten" and "tent" if the customer happens to have size-10 feet—a challenging task. However,

because the app knows that the user is providing an input for shoe size, there's no need to even consider the options contained in the product grammar, and the app should disable that grammar altogether.



Note In the code samples thus far, we've elected to show a transient instance of *SpeechRecognizer* to keep things simple while you learn the basics of speech. In a real app, you would want to have a longer-lived object even in these basic scenarios because of the potential for the user to be unsatisfied with the first recognition result. Later in the chapter, we will show an example of using a long-lived *SpeechRecognizer* instance with grammars that are enabled and disabled as necessary. There is one caveat, however: a single *SpeechRecognizer* instance cannot contain a mixture of predefined grammars (that is, the Microsoft-provided dictation or web search grammars) and user grammars (either lists or the Speech Recognition Grammar Specification grammars covered in the next section).

Adding Grammars by Using Speech Recognition Grammar Specification

The third option for adding a grammar to the speech recognizer is through an XML file which conforms to the Speech Recognition Grammar Specification (SRGS), a W3C standard for speech recognition. Using SRGS, apps can provide much richer grammars for more complex scenarios, most of which are outside the scope of this book. However, the *ToDoList_SRGS* solution includes a simple example of how apps can work with SRGS.

For this example, you return to the *MainPage* of the app, which shows all of your to-do items, and add a second app bar button, again taking advantage of the microphone icon to indicate speech. You add an SRGS XML file to our project through the Add Item Wizard and then replace the default template with the following XML:

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

<grammar
    version="1.0" xml:lang="en-US" root="MarkTasksComplete"
    tag-format="semantics/1.0" xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:sapi="http://schemas.microsoft.com/Speech/2002/06/SRGSExtensions">

    <rule id="MarkTasksComplete" scope="public">
        <one-of>
            <item>Mark</item>
            <item>Set</item>
            <item>Make</item>
        </one-of>
        <item repeat="0-1">all</item>
        <one-of>
            <item repeat="0-1"> high priority <tag>out.priority="high"</tag></item>
            <item repeat="0-1"> normal priority <tag>out.priority="normal"</tag></item>
            <item repeat="0-1"> low priority <tag>out.priority="low"</tag></item>
        </one-of>
    </rule>
</grammar>
```

```

<one-of>
  <item>tasks</item>
  <item>items</item>
  <item>to dos</item>
</one-of>

complete
</rule>
</grammar>

```

The structure and content of the file is mostly self-evident. The intent is to provide the framework of a valid utterance so that the speech recognizer can match it appropriately. By providing multiple options for a given phrase using the *one-of* element, you can make it easier for the user to succeed in speaking to the app without having to learn the exact format and wording of the accepted commands.

The portion of the XML that might not be so intuitive are the tag elements found within each item in the list of priority options. Tags allow creators of SRSR grammars to create simple key-value pairs which are set when the speech recognizer hits that portion of the grammar in a match. For the app that needs to parse the recognition result, these tags make it easier to pull out the variable portions of the utterance which might impact how the app behaves. Observe how in this example, you only include tags for the priority options because that is the only part of the grammar that makes a difference in what the app will ultimately do. We've included different options for the statement's verb ("mark", "set", "make") and noun ("tasks", "items", "to-dos") to provide flexibility to the user, but it makes no difference to the behavior of the app which one of those the user chooses to say, so you don't need to create tags for them.



Note The syntax for tags is `out.key="value"`.

In this case, you do all of the work in the click event handler for the app bar button showing the microphone icon. As a means of illustrating that you can mix and match custom grammars with the built-in UI, you will once again use the *SpeechRecognizerUI* class here, rather than the *SpeechRecognizer* class that you've used for the last few examples. Note, however, that *SpeechRecognizerUI* is just a wrapper over *SpeechRecognizer*, which adds the UI-specific properties—a full-fledged *Speech Recognizer* instance is available as a member of *SpeechRecognizerUI*. The full code for the event handler is shown in the following:

```

private async void speechAppBarButton_Click(object sender, EventArgs e)
{
    using (SpeechRecognizerUI recognizerUI = new SpeechRecognizerUI())
    {
        recognizerUI.Settings.ListenText = "Listening...";
        recognizerUI.Settings.ExampleText = "eg. Mark all high priority tasks complete";
        recognizerUI.Settings.ReadoutEnabled = false;
        recognizerUI.Settings.ShowConfirmation = false;
    }
}

```

```

recognizerUI.Recognizer.Grammars
    .AddGrammarFromUri("MarkCompleteIncomplete",
        new Uri("ms-appx:///ToDoGrammar.xml", UriKind.Absolute));

SpeechRecognitionUIResult result = await recognizerUI.RecognizeWithUIAsync();

if (result.RecognitionResult.TextConfidence == SpeechRecognitionConfidence.Rejected)
{
    return;
}

SemanticProperty priority;

// check if the user specified that only high, normal, or low priority
// tasks should be marked complete
// otherwise mark everything complete
if (result.RecognitionResult.Semantics.TryGetValue("priority", out priority))
{
    foreach (ToDoListItem todoItem in
        App.ListItems.Where(item => item.Priority == (string)priority.Value))
    {
        todoItem.IsCompleted = true;
    }
}
else
{
    foreach (ToDoListItem todoItem in App.ListItems)
    {
        todoItem.IsCompleted = true;
    }
}
}
}

```

The first thing you might notice in the preceding code is that the audio readout and confirmation screen on the built-in UI are turned off. If you believe that your grammar is tightly constrained and the likelihood of misinterpretation is low, this is a good way to speed up the overall experience, especially if you are asking the user to provide multiple voice inputs.

The next thing you do is load the *ToDoGrammar.xml* file that you created earlier. Because the file resides at the top level of the app package directory, you use the *ms-appx* URI scheme to denote its path.

Preloading Grammars

The *SpeechRecognizer* class includes a method called *PreloadGrammarsAsync*, with which it can parse any grammars that the app has provided prior to the first request to recognize speech. This call is optional, but because it performs an action that will need to be done regardless, it's a good idea to get it out of the way *before* the user has tapped the button to start speaking, especially for cases in which you are providing large or complex grammars that might take some time for the recognizer to ingest. By preloading those grammars, you can avoid any delays in beginning to recognize the user's speech, which will ultimately lead to higher accuracy. It is not shown in the previous code sample, because that is all running in the context of the *Button*'s click event handler.

When *RecognizeWithUIAsync* returns, you need to determine whether the user provided a priority qualifier to his request. This is where the tags that you added to the grammar file come in. Even though the recognition result contains the complete version of the string matched to the user's input, you only care about whether the user included a priority in his command and, if so, which priority it was. Recall that tags are simply a set of key-value pairs, so they naturally surface in the result as an *IReadOnlyDictionary*. You just need to check whether the priority tag is present and, if so, what its value is. Keep in mind that you can perform a simple equivalence check between the value of the tag and the *Priority* property of *ToDoListItem* because they're both defined as simple lowercase strings with possible values "high", "normal", and "low".

Text-to-Speech

So far in this chapter, you have covered topics that involve the user speaking into the phone and having that speech translated into actions or text for the app to handle. The reverse direction is also possible with Text-to-Speech (TTS), by which apps can turn text content into spoken words that can be played through the phone's audio outputs.

Simple TTS with *SpeakTextAsync*

The *ToDoList_TTS* solution shows how you can use TTS to read out the number of incomplete tasks when the app starts. The critical class in this case is *Windows.Phone.Speech.Synthesis.SpeechSynthesizer*, which offers a *SpeakTextAsync* method taking a single string. You will add a simple helper method called *ReadNumberOfIncompleteTasksAsync*, which will be invoked when the main page loads.

```
private async Task ReadNumberOfIncompleteTasksAsync()
{
    SpeechSynthesizer synth = new SpeechSynthesizer();
    int numberOfIncompleteTasks = App.ListItems
        .Count(item => item.IsCompleted == false);
    await synth.SpeakTextAsync("You have " + numberOfIncompleteTasks + " incomplete tasks.");
}
```

Taking Control with Speech Synthesis Markup Language

The *SpeakTextAsync* method works well for simple strings. For more sophisticated TTS, however, you need to provide guidance to the speech synthesizer.

Crafting the Speech Output

Now that the user knows how many incomplete tasks she has, she will probably want to know what they are, so you will try reading those out with the *SpeechSynthesizer*, as well. You will add another button to the app bar of your *MainPage* which will provide a quick way for the user to get a read-out of her incomplete tasks. Figure 17-8 illustrates the new UI.



Note The play icon (*transfer.play.png*) is available in the Icons folder of the Windows Phone SDK.

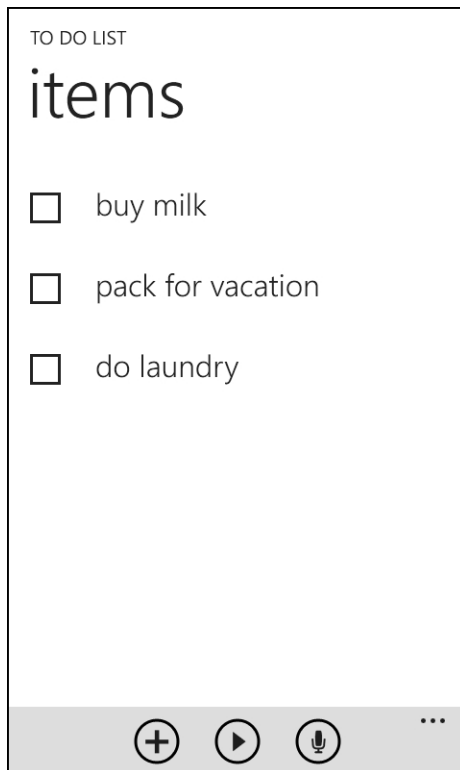


FIGURE 17-8 The app can now read out your incomplete tasks—no more excuses for putting off that laundry!

You will create a new helper method, *ReadAllIncompleteTasksAsync*, to read out the tasks, which will be invoked by the click event handler for the app bar button. Clearly, the easiest way to implement that method would be to concatenate the names of all incomplete items together into one

long string and pass that to *SpeakTextAsync*. The problem is that once you translate a distinct set of to-do items into a single concatenated string, there's no notion of separation between them and the synthesizer reads them out as if they were a single incoherent sentence.

Thankfully, there is a way to give the synthesizer some hints on how to read the text by using a W3C standard XML format known as Speech Synthesis Markup Language (SSML).



Tip You can find the W3C spec for SSML v1.0 at <http://www.w3.org/TR/speech-synthesis/>. The Windows Phone speech synthesizer supports this standard with only very minor differences.

Using SSML, apps can control the pace, pronunciation, and emphasis of spoken text. For this particular task, you use the SSML element *break* to add a one second pause between each task. The code for *ReadAllIncompleteItemsAsync* is shown in the following example:

```
private async Task ReadAllIncompleteItemsAsync()
{
    readItemsAppBarButton.IsEnabled = false;

    string incompleteItemsListWithSSML =
        "<speak version=\"1.0\" xml:lang=\"en-us\">";

    foreach (ToDoListItem item in App.ListItems.Where(item => item.IsCompleted == false))
    {
        incompleteItemsListWithSSML += item.ItemName + "<break time=\"1s\" />";
    }

    incompleteItemsListWithSSML += "</speak>";

    SpeechSynthesizer synth = new SpeechSynthesizer();
    await synth.SpeakSsm1Async(incompleteItemsListWithSSML);

    readItemsAppBarButton.IsEnabled = true;
}
```

The SSML standard offers a high degree of customizability to TTS, most of which is beyond the scope of this book. However, Table 17-2 lists several elements worth highlighting if you are interested in adding polish to your speech output without investing significant time in creating detailed pronunciation specifications.

TABLE 17-2 SSML Offers Numerous Ways to Customize Text-to-Speech Output

Element	Description	Example
<i>say-as</i>	Specifies how dates, times, numbers, and groups of characters are pronounced.	SSML input: This task is due on <say-as interpret-as="date" format="mdy">01-15-2013</say-as>. Audio output: This task is due on January fifteenth, two thousand thirteen.
<i>emphasis</i>	Places enhanced or reduced emphasis on a word or phrase.	This task is <emphasis level="strong">high</emphasis> priority.
<i>voice</i>	Sets the gender and language of the speech synthesizer voice.	<voice xml:lang="en-gb" gender="male">You have three incomplete high-priority tasks</voice> This text will be read in a male British voice (assuming that language is present on the phone).

Keeping Speech and Screen In Sync

Apps that use speech as an alternative way to deliver content to the user will naturally run into the question of how to keep the visual output of the app synchronized with what the speech synthesizer is saying. An ebook reader app, for instance, will probably want to “turn the page” on the user’s behalf as the synthesizer reads along. This can be done by using the *mark* SSML tag and the *BookmarkReached* event on the *SpeechSynthesizer*.

You will demonstrate this in the to-do list app by putting a small border around the item currently being read out in the device theme color. First, you need to make a small modification to *ReadAllIncompleteItemsAsync* to add the *mark* tags and register a handler for the *BookmarkReached* event which will be raised as the synthesizer passes over those tags. Notice how in addition to adding a tag before each item, there is one added for the end of the list; this makes it possible for you to clear the formatting from the final item in the list.

```
private async Task ReadAllIncompleteItemsAsync()
{
    string incompleteItemsListWithSSML = "< speak version=\"1.0\" xml:lang=\"en-us\" >";
    foreach (ToDoListItem item in App.ListItems.Where(item => item.IsCompleted == false))
    {
        incompleteItemsListWithSSML += "< mark name=\"" + item.ItemName + "\" />";
        incompleteItemsListWithSSML += item.ItemName + "< break time=\"1s\" />";
    }

    incompleteItemsListWithSSML += "< mark name=\"EndOfList\" />";
    incompleteItemsListWithSSML += "< /speak >";

    SpeechSynthesizer synth = new SpeechSynthesizer();
    synth.BookmarkReached += synth_BookmarkReached;
    await synth.SpeakSsmlAsync(incompleteItemsListWithSSML);
}
```

The handler for *BookmarkReached* uses name of the bookmark to find the item in the list and then uses the *ItemContainerGenerator* member of *ListBox* to find the actual *ListBoxItem* to which the *ToDoListItem* is bound, because that's the visual element you need to change.



Note A real app would do most of this visual manipulation by using data binding and value converters. However, for the purposes of consolidating the logic in one place, we have chosen to show it in code.

```
ListBoxItem currentListBoxItem = null; // reference to the currently highlighted list box item

void synth_BookmarkReached(SpeechSynthesizer sender, SpeechBookmarkReachedEventArgs args)
{
    Deployment.Current.Dispatcher.BeginInvoke(() =>
    {
        // reset the last item to its default state before highlighting the next one
        // this will also set the last item in the list to its default state
        // when the EndOfList bookmark is hit
        if (currentListBoxItem != null)
        {
            currentListBoxItem.BorderBrush = new SolidColorBrush(Colors.Transparent);
        }

        if (args.Bookmark == "EndOfList") return;

        ToDoListItem currentItem =
            App.ListItems.First(item => item.ItemName == args.Bookmark);

        currentListBoxItem = (ListBoxItem)toDoListItemsListBox
            .ItemContainerGenerator.ContainerFromItem(currentItem);

        if (currentItem != null)
        {
            currentListBoxItem.BorderBrush =
                App.Current.Resources["PhoneAccentBrush"] as SolidColorBrush;
        }
    });
}
```

Putting It Together to Talk to Your Apps

Now that you've seen how to use SR and TTS separately, it's time to bring them together and show how to implement an experience wherein users can actually participate in an interactive dialog with apps. The *ToDoList_Dialog* solution includes the code for this section.

In the Voice Commands section earlier in this chapter, you included a command for creating a new item in the list. You will build on that initial command to provide a completely voice-driven way of adding a new item. In particular, you will use the version of the command that did *not* accept a priority by voice command, because that will make up one part of the dialog in this example.

When the app is launched, you check the incoming query string parameters to *NewListItemPage* to determine if it was launched to handle the *CreateNewItem* voice command. If so, you call a new helper method named *AddItemBySpeech*. This method implements a simple dialog, asking the user to provide the item name and priority and then confirm the addition. This method in turn relies on another helper method named *ListenForInput* to actually receive the successive commands, looping until it correctly recognizes an utterance.

```
protected async void AddItemBySpeech()
{
    this.IsEnabled = false;

    List<string> yesNoOptions = new List<string>() { "yes", "no" };
    List<string> priorityOptions = new List<string>() { "high", "normal", "low" };

    using(SpeechSynthesizer synthesizer = new SpeechSynthesizer())
    using(SpeechRecognizer userGrammarRecognizer = new SpeechRecognizer())
    using(SpeechRecognizer dictationRecognizer = new SpeechRecognizer())
    {
        dictationRecognizer.Grammars
            .AddGrammarFromPredefinedType("dictation", SpeechPredefinedGrammar.Dictation);

        userGrammarRecognizer.Grammars.AddGrammarFromList("YesOrNo", yesNoOptions);
        userGrammarRecognizer.Grammars.AddGrammarFromList("Priorities", priorityOptions);

        await userGrammarRecognizer.PreloadGrammarsAsync();

        userGrammarRecognizer.Grammars["Priorities"].Enabled = false;

        string newItemName =
            await ListenForInput
                ("Say the name of the item", synthesizer, dictationRecognizer);
        newListItemNameTextBox.Text = newItemName;

        string isItemCorrect =
            await ListenForInput
                ("Was that correct?", synthesizer, userGrammarRecognizer);

        if (isItemCorrect == "no")
        {
            // Start over
            AddItemBySpeech();
            this.IsEnabled = true;
            return;
        }
        else if(isItemCorrect == "cancel")
        {
            this.IsEnabled = true;
            return;
        }

        userGrammarRecognizer.Grammars["Priorities"].Enabled = true;
        userGrammarRecognizer.Grammars["YesOrNo"].Enabled = false;
    }
}
```

```

        string priority =
            await ListenForInput
                ("Set the item's priority. You can say 'high', 'normal', or 'low'",
                 synthesizer, userGrammarRecognizer);
        userGrammarRecognizer.Grammars["Priorities"].Enabled = false;
        userGrammarRecognizer.Grammars["YesOrNo"].Enabled = true;

        await synthesizer.SpeakTextAsync("Setting priority to " + priority);
        SetPriorityListPicker(priority);

        string isUserReady =
            await ListenForInput
                ("Are you ready to add this item?", synthesizer, userGrammarRecognizer);

        if (isUserReady == "yes")
        {
            await synthesizer.SpeakTextAsync
                ("Adding " + newListItemNameTextBox.Text + " to your to-do list.");
            await AddItemAndReturnToMainPage();
        }
        else
        {
            return; // let user proceed without speech
        }
    }

    this.IsEnabled = true;
}

private async Task<string> ListenForInput
    (string requestToRead, SpeechSynthesizer synthesizer, SpeechRecognizer recognizer)
{
    string input = string.Empty;
    bool inputRecognized = false;

    // keep attempting to recognize speech input until we get a match
    do
    {
        await synthesizer.SpeakTextAsync(requestToRead); // eg. "Set the item's priority"

        SpeechRecognitionResult srr = await recognizer.RecognizeAsync();
        if (srr.TextConfidence != SpeechRecognitionConfidence.Rejected)
        {
            input = srr.Text;
            inputRecognized = true;
        }
        else
        {
            await synthesizer
                .SpeakTextAsync("Sorry, didn't catch that. Please try again.");
        }
    }
}

```

```
        while (inputRecognized == false);

        return input;
    }
```

Observe the reuse of the *userSpeechRecognizer* object for handling confirmation of the item name, the selection of priority, and the final confirmation of item addition. Recall from the speech recognition section that it is more efficient to reuse a *SpeechRecognizer* object if you intend to handle multiple inputs against the same grammar than to constantly create new instances. It is much easier for the speech service to turn grammars that have already been parsed on and off than to reparse them from scratch. On the other hand, the *SpeechRecognizer* does need to allocate memory to store the parsed grammars, so it is not advisable to maintain global static instances that contain grammars from across the app which might get used at widely varying rates. The code sample illustrates the disabling of a grammar by turning off the “Priorities” grammar after the user has chosen a priority. Also recall that user and system grammars cannot be combined in a single *SpeechRecognizer* instance; hence, the creation of a separate *dictationGrammar*.

Summary

In this chapter, you learned the three core parts of the Windows Phone speech framework:

- Voice commands
- Speech recognition
- Text-to-Speech

Some apps, such as those providing hands-free, turn-by-turn navigation, will necessarily include speech as a core part of their experience, but all developers should consider intelligently incorporating speech into their apps. In many cases, it offers a faster, more convenient, and just plain fun way to interact with smartphones that takes very little code to light up.

