UWP

Succinctly®

by Matteo Pagani

Syncfusion | Technology Resource Portal

UWP Succinctly

By
Matteo Pagani

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: James McCaffrey

Copy Editor: Jacqueline Beringer, content producer, Syncfusion, Inc.

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Darren West, Content Producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Matteo Pagani is a developer with a strong passion about mobile development and a strong focus on the Windows platform. After graduating in computer science in 2007, he started his first job experience as a web developer.

However, in the following years, he started to grow an interest in the mobile world and to discover the community world. He started to share his knowledge with other developers by co-founding a local community, opening a blog, writing articles for Microsoft and other independent technical websites, and speaking at technical conferences across Europe and the United States like TechReady, Microsoft Tech Summit, .NET Spain Conference, MVP Summit, Codemotion, WPC, and more.

He is the author of many books about Windows Phone, Windows, and Universal Windows Platform apps development.

He has been a Microsoft MVP in the Windows Platform Development category and a Nokia Developer Champion for almost 5 years (from 2011 to 2014) until, in November 2014, he joined Microsoft as a Windows AppConsult Engineer, supporting developers from all over the world in bringing their apps to the Windows ecosystem.

Chapter 1 Introduction

In the past, the association between Microsoft and Windows was automatic. Windows is, without any doubt, the most famous product (together with Office) created by the software house from Redmond. And today, despite all the changes in the computing world, Windows is still the most widely adopted operating system in the world, both by consumers and enterprises.

Today, Microsoft doesn't mean just Windows anymore. The new digital transformation has deeply changed the Microsoft mission. The original goal of the company was to bring a computer to every house and every office, but the world has changed. Computers aren't just used for technical and business purposes anymore. Additionally, new kinds of computing experiences have emerged, like mobile phones and virtual reality. Consequently, the Microsoft mission, under the leadership of the new CEO Satya Nadella, has changed to embrace the quick transformation happening in the digital world: empower all people and every company in the world to achieve more. This goal can't be achieved with just traditional computers. Mobile, cloud, Internet of Things, and business intelligence are just a few of the technologies opening new and exciting scenarios every day. As a result, in recent years Microsoft has started to embrace a deep change by embracing a new and bold approach. They pushed the cloud, turning Azure into one of the most productive cloud platforms in the world; they started to work with the open-source world by collaborating with communities to improve development tools and frameworks; they started to release their products and services on competitive platforms, like Android, iOS, and OS X, because Microsoft wants to help people be productive no matter which is their platform of choice.

Despite these changes, Windows remains one of the key products of the company and the best environment to empower people and companies to achieve their goals, no matter if we're talking about playing the latest triple A game or writing a document in collaboration with a work team.

Windows 10 is the current generation of Windows and it's simultaneously an end and a starting point of a journey. It's the end of a convergence path with the goal of unifying all efforts to create a modern operating system running on modern devices; it's the start of a new concept of operating system, more agile and able to adapt more quickly to the continuous changes that are happening in the digital world.

This book is dedicated to developers and to the Universal Windows Platform, the new development platform that can be leveraged to create modern and powerful applications, capable of running not just on traditional desktops, but on all kinds of modern devices, from the most traditional ones (like mobile phones or tablets) to the most innovative technologies, like 2-in-1s, the Internet of Things, or virtual reality.

The convergence journey

Windows has always been a very conservative ecosystem with a strong focus on productivity and long-term commitment and support, rather than being known for its innovative user experience or its disruptive features.

However, with the release of Windows Phone 7 in 2010, something changed. For the first time, Microsoft took the bold approach by releasing a platform with a completely new user experience and a new UI language, both from consumer and developer points of view. It was something new in the mobile world that offered a completely different experience from its predecessor, Windows Mobile.

The change introduced in Windows Phone slowly started to spread across all the other products of the Microsoft family. At first, the focus was on the unification of the user experience: distinctive features like tiles and a Store to look for apps and games in started to appear on other Microsoft platforms, like Xbox and Windows, with the release of Windows 8.

At the same time, Microsoft also started a convergence journey from a technical point of view. The goal was to create more synergy among teams that were working on a similar product (an operating system) that, in the end, they were just running on different devices.

Windows 8 was the first step of this convergence, sharing the same kernel that was used for the Xbox One and Windows Phone 8. For the end user or the developer, this change didn't have such a big impact. However, under the hood, it had a huge impact for OEMs and the engineering teams, because one kernel means that the same set of drivers can be used across multiple devices, reducing the effort required to support all the different Windows platforms.

Meanwhile, Windows 8 also introduced a development platform called Windows Runtime (WinRT), which leveraged a completely different approach from the traditional .NET framework. Instead of a managed environment that can be installed on multiple Windows versions, the Windows Runtime is a native layer of APIs that can talk directly with the Windows kernel. We're going to get into more detail when we talk about the Universal Windows Platform. Windows Runtime introduced a lot of benefits for apps developers. It's faster than a managed environment, and it offers a broader set of APIs to solve many of the challenges developers experience in the modern world, like cloud access, integration with sensors and Bluetooth devices, optimization for battery power and performance, etc. However, the original version of the Windows Runtime had a downside: it was supported only by Windows 8. Consequently, despite Windows 8 and Windows Phone 8 sharing many of the same features, they were implemented in a different way. Windows 8 was based on the Windows Runtime, while Windows Phone 8 was still based on Silverlight and the traditional .NET Framework. Therefore, the convergence was more theoretical than practical. To support both platforms, a developer was still required to write two completely different applications.

The new update of both operating systems (Windows 8.1 and Windows Phone 8.1) took the journey to the next step and, finally, the end users and developers started to gain some benefits from this convergence. Windows 8.1 expanded the Windows Runtime by adding many missing features required by developers. Additionally, the improved version of Windows Runtime appeared for the first time on the mobile platform with Windows Phone 8.1. There were still some differences to handle (due to the different natures of phones and desktops or tablets), but most of the codebase could be shared between the two platforms, making it much easier for developers to target both platforms with the same application.

This change introduced benefits for the end users, too. Switching between Windows and Windows Phone was a breeze, thanks to a common user experience and a common set of apps. Windows Runtime's new features (like the concept of roaming storage) made it easier for developers to keep data and information in sync across the platforms.

However, the convergence wasn't complete yet. Thanks to the concept of **shared projects** (which allow users to easily share code among multiple projects) it was easy to share code but, in the end, developers still had to deal with two different platforms, which meant handling two different projects in Visual Studio, two different build packages, two different Stores in which to publish their apps, and doubling their efforts in order to reach both platforms.

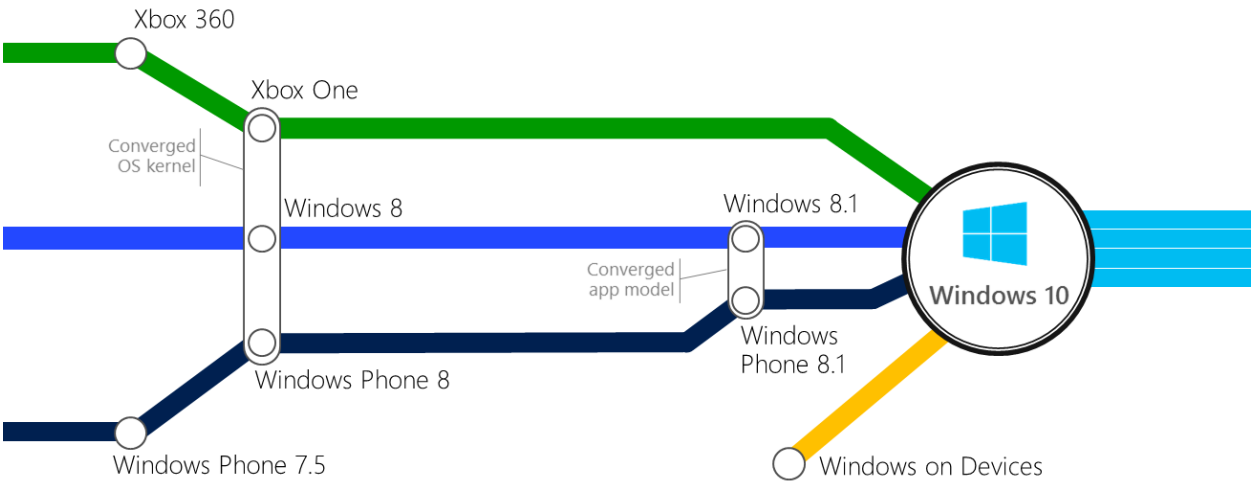


Figure 1: The platform convergence journey.

Windows 10: The end of a journey

Windows 10 was the last step in the convergence journey and, finally, it brings all the pieces to the right place. With Windows 10, Microsoft achieves the goal of having a common operating system, a common kernel, and a common development experience across multiple platforms by removing all the idiosyncrasies that made developers' lives more complicated in the past.

Windows 10 is a single operating system capable of running on multiple devices. No matter if it's a phone, tablet, traditional desktop, or gaming console, they all share the same core, the same kernel, and the same set of APIs, thanks to an evolution of the Windows Runtime called Universal Windows Platform. The only difference among the devices is in the shell, which is the interface leveraged by the user to interact with the device. A traditional desktop offers a shell optimized for mouse and keyboard; a phone or tablet has a shell optimized for touch interaction; an Xbox One has a shell optimized for game controllers.

The app experience has also been unified. As a developer, you won't have to deal anymore with multiple projects, one for each platform. Instead, you'll have a single project capable of running on any device and, thanks to the new features added to the development platform (like adaptive triggers and capabilities detection, which we're going to explore in the other chapters), the application can be easily optimized for the various shells by handling key factors like screen size or input control. This unification is also reflected in the Store. The discovery and download experience is now the same across every platform, which means that an app bought on a phone will be automatically also available on the PC or console since, under the hood, the user will be downloading the same app.

Last but not the least, Windows 10 has also expanded the Microsoft ecosystem by supporting new and innovative devices:

IoT Core is a Windows 10 version dedicated to makers who want to leverage the power of the Universal Windows Platform and the Windows Core to create new experiences in the Internet of Things world, a trending topic nowadays. Windows 10 IoT Core can run on microcomputers and boards like the Raspberry PI, which are very popular among makers and manufacturers to connect traditional objects to the Internet in order to collect and analyze data.

Surface Hub is a device dedicated especially to business and enterprises that can easily become a game changer in conference rooms and collaborative scenarios. It's a computer with a huge touch screen (either a 55'' or 84'' screen) and a 100point multitouch sensor. It comes with a Windows 10 shell capable of running UWP apps and with a set of built-in productivity tools that are optimized to empower collaboration scenarios (like conference calls, shared draw boards, etc.).

HoloLens is the outcome of eight years of research and development in virtual reality and augmented reality, the first of its kind in a new category called mixed reality. It's a special set of glasses capable of running without a connection to a PC or to a phone. They can project holograms over the existing environment, opening up a wide range of new scenarios and possibilities in the consumer and enterprise worlds. HoloLens is capable of running applications based on the Universal Windows Platform. In this case, developers will be able to leverage a specific set of APIs to create a new kind of experience by detecting, for example, the movement of the user's eye or hand gestures. If you want to know more, I suggest you visit the official YouTube channel of the HoloLens team.

Figure 2: The mixed reality experience made possible by HoloLens.

Windows 10: The beginning of a new journey

Windows 10 is also the beginning of a new journey, since it introduces a completely new lifecycle model. In the past, Microsoft has typically released a new version of the operating system every 3-4 years, with a series of service packs in between. These were mainly collections of patches and security fixes to make the computer more secure and to support new technologies that may have been released (like USB 3.0).

This approach doesn't fit anymore into the current digital world, where things change at a much faster pace. Every year we see new trends and new disruptive technologies, so 3-4 years is too long a time frame to stay up-to-date with all these changes.

As such, Windows 10 has introduced a new model called **Windows as a service**, which led many tech journalists to define it as "the last version of Windows." Technically, this definition is true, but not because Microsoft is going to quit the operating system business after the release of Windows 10, but because Windows 10 will be continuously evolving by adding new features over time.

The update frequency will be higher than the traditional service pack approach (approximately, Microsoft is going to release two updates per year) and each update won't bring just the usual patches and security updates (which will continue to be delivered regularly when needed), but also new features for users and new APIs for developers.

Each major update corresponds to a specific build number and to a specific SDK release. Let's see the versions released so far.

Windows 10 (build 10240)

The original version of Windows 10 was fully released to the public on July 29, 2015, and was identified by the build number 10240. This version was dedicated to traditional desktop and tablet devices: at the time of release, the mobile version of Windows 10 wasn't ready yet.

The goal of this first release was to make Windows 8 a more modern operating system, optimized also for touch devices, while still keeping the familiar Windows experience users had appreciated since Windows XP.

Here are some of the most important added or improved features.

The Start menu

Windows 8 introduced a completely revamped Start menu, inspired by the Windows Phone start screen and greatly optimized for touch devices, offering a full screen experience, live tiles, and much more.

However, many Windows users found it challenging to use, since it was a completely different approach than the one offered by its predecessors. Additionally, the full screen experience was forcing the user to continuously shift the focus from the task they were doing on the desktop to the Start menu whenever they needed to open a new app or perform another task.

Windows 10 introduced a new Start Menu offering the best of both worlds: a familiar visual experience, which takes just a small portion of the screen and displays an alphabetical list of all the applications installed on the computer, combined with an additional workspace for tiles, contacts, music playlists, etc.

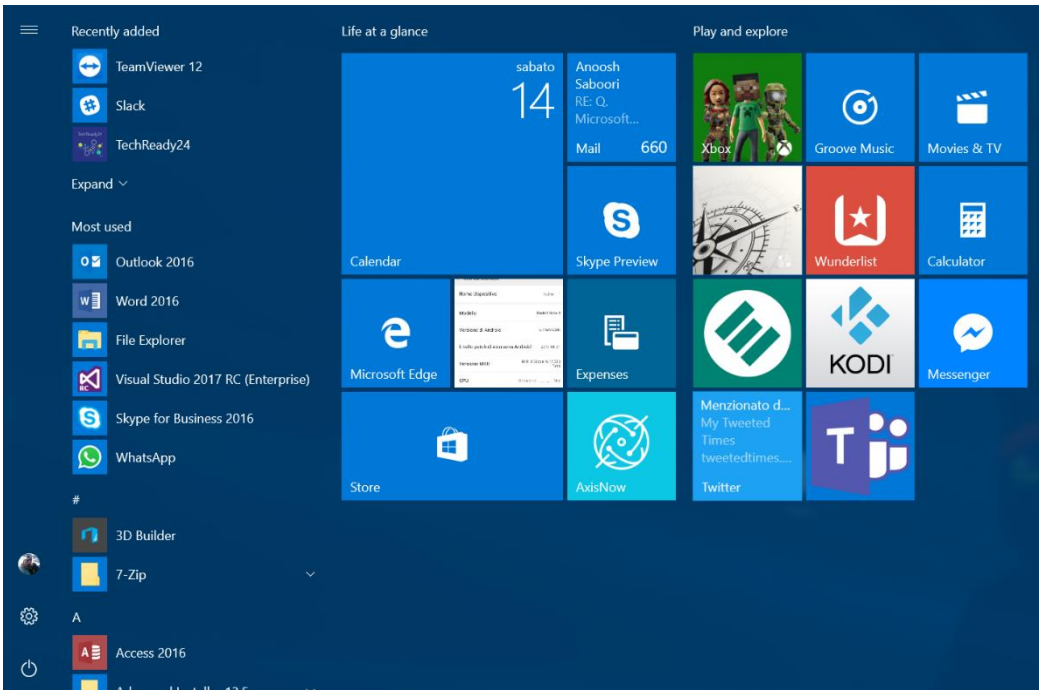


Figure 3: The new Start menu in Windows 10.

However, Windows 10 also offers a feature called **Tablet Mode**, which can be manually activated using an option in the Action Center or that some 2-in-1 devices (like the Surface) can automatically enable when they detect that the keyboard has been detached. When this mode is enabled, you get a fully optimized experience for touch devices, which is more like the one from Windows 8.1. Both the Start Screen and Universal Windows Platform apps are displayed in full screen, like in the following image.

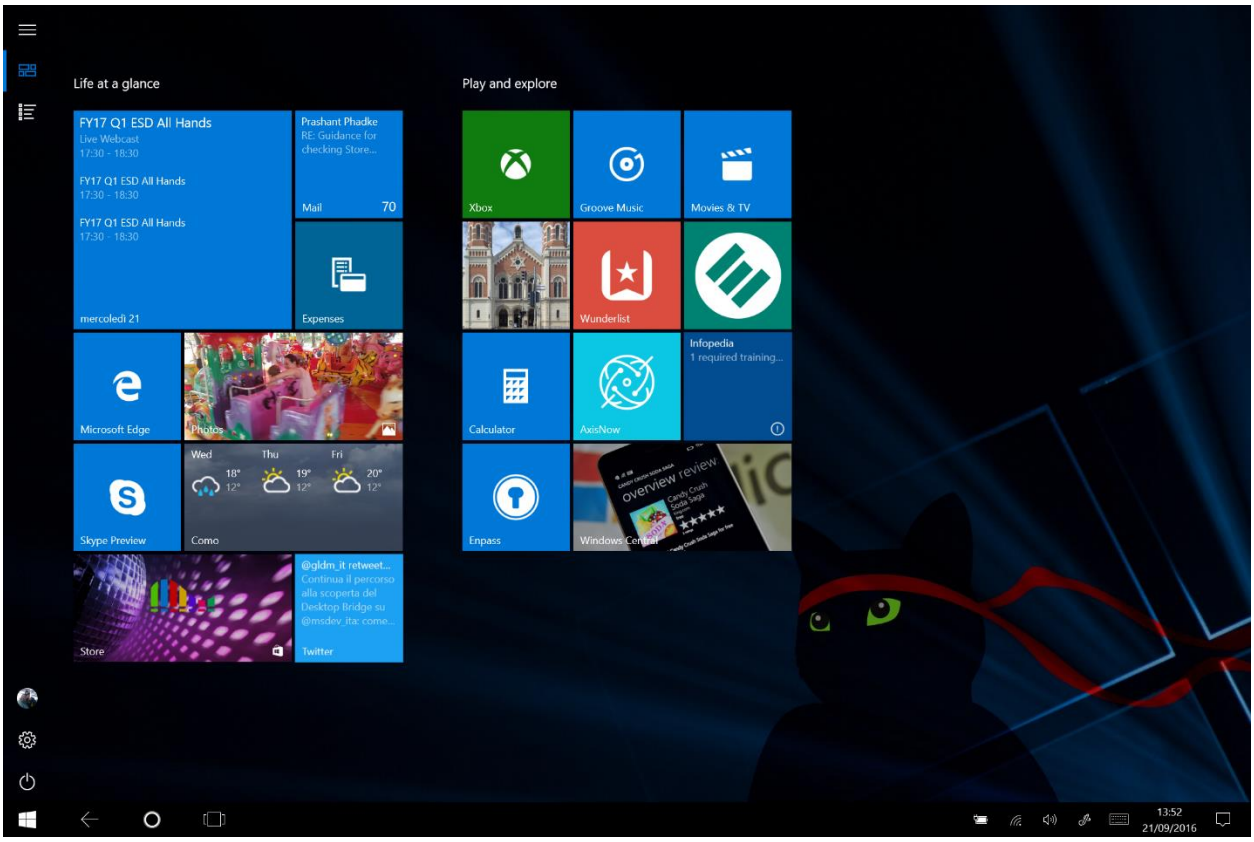


Figure 4: The Start Menu when Windows 10 is running in tablet mode.

Universal Windows Platform apps

Windows 8 already introduced the concept of Windows Store apps. They were built on top of the Windows Runtime and they could be downloaded from a centralized Store, making the whole installing/uninstalling/upgrading experience easier and more like the one we find on all the major mobile platforms.

However, on Windows 8 and 8.1, Windows Store apps had a downside: they could only run in full screen mode, forcing users to lose focus on a current task when they wanted to switch from a traditional desktop application (like Word or Photoshop) to a new, modern application. Because of this, these new applications were really appreciated by users with tablets or 2-in-1 devices (equipped with a touch screen), but they weren't used very often by owners of traditional computers with a mouse and keyboard.

Windows Store apps (called Universal Windows Platform apps) are now capable of running inside a window, exactly like every other traditional desktop application. As such, these modern applications can be used side-by-side with traditional ones, making them much easier to use on traditional desktop computers. Additionally, Universal Windows Platform apps that run on a desktop can leverage some features that are specific to the desktop world, like supporting the dragging and dropping of content inside a window.

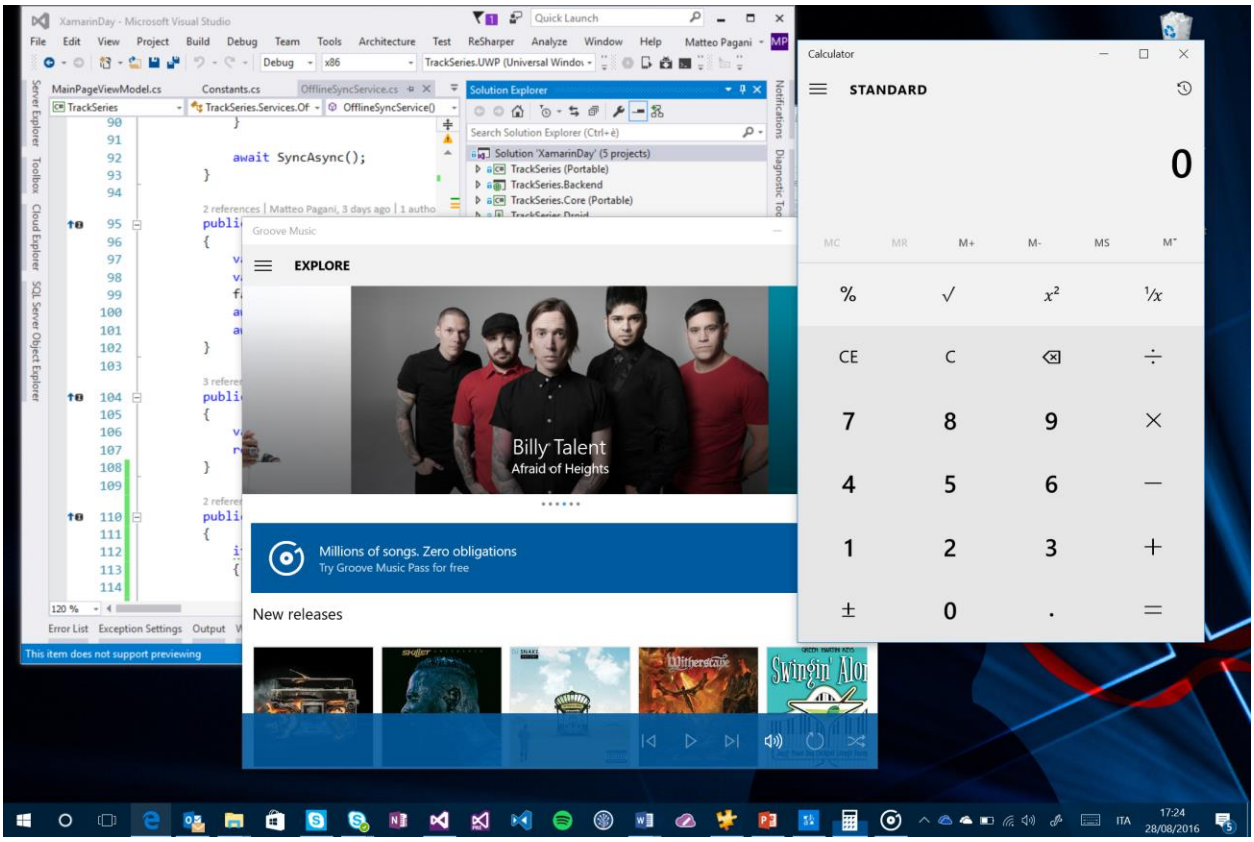


Figure 5: Visual Studio, which is a traditional desktop application, running side-by-side with two Universal Windows Platform apps: Calculator and Groove Music.

Action Center

This feature is inherited directly from Windows Phone 8.1 and it's known also as the notification center. Every time one of our applications sends us a notification (for example, when we receive a new email or a new mention on Twitter), it's stored in the Action Center so that it doesn't get lost because we weren't paying attention to the screen when we received it. Additionally, the Action Center contains a set of quick action buttons to quickly turn on or off features like Bluetooth, airplane mode, VPN settings, etc.

On a desktop, the Action Center is activated by clicking an icon on the bottom right corner of the screen or with a swipe from the right side of the screen on a touch-screen device; on mobile, it's displayed by swiping from the top to the bottom of the screen.

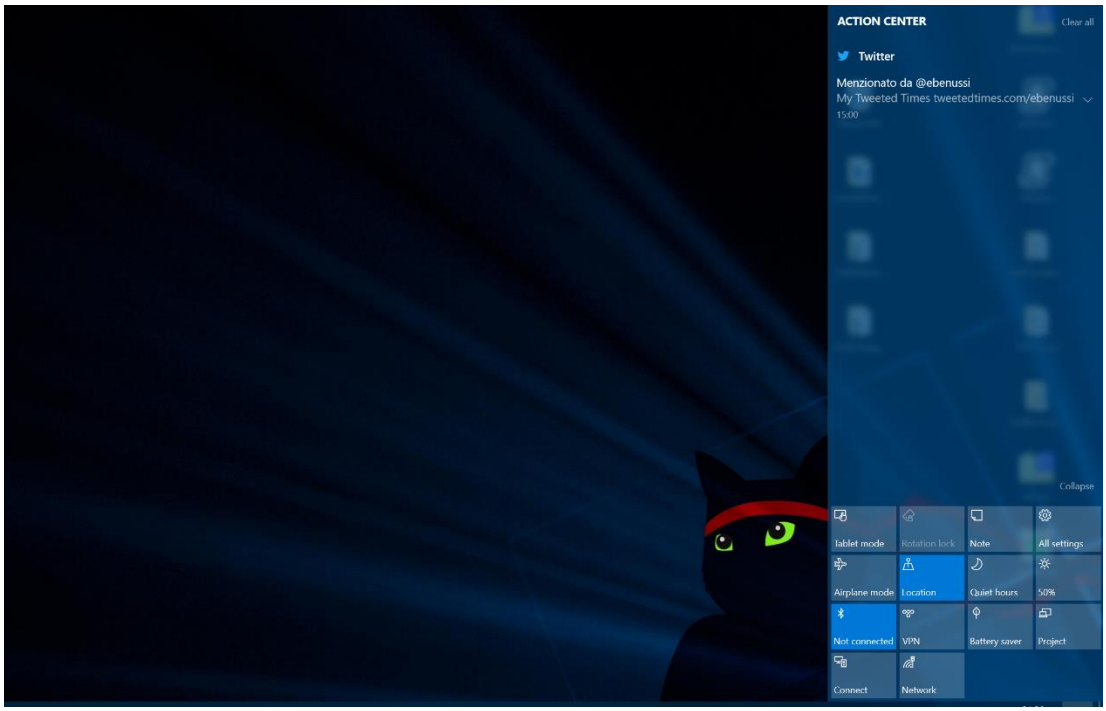


Figure 6: The Action Center in the desktop version of Windows 10.

Cortana

Cortana is another new feature introduced first in Windows Phone 8.1 and later expanded to Windows 10 across all devices. Cortana is a digital assistant that we can use to quickly get information about a topic (like how much time it will take to travel from home to the office), to set reminders based on time or location, to get notifications on topics that are relevant to us, to keep track of our appointments during the day, and much more. Everything can be achieved by simply typing our requests in the search box, but we can also leverage more natural ways like voice and inking. With the anniversary update, Cortana has been further expanded, allowing users to interact with her even when the computer is locked.

Cortana is also important for developers, since we have a set of APIs to integrate her into our applications, allowing users to perform tasks just by talking and without having to open an app or unlock their devices.

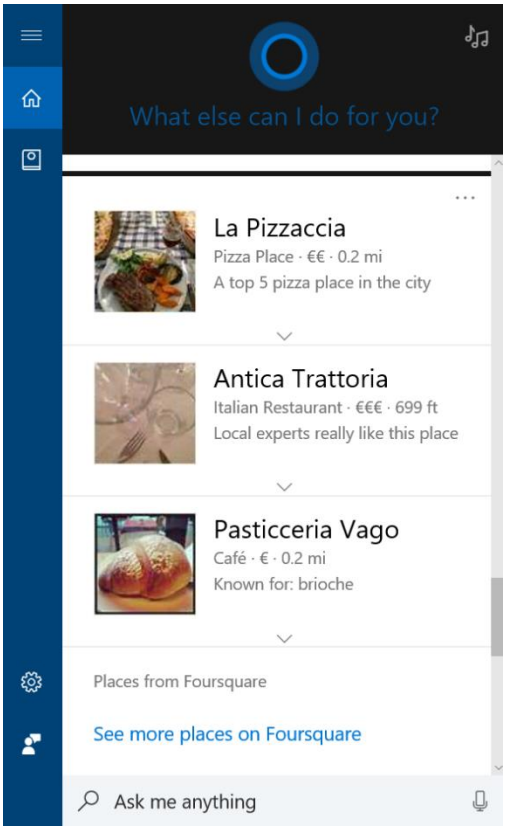


Figure 7: Cortana on Windows 10 is displaying information about the best restaurants near the current location.

Microsoft Edge

Microsoft Edge is a new browser that replaces Internet Explorer as the default browser in Windows 10. Edge has been built from scratch with the goals of supporting all modern web technologies and getting rid of all the legacy code and plugins that were often the cause of security and performance issues. The philosophy behind Edge is the same as the one behind Windows 10: the virtual world is moving fast, so releasing a new version of the browser every 2-3 years wasn't sustainable anymore, especially when the competition releases a new version of their browser every week.

Edge offers great performance, support to the most recent web standards, and, despite it using its own open-source engine (Chakra), it's fully compatible with all the features offered by other engines like WebKit (which is used by very popular browsers like Chrome and Safari).

In this case, Edge isn't just a consumer feature. Developers can also take advantage of its engine to create web-based applications or to integrate web features into their Universal Windows Platform apps.

Windows 10 November update (build 10586)

The November update, identified by build number 10586, was released in November 2015 and marked the first official release of the platform for mobile devices. It didn't add any major new consumer feature, but it continued to improve the Windows 10 experience by refining the Start menu and tablet mode, by expanding Cortana into new markets, and by adding a new SDK for developers with new APIs and features, like a new animation system.

Windows 10 Anniversary Update (build 14393)

The Anniversary Update was released one year after the original Windows 10 version and it's identified by build number 14393. This update expanded the device families that can be targeted by developers even more. This version, in fact, was also released for Xbox One, bringing some of the features of the other versions (like Cortana) and the Universal Windows Platform, and allowing apps to run on the game console. The only exception to cross-device publication is games. If you're a game developer, your game still needs to be approved by Microsoft by submitting it through the ID@Xbox program (you can find more info on the official website).

From a consumer point of view, the Anniversary Update has added some notable features, like extensions support for Edge, notifications synchronization among multiple cross-platform devices (not only Windows 10 Mobile, but Android is also supported), and new and improved built-in apps, like a new version of Skype. Inking support has been vastly improved, too, by a set of new features (both for developers and consumers) that make Windows 10 great to use with digital pens on supported devices, like the Surface family.

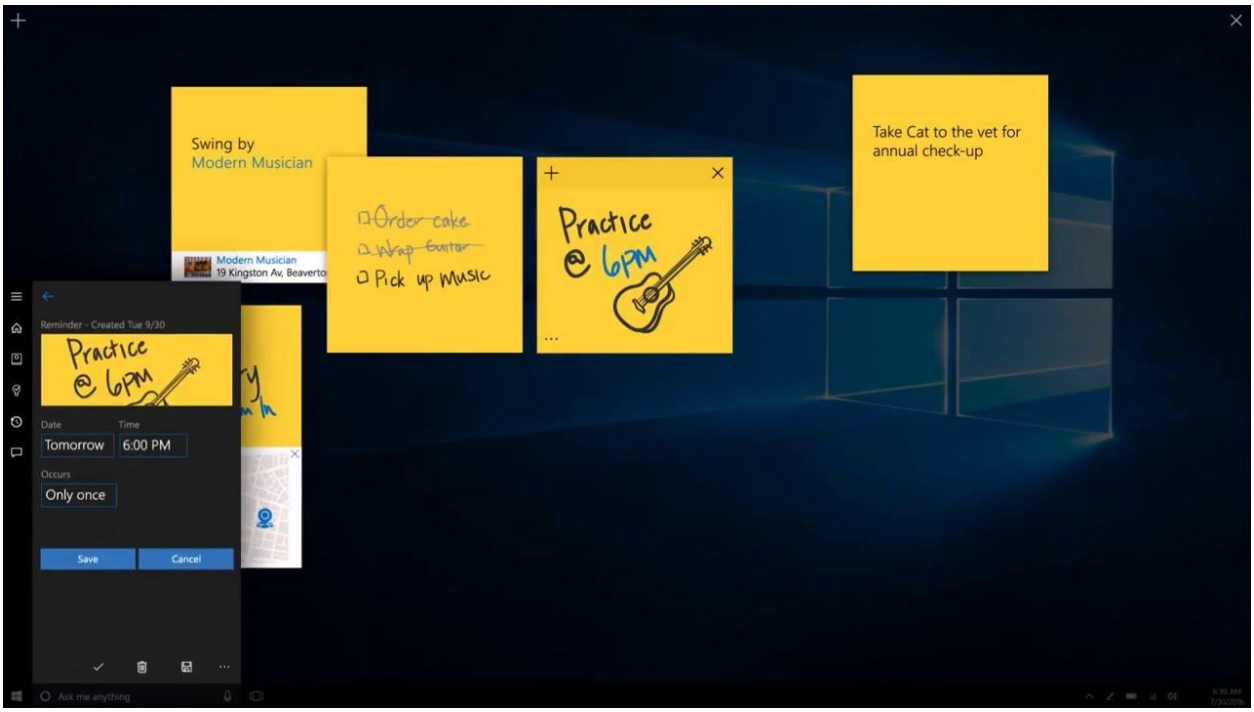


Figure 8: The Windows Ink Workspace on Windows 10 Anniversary Update.

From a developer point of view, other than the release of a new SDK with a new set of APIs, which will be discussed in detail in this book, it introduced two new important features:

- **The Windows Subsystem for Linux** is a user space developed by Ubuntu that runs natively inside Windows, allowing developers to use a native version of the BASH shell. This feature follows the open-source approach embraced by Microsoft, allowing developers familiar with UNIX tools to use them in a native way, without having to create a separate virtual machine with Linux. This feature, which is especially appreciated by web developers, helps to get the best of both worlds by allowing you, for example, to create a web project using a UNIX command line tool, but to edit it using a Windows tool like Visual Studio.

- **The Desktop Bridge**, previously known by the codename Project Centennial, is a series of tools that allow traditional Win32 apps (based on technologies like the .NET Framework, VB6, Java, etc.) to be packaged as UWP apps and to be distributed traditionally, but also on the Store, along with standard UWP applications. Thanks to Desktop Bridge, you'll be able to bring traditional desktop apps to the Store and, at the same time, start leveraging some of the UWP APIs and features in a Win32 app without having to rewrite it from scratch as a Universal Windows Platform app.

Windows 10 Creators Update (build 15063)

Windows 10 Creators Update is the latest version of the operating system, which started the official rollout on 11th April 2017. As the name says, the main target audience of this new version are creators: other than the usual improvements to the overall operating system (like a better support to high DPI devices; a night mode that makes the computer less fatiguing to use for the eyes during the night; a more consistent Settings experience, etc.), it has added a specific new set of application and tools dedicated to makers, designers and game players and game developers.

Some of the most interesting new features are:

- **Paint 3D**, which is a new version of one the most famous graphic application in the world. As the name says, Paint 3D can be used to create 3D models, that can be easily exported and reused with popular tools to work on three dimensional applications.
- **Windows Mixed Reality**: it's a new platform, both for consumers and developers, which brings together all the efforts to explore the new Mixed Reality ecosystem that, at first, was introduced by HoloLens and that now is being brought to the masses thanks to a new generation of devices, produced by Microsoft partners, that will make the holographic and immersive experiences merge together. Mixed reality blends real-world and virtual content to create compelling interactive experiences and, no matter if you're going to leverage an immersive experience or a holographic one, both are empowered by the same platform, allowing:
    - Developers to create three dimensional experiences that can be easily adapted and consumed with both categories of devices.
    - Users, to have a common virtual experience across every headset and mixed reality experience on Windows, no matter if it's a game or an educational application.

- **Gaming:** the Creators Update is the perfect operating system for gamers, by offering a set of features like Game DVR (to record a game session), Broadcasting (to stream a game session to other spectactors) or Game Mode (to focus all the hardware power of the machine on the game, so that it can run in the best possible way). Also developers can take advante of these gaming features, by offering the same game both on Xbox One and on PC; by adding a SDK to integrate Xbox Live features like achievements, leaderboard, etc.

This book is based on the Creators Update and, therefore, will cover most of the features and APIs introduced in the 15063 Build SDK.

Another important announcement related to this version of Windows is the launch of **Windows 10 S**, a special version of Windows 10 (based on the Creators Update) that, starting in Summer 2017, will ship on many computers and 2-in-1 devices (including a new device made by Microsoft called **Surface Laptop**). It's dedicated especially to the education world and to users that want a system as much safer and fast as possible, even after years of usage. Windows 10 S, in fact, supports acquiring applications only from the Windows Store, making almost impossible for a casual user to install adwares, malwares or classic desktop applications that can slow down the system. Desktop applications, however, will still be available and downloadable from the Store, thanks to the Desktop Bridge, the technology introduced and already mentioned in the description of Windows 10 Anniversary Update.

Windows 10 also introduced a big change in the pricing model. For the first time, a Windows version was offered as a free update to all the existing Microsoft customers. For a limited time of one year (from 29 July, 2016, to 29 July, 2017), Windows 10 has been made available for free through Windows Update to every existing Windows Vista, Windows 7, or Windows 8.1 user.

The Universal Windows Platform

As already mentioned, Windows 8 wasn't a break from the past just from the user experience point of view, but also for developer. It introduced the concept of Windows Store apps based on a new framework called Windows Runtime, instead of relying on the .NET Framework that developers had learned to use to create desktop and web apps a few years ago. It's a native runtime that is built on top of the Windows kernel and offers a set of APIs that apps can use to interact with the hardware and the operating system. It uses a similar approach to the old one based on COM (Component Object Model), created in 1993 with the goal of defining a single platform that could be accessed with different languages.

Windows Runtime uses a similar approach by introducing language projections, which are layers that are added on top of the runtime that allow developers to interact with the Windows Runtime using well-known and familiar languages, instead of forcing them to learn and use just C++. The available projections are:

- **XAML and C# or VB.NET**: this projection allows you to create applications using C# or VB.NET for the logic and XAML to define the layout. It is implemented with a special subset of the .NET Framework.

- **HTML and JavaScript**: this is the projection that will be appreciated most by web developers, since it allows them to use web technologies to create apps. The

application's layout is defined using HTML5, while the logic is based on a special library called WinJS, which grants access to the Windows Runtime APIs using JavaScript.

- **XAML and C++**: this projection can be used to create native applications using a C extension called C++ / CX that makes it easier for developers to interact with the Windows Runtime using C++.

- **C++ and Direct X**: this projection is especially useful for games, since it allows you to create native applications that make use of all the powerful features offered by the DirectX libraries to render 2-D and 3-D graphics.

The Windows Runtime libraries (called **Windows Runtime Components**) are described using special metadata files, which make it possible for developers to access the APIs using the specific syntax of the language they're using. This way, projections can also respect the language conventions and types, like uppercase if you use C# or camel case if you use JavaScript. Additionally, Windows Runtime components can be used across multiple languages. For example, a Windows Runtime component written in C++ can be used by an application developed in C# and XAML.

The Universal Windows Platform can be considered the successor of Windows Runtime and it's built on the same technology. In fact, if you already have some experience with the development of Windows Store apps for Windows 8.1 and Windows Phone 8.1, you'll find yourself at home. The Universal Windows Platform has added many new features and brought unification across all the platforms, but most of the APIs and features are the same as those you've learned to use with Windows Store apps development. This is a great news, because it helps keep the learning curve moving from Windows 8.1 to Windows 10 very gentle, unlike what happened in the shift between Windows Phone 8 with Silverlight and Windows Phone 8.1 with the Windows Runtime. In fact, at that time, the shift between the two technologies was quite radical and, despite sharing the same fundamental concepts, basically all the APIs and the XAML controls were different, forcing developers to rewrite most of the existing code base to target the new platform.

The most important feature of the Universal Windows Platform is that it offers a common set of APIs across every platform. Whether the app is running on a desktop, a phone, or a HoloLens, you're able to use the same APIs to reach the same goals. The following image shows where the Universal Windows Platforms fits in the overall Windows 10 architecture: every device, despite having a different shell, shares the same kernel, which is the Windows Core. On top of the core, the Universal Windows Platform provides a consistent set of APIs to access to all the functionalities that are exposed by the core, like network access, storage, sensors, etc. A Windows app runs on top of the Universal Windows Platform and is a single binary package, unlike with Windows 8.1, where you had to deal with one package for the phone and one for the PC.

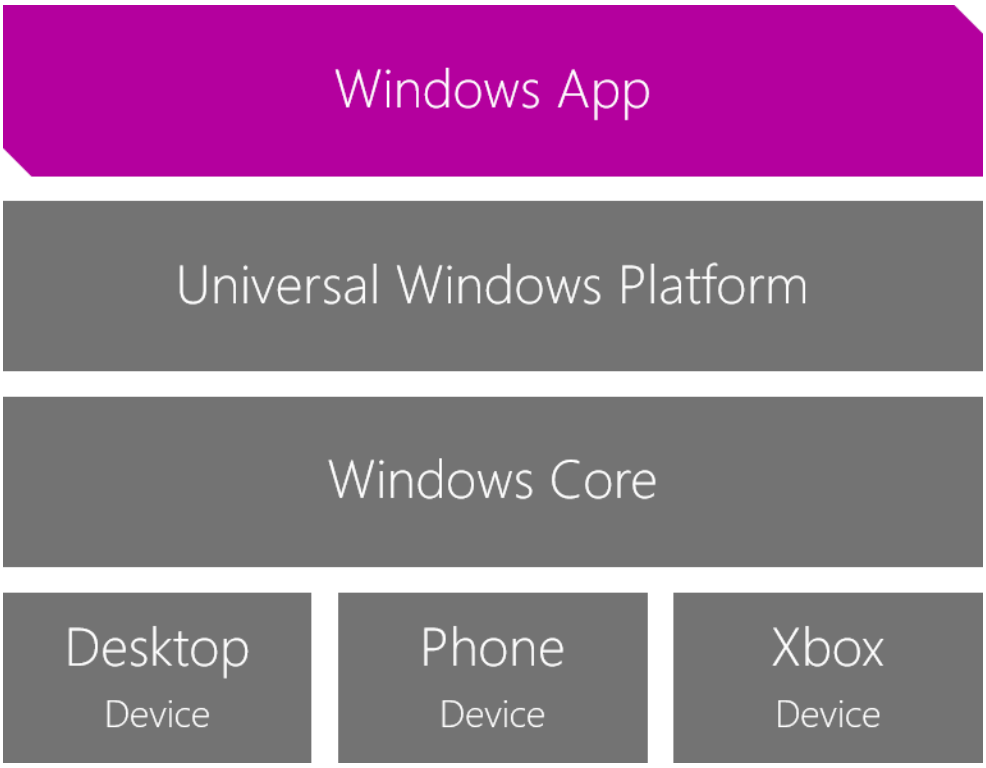


Figure 9: The Windows 10 ecosystem for developers.

Another important change in the Universal Windows Platform is that, when it comes to the C# and XAML project, it doesn't leverage the standard .NET Framework anymore, but a new, modern framework called .NET Core.

.NET Core can be considered the successor of .NET Framework. It has been built from scratch by following the new Microsoft principles: open source and cross platform. The adoption of .NET Core makes it easier to reuse the parts of the code base that don't have a specific dependency on the platform (like network communication) with other projects, no matter if it's a web application or a traditional desktop application. However, it's important to understand that this doesn't mean Universal Windows Platform apps can also run on competitive platforms like Android and iOS. The Universal Windows Platform, in fact, is built on top of .NET Core and it leverages its common base, but adds a set of APIs and features which are Windows specific.

The cross-platform nature of .NET Core, at the time of writing, can be leveraged especially by web developers. Thanks to this new framework, you'll be able to create web applications based on ASP.NET that are no longer forced to run only on a Windows server running IIS, but that can also be hosted on Linux and OS X.

This series of books is dedicated to exploring Universal Windows Platform apps development with C# and XAML. All the concepts and features that will be described in the next chapters can be leveraged by any supported language, but all the code samples published in the book will be in C# and XAML.

The Extension SDKs

Having a common set of APIs to target is great because it helps to maximize the code reuse across every platform. However, even if they share the same Windows Core, every device has some unique differentiators. For example, a desktop is typically controlled with a mouse and a keyboard, so a typical requirement of a desktop app is to support drag and drop; a phone, instead, can have a dedicated camera button to quickly take pictures, which is something that a PC or tablet usually doesn't have; a Raspberry Pi 3 offers an integrated circuit (called GPIO) where you can connect various sensors and devices, like LEDs, temperature sensors, or humidity sensors, but this is something that only an IoT device can offer and you won't find it on a PC or phone.

How does Windows 10 handle the fact that it has a common set of APIs for every platform (the Universal Windows Platform), but must still allow developers to take advantage of the specific features offered by each of them?

The answer is **extension SDKs**, which is a set of additional libraries that can be added in a UWP app directly through Visual Studio and that contain a set of APIs that are specific for each platform. As such, you will find a **Windows Desktop Extensions** library, a **Windows IoT Extensions** library, etc.

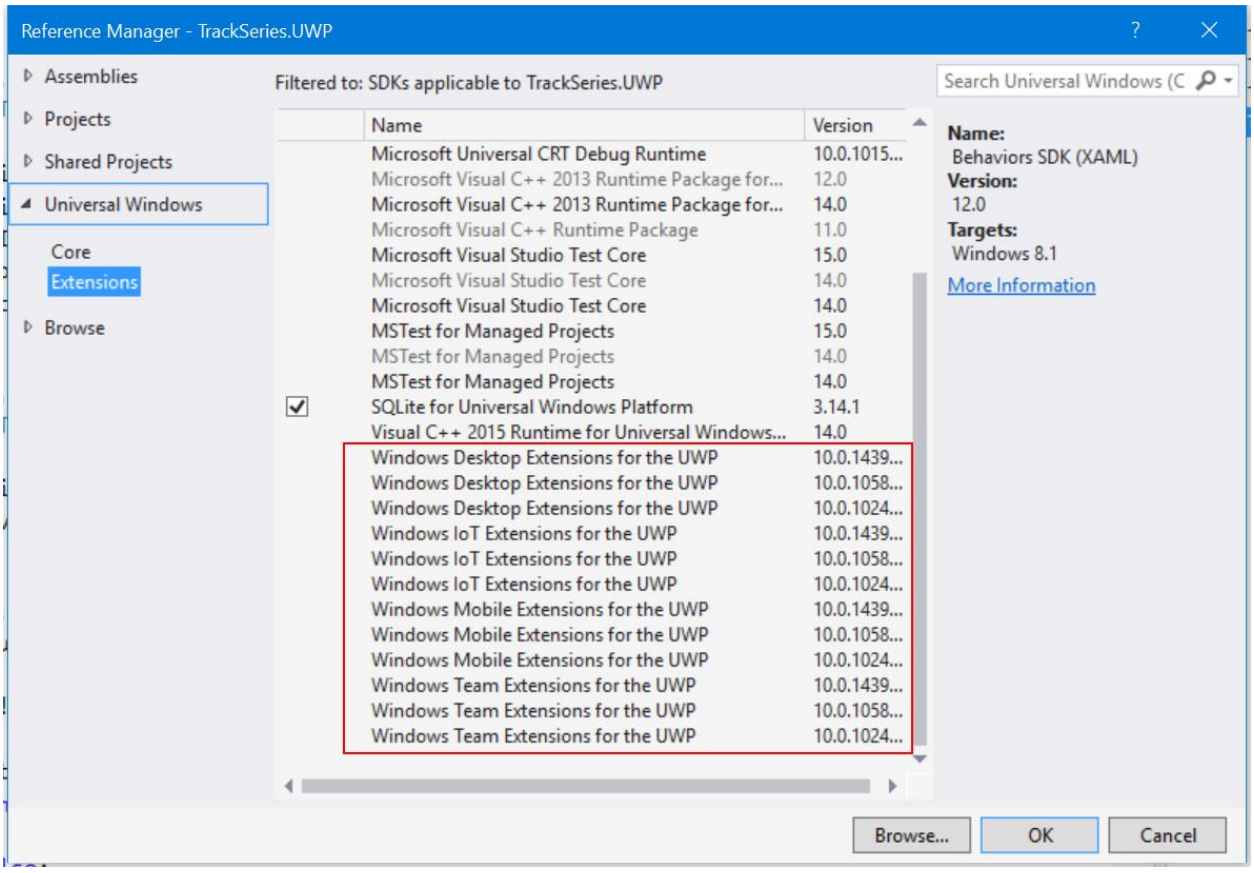


Figure 10: The various extension SDKs available for the Universal Windows Platform.

The APIs contained in these extensions are available in the base Universal Windows Platform. Consequently, even if you're going to add one or more of them to your project, you'll always be able to compile it without issues. However, on all the platforms that don't support these APIs, they aren't really implemented: you can think of them as stub APIs. They're declared, the various classes and methods are available, but they're not really implemented. Consequently, they will work only on the platforms for which they've been designed. If you try to execute some platform-specific code (like access to the camera button) on a platform which doesn't support it (like a desktop computer), you will get an exception at runtime.

To properly handle extensions SDKs, we need to introduce another concept that will be familiar to those of you who have some experience with web development: **capability detection**. In the beginning of the new web era, when technologies like JavaScript, HTML5, and CSS3 opened powerful new possibilities, developers started to leverage techniques like browser detection to test if it was possible to execute some code or use a specific HTML5 feature. However, this approach soon started to show its limits: browsers started to move very fast, with companies like Google and Firefox releasing a new version every week. It was impossible for a web developer to maintain a website that needed to check every available combination of browser and version. Consequently, the web has started to move from a version detection approach to a capability detection approach. Instead of detecting the browser's type and version, developers started to check if a specific capability (for example, video playback without plugins) was supported. This way, the website started to become independent from the browser. If a new version of Microsoft Edge or Google Chrome decided to implement a feature that the website was using, it would automatically start to work.

The same exact approach can now be applied to Windows 10. With the Windows as a Service model, it doesn't make sense anymore to check if the app is running on a specific build, because features and API implementation can quickly change between each version. As such, developers are now required to check, instead, if an API is properly implemented on the platform on which the app is running.

For example, let's say that you have created a powerful photo editing application that, only when it runs on a phone, should be able to leverage the physical camera button to quickly allow users to take a photo. In the old Windows world, we would have written some code like the following one:

Code Listing 1

```
Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
CameraButtonPressed;
```

The **Windows Mobile Extension for the UWP** offers a class called **`HardwareButtons`**, which is part of the **`Windows.Phone.UI.Input`** namespace, that allows you to handle the different buttons available on a phone. One of them is the camera button, so we can simply subscribe to the **`CameraPressed`** event and declare an event handler that will contain the code to acquire the photo when it's triggered.

However, if your app is running on a desktop or Xbox One, as soon as it starts, it will raise an exception like the following one:

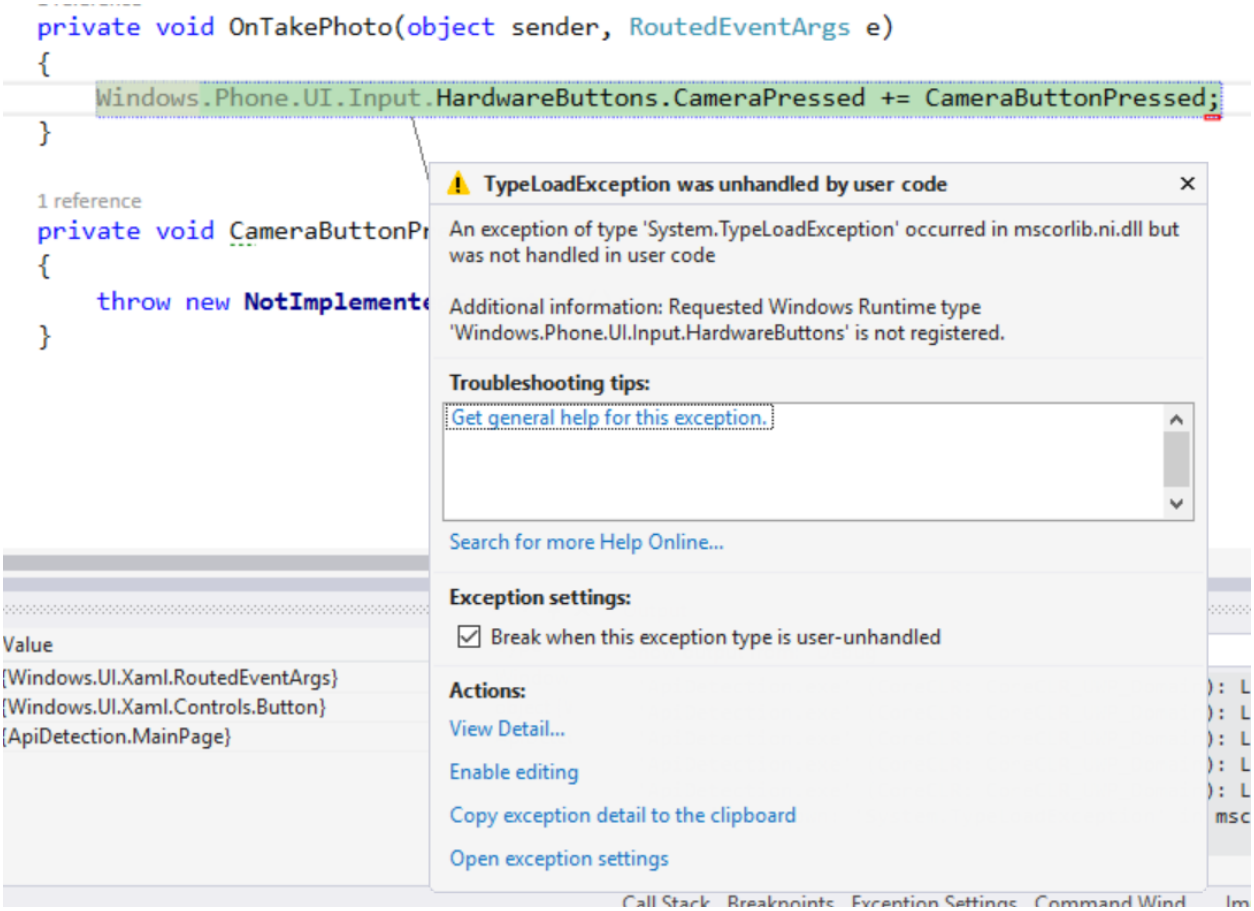


Figure 11: The exception raised when you try to use an API that isn't supported by the current device.

This is because the **HardwareButtons** class is available in the Universal Windows Platform (otherwise, your project won't be able to compile, since the binary package is universal), but it isn't implemented on the desktop.

This is where the capability detection concept comes in. The Universal Windows Platform includes a set of APIs that can be used to detect, at runtime, if an API is available and implemented. As such, before using a platform-specific API, you will always have to call these APIs to detect if the feature is really supported.

This API is called **ApiInformation** and it's part of the **Windows.Foundation.Metadata** namespace. It offers many methods to check various types of implementation and conditions. In the case of a simple API like the previous one, we need to use a method called **IsTypePresent()**, passing as parameter a string with the full namespace of the class we're trying to use. Here is how our previous code should look in a proper UWP app:

Code Listing 2

```
string api = "Windows.Phone.UI.Input.HardwareButtons";
if (Windows.Foundation.Metadata.ApiInformation.IsTypePresent(api))
{
    Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
CameraButtonPressed;
```

```
}
```

The method (like every other method offered by the **`ApiInformation`** class) returns a **`bool`** that will tell you if the feature you've requested is implemented or not. In the previous sample, we're going to subscribe to the **`CameraPressed`** event handler only if the app is running on a platform where the **`HardwareButtons`** class is implemented.

If you think again about the web scenario previously mentioned, it should be clear the advantage of this approach. At any point in time, if another Windows 10 platform should decide to start supporting an API, your application will be ready to handle it, without requiring you to change the code and release an update.

Handling the different versions of Windows 10

The Windows as a Service approach introduces a challenge that, as developers, we didn't have to face before. In the past, when we started to create an application, usually we targeted a specific version of Windows. If we created a Windows Phone 8.1 application, it couldn't run on Windows Phone 8.0. Additionally, we knew exactly which features and APIs we were allowed to use, because we were targeting a specific version of the operating system with its own SDK and set of APIs.

The new Windows 10 approach introduces, instead, a potential fragmentation issue. When we create an application, we target a specific Windows 10 SDK, but we don't know in advance which version of Windows 10 the users will have on their devices.

Also in this case, exactly like we did with extensions SDKs, we can leverage capability detection to solve this problem. When you create a new Universal Windows Platform app in Visual Studio, this is the message dialog you're going to see:

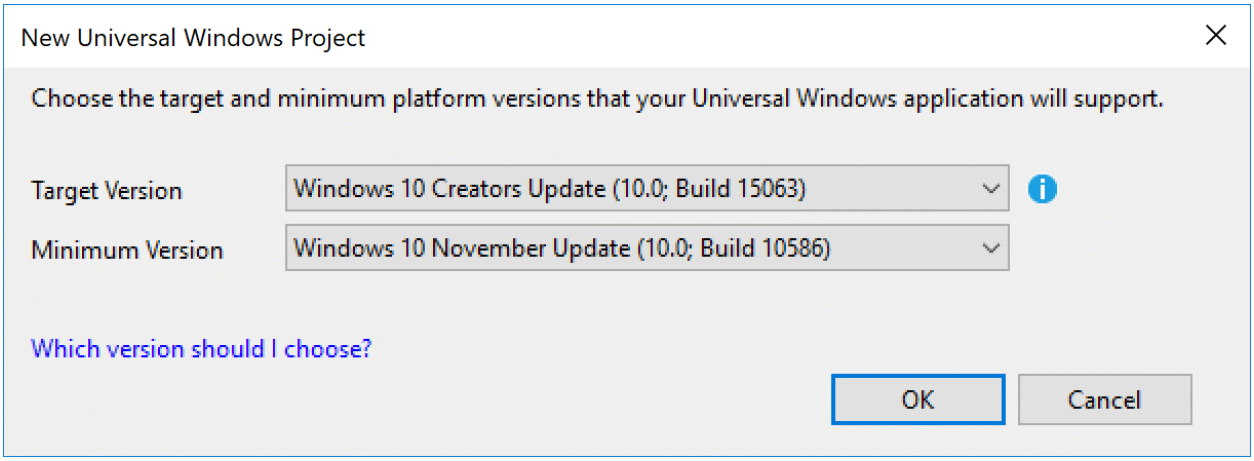


Figure 12: The dialog displayed when you create a new Universal Windows Platform app in Visual Studio.

As you can see, you can set a **Minimum Version** and a **Target Version** for your application. The first parameter defines the minimum Windows 10 version supported by your app. If the user has a build older than the minimum one, they won't be able to install the application at all. The

second parameter, instead, defines the set of APIs that your app can leverage. If the app is running on a device with the same (or a newer) version of the target SDK, it will be able to leverage all the features included in the SDK.

The previous image shows the default configuration when you create a new project in Visual Studio 2017 with the Creators Update SDK installed:

- The minimum version is set to Windows 10 build 15063 (Creators Update).
- The target version is set to Windows 10 build 10586 (November Update).

This means that the application can run on devices with the November Update, but if it's running on a device already using the Creators Update, you can take advantages of the new APIs and features included in this version. From a code point of view, you can handle this scenario in the same way we did before with Extension SDKs. You will use the **`ApiInformation`** class before using a feature that is available only on the Creators Update, otherwise people who are still running the November Update will face a crash as soon as they try to use this feature.

For example, the following sample code shows how to leverage a feature connected to animations that has been added in the Anniversary Update, but avoid the app crashing when it's running on an older device.

Code Listing 3

```
if
(ApiInformation.IsTypePresent("Windows.UI.Xaml.Media.Animation.ConnectedAni
mationService"))
{
    ConnectedAnimationService cas =
ConnectedAnimationService.GetForCurrentView();
    cas.PrepareToAnimate("ImageSeries", imgSeries);
}
```

Additionally, to mitigate the fragmentation issue, Microsoft has adopted a new update strategy. In the past, Windows users could disable Windows Update and decide not to keep their computer up-to-date with the latest patches and fixes. This approach has always been discouraged by Microsoft, especially for security reasons. In a world where malicious developers keep trying to steal data from users or to invade their privacy, it's critical to keep the operating system always up-to-date with all the latest patches.

With Windows 10, only the enterprise users (who have installed Windows 10 Enterprise) can control the update cadence, because we are typically talking about a Windows edition that is usually installed in complex environments like big enterprise companies, with many legacies but critical apps that need to be tested against every Windows update to make sure that everything continues to run just fine. Windows 10 Home and Pro users can defer updates, but just for a short period of time; they can choose the date and time that best fits their needs (to avoid the update process starting in the middle of an important task); however, they can't decide not to install a Windows 10 update. As such, it may take a while a before every user is aligned on the same version (since, typically, Microsoft deploys Windows 10 updates with a gradual rollout, to minimize potential deployment issues), but, at some point, every Windows 10 user will be on the

same version. This approach gives developers the confidence that, in a short amount of time after the release of the new update, they'll be able to leverage all the new features and APIs in their apps.

.NET Native

Another important feature introduced in the Universal Windows Platform is .NET Native. During the old Windows Phone times, the application's build process was the typical one offered by the .NET Framework, which leveraged just-in-time compilation. This means that the Visual Studio output was a package containing intermediate code, which was converted into native code at the first execution of the app directly on the device (in the initial versions of the platform) or on the cloud (with the Windows Phone 8.1 release).

.NET Native is a new technology that allows you to compile your Universal Windows Platform app directly in native code, introducing many advantages:

- Better performance
- Quicker start-up times
- Smaller memory footprint

The downside of .NET Native is that it makes the debugging experience more complex, because the code isn't managed anymore and, as such, stack traces are much harder to decipher. Additionally, native compilation requires longer build times than managed compilation. Therefore, by default, when a project is set in Debug mode in Visual Studio, it will be compiled in the traditional way, giving you the usual great debugging experience and fast building times. .NET Native will kick in only when you're going to perform the compilation in Release mode, which typically happens when you're ready to release the application on the Store or to distribute it in your enterprise environment.

However, there's an important aspect of .NET Native to keep in mind. If you have some previous experience with Windows Store apps development, you'll remember that you could build your application using as target platform **Any CPU**. With this configuration, you ended up with a package that was cross compiled for multiple architectures and could run on any device, whether it was based on a x86, x64, or ARM architecture.

This is a specific feature allowed by the managed architecture. Native code, instead, can't be cross compiled. Because of this, in Universal Windows Platform apps, you won't find the **Any CPU** platform in the project's configuration anymore, but you'll have to choose the correct platform based on where you are deploying your application (for example, x86 if you're testing it on a desktop or ARM if you're launching it on a phone or on a Raspberry Pi).

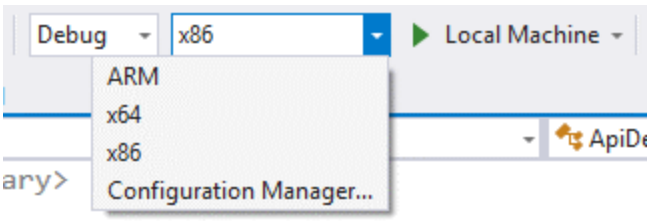


Figure 13: The AnyCPU configuration isn't supported anymore by Universal Windows Platform apps.

However, as we're going to see in last part of this book's series, this requirement doesn't change the way we're going to distribute our application on the Store. Thanks to a feature called **package bundle**, we'll be able to create a single package that will include the application compiled against each architecture, and then it will be up to the Store to distribute to the user the proper version based on the user's device.

The development tools

The tool to develop Universal Windows Platform apps, which has already been mentioned multiple times, is **Visual Studio.** The most recente version is **Visual Studio 2017**, which is the version you need if you want to target the Creators Update SDK. Visual Studio 2015 too supports the development of Universal Windows Platform apps, but the maximum supported SDK is the Anniversary Update (14393) one. Both Visual Studio 2015 and Visual Studio 2017 are available in different versions:

- The **Community Edition**, which can be downloaded from http://www.visualstudio.com/en-us/products/visual-studio-community-vs. It's a completely free version with the same features offered by the Professional one and can be used by individual developers, students, open-source project developers, and small companies.

- Two versions dedicated to professional developers and companies, called **Professional** and **Enterprise**, which can be purchased individually or through an MSDN subscription. These give you access not just to Visual Studio but also to the whole range of Microsoft products for development and testing purposes, plus many more benefits (like monthly credit on Azure, a free token to create a developer account for the Store, etc.)

However, regarding the Universal Windows Platform apps development experience, you won't find any difference or limitation in the Community version. You'll be able to create and publish apps in the same way you can with a professional Visual Studio version. If you're a student, you can also check the DreamSpark program, which allows you to get all the Microsoft professional tools for free.

Since this book is about the latest version of Windows 10, the Creators Update, we will use as a reference the most recent version of Visual Studio, which is the 2017 edition. This new version is a big step forward compared to the previous versions, thanks to a more lightweight installer (which easily allows a developer to install only the tools he needs) and a more agile development approach (which means more frequent updates, in order to fix more quickly the bugs identified by the community).

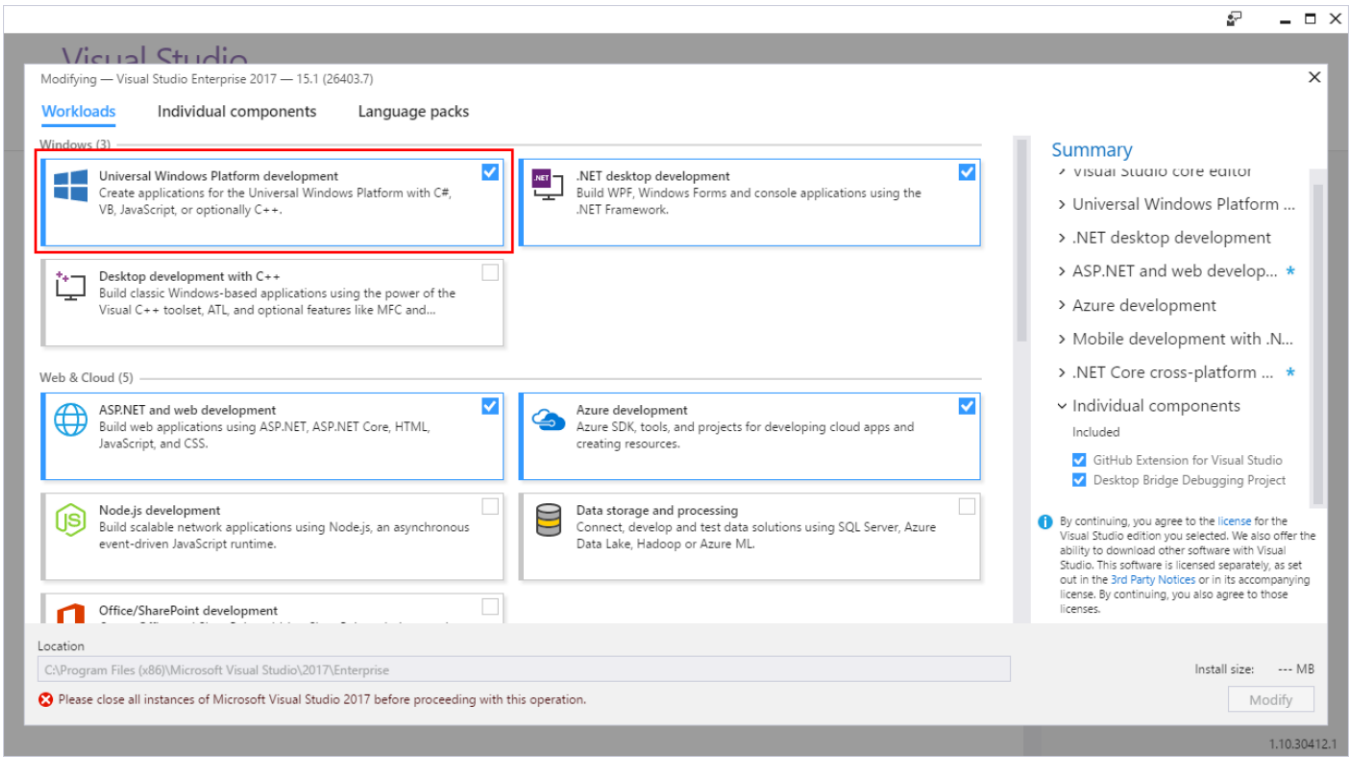


Figure 14: Make sure to check the Universal Windows App development tools in the Visual Studio 2017 setup to install everything you need to start developing UWP apps.

Of course, the best environment to start developing Universal Windows Platform apps in is Windows 10. However, the Universal Windows App Development Tools can also be installed on Windows 8.1 or Windows 7, with the following limitations:

- On Windows 8.1, the XAML designer doesn't work and you must deploy the app on a remote device (like a Windows 10 tablet or a Raspberry PI 3), on a Windows 10 Mobile phone connected through the USB cable, or on the Windows 10 Mobile emulator. You can't launch the app locally, since Windows 8.1 doesn't include the Universal Windows Platform.

- On Windows 7, in addition to the same limitations you have on Windows 8.1, you can't use the Windows 10 Mobile emulator, since it's based on a technology (Hyper-V) that was introduced for the first time on a consumer version of Windows in Windows 8.1.

The minimum hardware requirements are a computer with a 1.6 GHz processor, 1 GB of RAM, and at least 4 GB of free space on the hard disk. If you are planning to use the Windows Mobile emulators, however, you will also need:

- A Professional or Enterprise Windows 10 64-bit version.

- A processor able to support SLAT, which is a hardware chip required by Hyper-V, the Microsoft virtualization technology developed by Microsoft used to run emulators. The post published at http://blogs.msdn.com/b/devfish/archive/2012/11/06/are-you-slat-compatible-wp8-sdk-tip-01.aspx will show you how to detect if your CPU supports this feature.

- At least 4 GB of RAM (8 GB are suggested if you're planning to run multiple emulators at the same time).

Additionally, Microsoft has released a special version of Visual Studio 2017 called **Visual Studio Preview**, which contains all the new features and extensions that are still in testing phase. This version can be installed side by side with the regular version, allowing you to test the new features and, at the same time, to keep your development environment for production clean and stable. You can download the Preview version from https://www.visualstudio.com/vs/preview/

Testing your apps

Assuming that you are following the easiest path to configure your development environment, which is installing Visual Studio 2017 directly on a Windows 10 machine, there are multiple ways to test your applications, based on the target device.

Traditional desktop or tablets

To test your app on a traditional desktop or tablet, the easiest option is simply to launch the debugger on the same machine Visual Studio is installed on. To use this approach, the computer needs to be unlocked for development, which can be achieved simply by opening the Windows settings, choosing **Updates & Security**, and enabling the **Developer mode** option under the section **For developers**. After this step, you can just choose **Local machine** as target in the debugger drop-down in Visual Studio to launch your application locally.

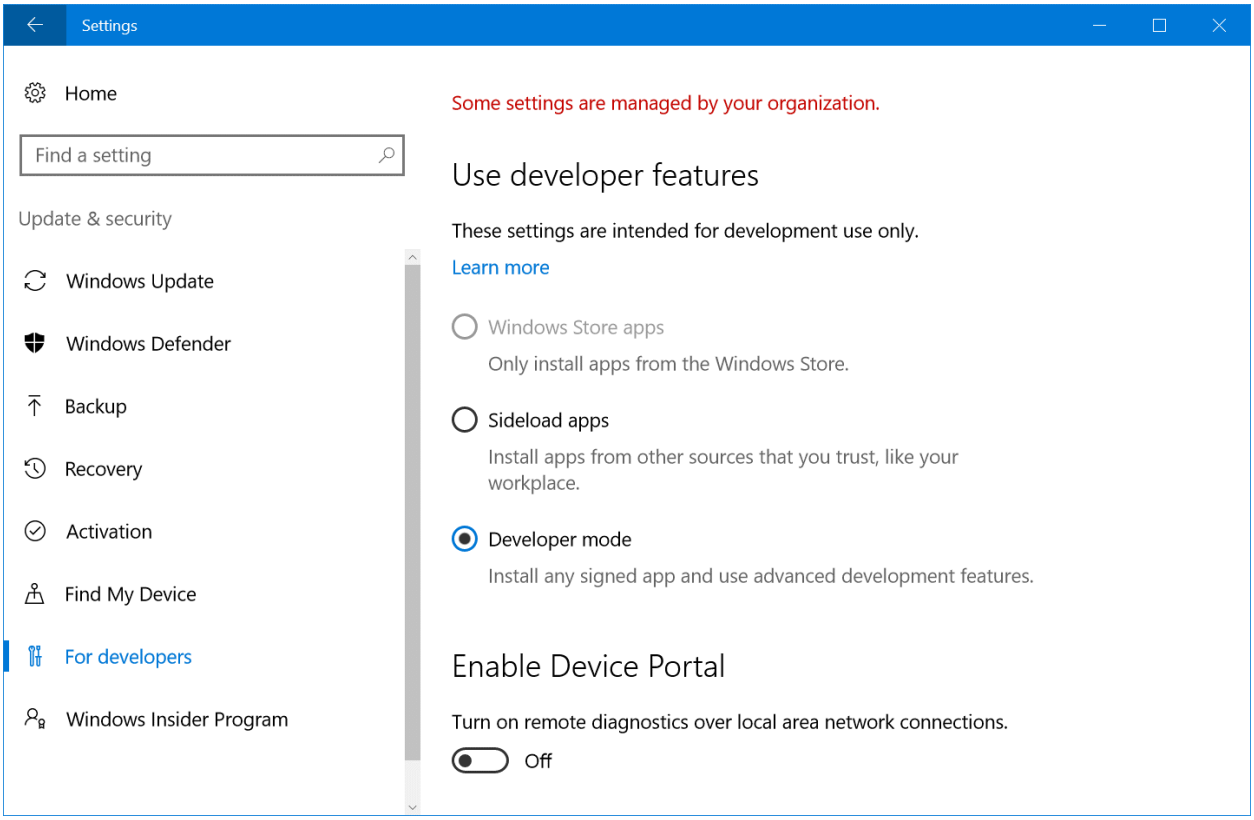


Figure 15: How to enable the Developer mode in Windows 10.

In case you want to test some features that your development machine doesn't have (for example, if you don't have a touch-enabled device like the Surface Pro), you can install the app on another device (like a Windows 10 tablet) and use the **Remote Tools for Visual Studio 2017**, which can be downloaded from https://www.visualstudio.com/downloads/ under the **Tools for Visual Studio 2017** section. Once you have installed it, you'll be able to connect to this device from your development machine. The only requirement is that they should be on the same network. To use this approach, you'll have to choose the option **Remote machine** as target in the debugger drop-down. You'll be asked some information about the target device, like its IP address or network name.

Another option is to use the **Simulator**, which is a special application that will launch a remote desktop session against your existing Windows installation. The advantage of the simulator is that it offers a set of tools to simulate scenarios that can be hard to test with a physical machine, like network speed, geolocation, or different resolutions and scaling factors.

Smartphones

If you want to test the mobile version of your app, you can use a physical Windows 10 Mobile device, like a Lumia 950 or a Lumia 550. You will have to first unlock the device for development. Since Windows 10 shares the same core and the same user experience across every platform, you can follow the same steps described for the desktop to enable the Developer mode on a mobile phone.

Deployment and debugging is supported only by connecting the mobile phone to the development machine with a USB cable; then, you can choose **Device** from the debugger drop-down in Visual Studio 2015.

Another option, if your machine supports it (see the requirements in the previous section), is to use the Windows 10 Mobile emulator, which supports a set of additional features that make some scenarios easier to test. For example, you can fake the device's location (to test apps that make use of geolocation services), you can simulate incoming notifications or an SD card, you can test different network conditions, etc.

To deploy and debug your application on a Windows 10 Mobile emulator, you can choose one of the available images in the debugger drop-down. There are multiple versions of the emulator to simulate different screen sizes and hardware features (like high-resolution screens or low-memory devices).

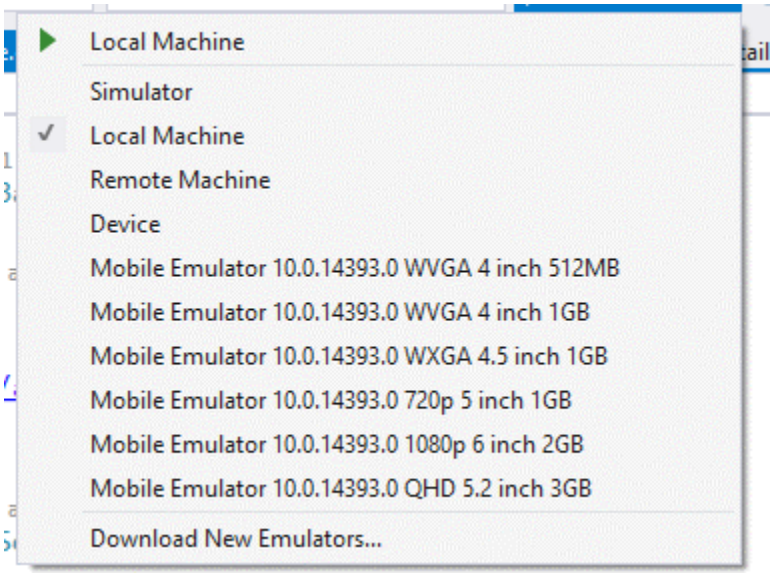


Figure 16: The various emulators available to test your apps on a Windows 10 Mobile device.

Windows 10 IoT Core

Windows 10 IoT Core comes with a special version of the **Remote Tools for Visual Studio 2017** already installed, which is started by default when you boot the device. As such, to deploy and debug a UWP app on Windows 10 IoT Core, you must choose the **Remote machine** target in the debugger drop-down menu and specify the IP address or network name of your IoT device, which needs to be connected to the same network of the development machine.

HoloLens

From a testing point of view, HoloLens is a mix of all the other testing modes we've seen so far:

- You can deploy your application directly on the device by connecting it to the PC using a USB cable. In this case, you choose **Device** as target, like you would do with a Windows 10 Mobile phone.

- You can deploy your application remotely by choosing as target **Remote machine** and specifying the IP address of the HoloLens.

- You can also test them against an emulator. However, unlike the mobile emulator, the HoloLens one isn't automatically installed with Visual Studio, so you'll have to install it separately from https://developer.microsoft.com/en-us/windows/holographic/install_the_tools.

Xbox One

The developer mode isn't built-in in the console, but requires a separate app to be installed and launched directly on the console. The app is called **Dev Mode Activation** and you'll have to search for it on the Xbox One Store. Once you have installed and launched it for the first time, the app will give you a code, like in the following image:

Figure 17: The code to activate the development mode on Xbox One.

Once you have the code, you can open your browser on your PC and head to the URL: https://developer.microsoft.com/xboxactivate. It will take you to a specific page of the Dev Center (the portal where you submit and manage your applications we'll talk about it in the second part of the book) where you insert the code.

At this point, your Xbox One will be unlocked and, by using the Dev Mode Activation app, you'll be able to switch your console from the standard retail mode (where you can play games, use apps, chat with your friends, etc.) to the dev mode, which will launch the **Remote Tools for Visual Studio 2017.** After that, the procedure to debug your UWP app on the console is the same we've seen, for example, with an IoT core device. You'll have to choose the option **Remote machine** from the debugging drop-down and specify the IP address of your Xbox One (which will be displayed directly, together with other info, on your TV screen once your console is in dev mode).

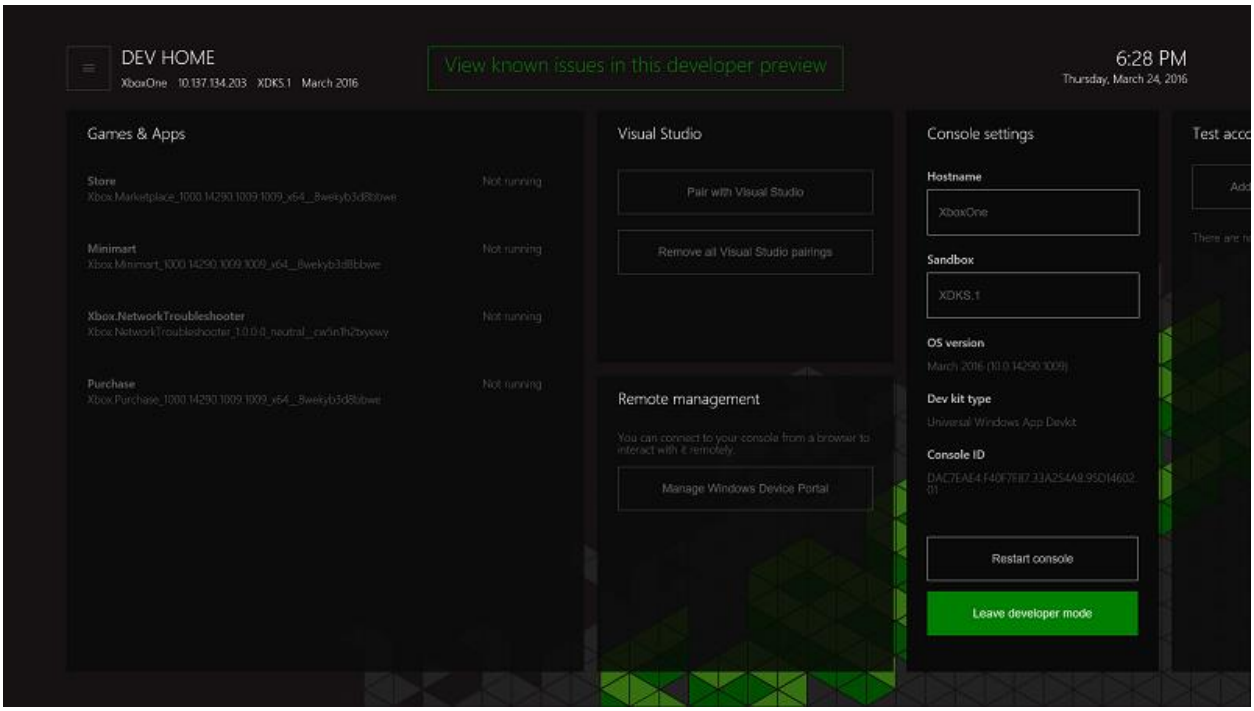


Figure 18: The Development Mode home on Xbox One.

Surface Hub

Testing and debugging apps on Surface Hub works like on an IoT Core device. Also, Surface Hub comes equipped with the **Remote Tools for Visual Studio 2017**, which are automatically launched when the device boots. As such, you must choose the **Remote machine** option from the debugging drop-down and specify the IP address of the Surface Hub.

Using NuGet

NuGet is a popular package manager that simplifies the developer's life. When it comes to adding a third-party library to our project, instead of searching the web and then downloading and manually adding a reference in our project, we can use NuGet to perform all these operations for us. NuGet will take care of downloading the library, adding a reference to the proper DLL (or DLLs), and setting up the project with all the required files and configurations.

NuGet is becoming more and more part of the integrated development experience in Microsoft, with the goal being to reduce the dependencies of a platform from its libraries and tools. A NuGet package, in fact, can be updated more frequently and independently from Windows and Visual Studio. For example, the base .NET Core version that the Universal Windows Platform is based on is a NuGet package itself that the Windows team can update independently from Windows 10 and Visual Studio. It's likely, in fact, that when you create a new Universal Windows Platform project, you will immediately find an update for the package **Microsoft.NETCore.UniversalWindowsPlatform** to apply (which is highly recommended, since it will allow you the benefit of all the improvements that have been made to .NET Core).

We're going to use NuGet in many scenarios in these books to install third party libraries that can simplify our job. Using NuGet is simple: right-click on your project and choose the **Manage NuGet packages** option. You will a see the NuGet main window, which you can use to find new packages on the core repository, uninstall an already installed package, or update an existing one. You can also access an option called **Manage NuGet packages for Solution** by right-clicking on the solution. In this case, you'll be able to manage the packages that are installed on every single project.

To add a package, it's enough to search for it by name or by a keyword in the **Browse** tab. After you've found it, just click the **Install** button displayed near the package description. NuGet will take care of everything for you.

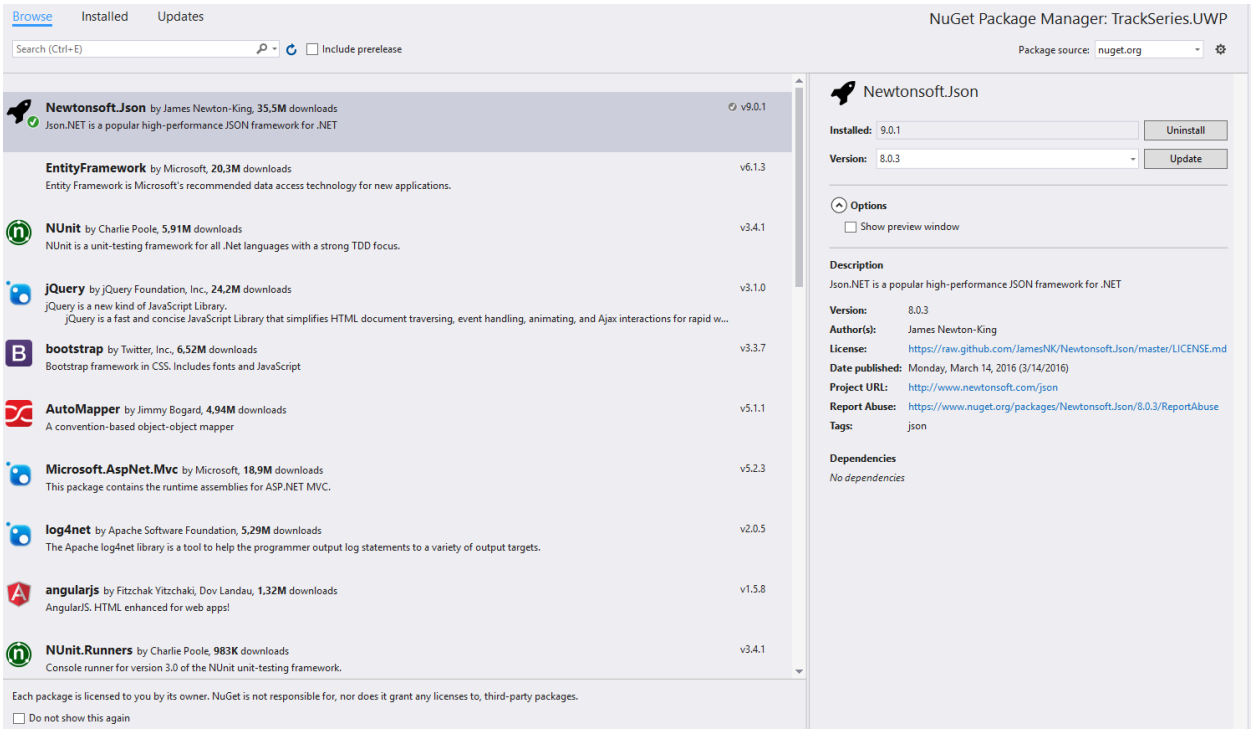


Figure 19: The NuGet window to handle the libraries installed in a Visual Studio project.

The package manager has also two additional tabs: **Installed** will list all the packages that have been installed to the current project or solution, while **Updates** will display available versions newer than the one currently installed.

The Windows Insider program

With Windows 10, Microsoft also started a new approach to getting feedback about the new releases of the operating system by launching the Windows Insider program, which is available for many platforms (the two most frequently updated ones are for desktop and mobile, but you will find Insider builds also for IoT Core).

When you subscribe your device to the Windows Insider program, you have the chance to get early builds of the new versions of the operating system and test all the upcoming features

before the final the release. This program targets not only consumers, but also developers. Developers subscribed to it, in fact, have the chance to get early access to preliminary versions of the new SDKs, to start experimenting with new APIs and scenarios.

The program helps enthusiastic users and developers stay always up-to-date on the latest features and announcements, but it also helps Microsoft to shape Windows 10 in a better way. Many decisions made by the engineering teams during the development of various Windows 10 updates are a direct consequence of the feedback reported by the Insider users through the **Feedback Hub**, a built-in application that, previously, was available only in the Insider builds but, starting from the Anniversary Update, can be leveraged by every user to share feedback about the platform.

To subscribe to the Insider program, it's enough to open the Settings on your device and, in the **Updates & Security** section, choose the **Windows Insider Program** tab. From there, you'll be able to connect your Microsoft account to the program, turning your Windows installation into a Windows Insider machine. The Windows Insider Program offers three different rings, which you can select from in a drop-down menu in the same branch:

- **Fast Ring:** with this ring, you'll receive a new build approximately every week. You'll be the first to test new features as soon as they are introduced, but you'll also more likely encounter some issues, since these builds can be very preliminary and not fully tested.

- **Slow Ring**: with this ring, you'll receive a new build approximately every month. It will take a while (compared to the fast ring) before you'll start to see new features, but the average quality of these builds will be higher, since they've gone across multiple test cycles and they've been released in the fast ring for a while.

- **Release Preview**: with this ring, you won't receive new major versions, but only cumulative updates for your current version, usually a couple of weeks before they're released to all other users.

The main difference between the first two rings and the Release Preview ring is that:

- Fast and slow rings release Windows 10 versions with a completely new build number, which means that they will trigger a full upgrade experience. For example, at the time of writing, the fast ring contains the first releases of the next Windows 10 version, which name, announced during the BUILD conference, will be **Fall Creators Update**. The build number of these releases starts from 16100.

- Release preview ring contains only updates that change the minor revision of the build, which on PC are applied as regular patches and they just require a reboot. On the phone, instead, the update procedure is always the same, regardless of whether it's a major or minor build. For example, at the at the time of writing, the release preview ring contains build 15063.**296** (as you'll notice, the major revision number is still 15063, the one related to the official Creators Update). The number that changes refers to the minor revision, which I have highlighted in bold.

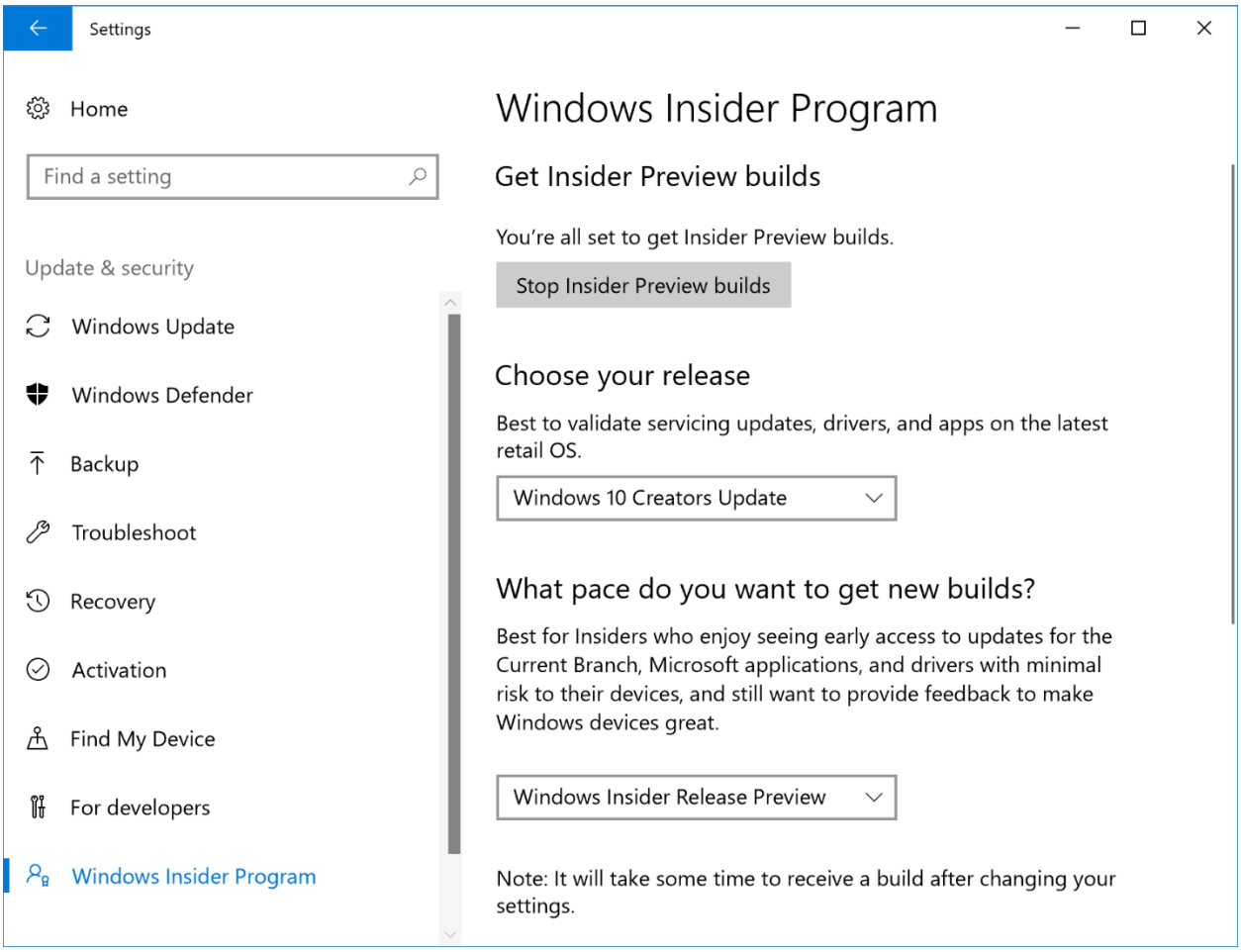


Figure 20: The Windows Insider Program page in the Windows settings.

The future of the Universal Windows Platform

During the BUILD conference in Seattle, in early May, Microsoft has unveiled some important announcements for developers that will have an important impact on the Universal Windows Platform. The major ones are:

1. **XAML Standard 1.0:** as we're going to learn in detail in the next chapter, XAML is the markup language that is used by the Universal Windows Platform application to design the user interface. However, at the time of writing, the features and the syntax of this language are a bit fragmented across the various Microsoft technologies. For example, another important Microsoft platform that uses this language is Xamarin Forms, which is a technology that developers can use to write cross-platfrom applications (for Windows, Android and iOS), sharing the same user interface and code. However, currently, the two platforms, despite they are both based on XAML, have some naming differences in the syntax, making not possible to reuse the same code moving from one technology to the other.

2. **.NET Standard Libraries 2.0**: .NET Standard Libraries are the evolution of the Portable Class Libraries, and they implement a subset of .NET ecosystem (typically, the parts that are independent from the platform), allowing developers to share the same library (like a backend library to interact with a cloud service) across multiple Microsoft platforms. The version 2.0 of this technology will add support for the Universal Windows Platform, allowing developers to share parts of the code of the Windows 10 application with an Android application built with Xamarin, with a website running on Linux developed with ASP.NET Core or with a classic desktop application for Windows created with WPF.

Chapter 2 The Essential Concepts: Visual Studio, XAML, and C#

The templates

When you launch Visual Studio 2015 for the first time and you choose to create a new project, you'll notice many different categories. The ones that are interesting for our purposes belong to the **Windows** category. Here you will find the following subcategories:

- **Universal** contains the templates needed to create a Universal Windows Platform app.
- **Windows 8** contains multiple subcategories for each previous version of the platform, like Windows 8.1, Windows Phone 8.1, or Silverlight apps for Windows Phone 8.0 and 8.1.
- **Classic Desktop** contains the templates to create classic desktop applications using technologies like Windows Forms or WPF, which are based on the standard .NET framework.

This book will cover only Universal Windows Platform apps, so the ones that can be created starting from the templates inside the **Universal** section. If you are required to create an application for prior Windows or Windows Phone versions, you can find different books written by me and published by Syncfusion in the *Succinctly* series.

Pages, XAML, and code-behind

Universal Windows Platform apps abandon the old windowing.based paradigm to use a new paradigm based on pages, which are organized in a hierarchical way. The user, when launching the app, lands on the main page. After that, he can move to the other pages, which contain different and specific content. For example, a news application can display a list of the most recent news in the main page. The user can tap one of the list items and navigate to a detail page, which will display the full text of the news. In addition, the application can have more pages: one for videos, one for photo galleries, one for a specific news category, etc.

All the pages inside an application are composed of two different files:

- The main one, which ends with the **.xaml** extension, contains the visual layout of the page and is written with a language called XAML, which is a XML dialect.
- The code-behind, which ends with the **.xaml.cs** extension, contains the code able to interact with the user interface and perform logic operations. To see this file in Solution Explorer, you'll have to click the little arrow that is displayed near the XAML file. In fact, the code-behind file is displayed as a child of the XAML file in the tree structure. The language used in the code-behind depends on the projection you're using. In the samples covered in this book, it will contain C# code and, thus, the file name ends with

the **.cs** extension. If, for example, you had chosen VB.NET as the development language, the code-behind file would have been named **xaml.vb**.

The project's structure

Regardless of the template you're going to use, there are some files and folders that are essential for a Universal Windows Platform app and that are included in every project. Let's look at the most important ones.

The App class

Initially, you may think of this file as representing one of the application pages. In fact, like any other page, it's composed of a XAML (**App.xaml**) and a code-behind file (**App.xaml.cs)**. The **`App`** class is a special one, since it's the entry point of every UWP app. It takes care of initializing everything needed by the application to properly work, like initializing the **`Frame`** class that manages the different pages of the application or handling all the entry points to manage the application's lifecycle.

One important feature of the **`App`** class is that its instance is kept alive until the app is closed or suspended. Every property declared in this class can be accessed for later usage when the application is running. Another consequence of this behavior is that the **`App`** class is the central point where you register all resources (like styles and templates), which can be used from the controls placed in the application's pages. We'll see in detail how to use this feature later in the current chapter.

The Assets folder

This folder typically contains all the visual assets (images, logos, icons, etc.) used in the application. It's not a strict requirement; you can place such data in any other folder of the project. However, it's a good practice to place everything under the Assets folder.

The manifest file

In the project, you'll find a special file called **Package.appxmanifest**. It's the **manifest file** and it's very important: its purpose is to define all the main features of the application, like the standard visual assets (logos, tiles, etc.), the metadata (name, description, etc.), the capabilities, the integration with operating system, etc. Under the hood, it's an XML file, but Visual Studio provides a visual editor, which is automatically loaded when you double-click it.

The manifest is composed of the following sections:

- **Application:** this section describes all the base metadata of the application, like the name, the default language, the supported orientations, and the push notifications configuration.

- **Visual Assets**: this section describes the visual layout of the application by defining all the default images used as logos for the Store as background for the default tile or as splash screen to display when the application is being loaded. Since Windows 10 supports multiple resolutions and screen sizes, this section will allow you to upload different formats for the same image. In the second book of the series, you'll better understand how Windows 10 manages this scenario.

- **Capabilities**: this section is used to set up which features (hardware and software) the application is using, like the Internet connection, the geolocation services, access to the picture library, etc. In this book, you'll find a special note every time we're going to talk about a feature that requires enabling a specific capability to be used.

- **Declarations**: this section is used to extend the application, so that it can deeply interact with the operating system or with other applications. Every time we're going to use some code that extends applications (like performing operations in the background or sharing content), we'll need to set up the entry points in this section. We'll detail this section later, when we talk about contracts and background execution.

- **Content URLs**: this section is specific for a control called **WebView**, which can be used to display web content inside the application (like a HTML page). This control offers developers a way to interact with the page by intercepting and calling specific JavaScript functions. This feature is enabled only for trusted websites, for which the URLs need to be added in this section. To improve security, only sites that use the HTTPS protocol are supported.

- **Packaging**: this last section can be used to customize some information about the package that you will publish on the Store, like the application name, the publisher name, or the version number. Much of the information detailed in this section is automatically set when you're going to associate your app with the Store to publish it. You'll find more details at the end of these e-books.

The XAML

XAML stands for Extensible Application Markup Language. It's based on XML and it's used to define the visual layout of a page, similar to the way HTML is used to define the layout of webpages.

Controls (like buttons or a block of text) are identified by an XML tag, which is inserted inside a page following a hierarchical structure. Tags can be inserted inside other tags to define a relationship. For example, this approach is widely used to define the layout of the page. There are some special controls (which we'll detail in Chapter 3) that act as container for other controls and, consequently, they are nested one inside the other.

Here is what a page definition looks like:

Code Listing 4

```
<Page
    x:Class="Styles.MainPage"
```

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

    <Grid>
        <StackPanel>
            <TextBlock Text="Hello world" />
        </StackPanel>
    </Grid>
</Page>
```

Every page inherits from a base class called **Page**, the starting node. Every other tag placed inside it will define the page layout. You can see that there's an attribute called **x:Class** in the page definition: it defines which is the code-behind class connected to the page. In the sample, it's a class called **MainPage**, which belongs to a namespace called **Styles**.

Namespaces

Namespaces should already be familiar to developers; it's a way to better organize your code by defining a logical path for your classes. This way (even if it's not a suggested approach) you can also have two classes with the same name, given that they belong to two different namespaces. Namespaces are separated using a dot. For example, if you have a class called **Person** that belongs to the **Entities** namespace, it will be represented by the full definition **Entities.Person**. Usually, as default behavior, namespaces inside a project are mapped with folders. If you create a folder called **Entities** and you create a new class inside it, by default it will belong to the **Entities** namespace.

Namespaces in XAML behave in the same way. XAML controls are, in the end, standard classes that belong to a specific namespace. When you want to use a control inside a page, you must make sure that the proper namespace is added in the page definition. This is true especially for custom controls created by the developer or for third party controls that are included in external libraries. In fact, most of the native controls (buttons, text boxes, etc.) can be used without having to worry about the namespace.

Let's see an example of how to use a namespace in XAML by including in our project the **Microsoft Store Services SDK** (which can be downloaded here) an external library that adds a series of controls and APIs to interact with the services provided by the Store. This library, among other things, includes a control to display advertising in your apps, called **AdControl**. However, since it's part of an external library and not embedded into the Universal Windows Platform, we need to declare the namespace it belongs to before using it. This control is included in the namespace **Microsoft.Advertising.WinRT.UI**; consequently you'll need to add the following declaration in the **Page** definition:

Code Listing 5

```
xmlns:ad="using:Microsoft.Advertising.WinRT.UI"
```

Every namespace starts with the **xmlns** prefix (XML Namespace), which is always required. Then you need to specify a unique identifier for the namespace, which will be used inside the page every time you need to gain access to a control or to a class that belongs to it (in the previous sample, it's **ad**). Here is the sample code to display the advertising control inside a page:

Code Listing 6

```
<ad:AdControl x:Name="MyAdControl" />
```

As you can see, we've added as prefix to the name of the control (**AdControl**) the identifier we've previously assigned to the namespace (**ad**).

Properties and events

Every control can be customized in two ways: by defining properties and by subscribing to events. Each of them is identified with an attribute of the control, even if they have two different purposes.

Properties are used to determine the control's aspect and behavior and they are set simply by assigning a value to the specific attribute. Let's say that we want to display a text on the page by using a control called **TextBlock**. In this case, we'll need to change the value of a property called **Text**, like in the following sample:

Code Listing 7

```
<TextBlock Text="Hello world" />
```

However, there are some properties that can't be expressed with a simple string like in the previous sample. For example, if you want to define an image as the background of a control, you need to set a property called **Background** using the extended syntax, like in the following sample:

Code Listing 8

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

The extended syntax is expressed with a node that is set as child of the control. The prefix is the same as the control's name, followed by the name of the property, separated by a dot. In the example, since we need to set the **Background** property of a control called **Grid**, we use the expression **Grid.Background**.

There's a special property offered by any control called **x:Name**. It's a string that univocally identifies it in the page (you can't have two controls with the same name). It's important especially because it allows developers to access the control from code-behind. Thanks to this identifier, you'll be able to read and set properties directly from the code.

For example, let's say you have a **TextBlock** control and you assign it a unique identifier in the following way:

Code Listing 9

```
<TextBlock x:Name="MyText" />
```

In the code-behind, you'll be able to interact with the control simply by using the value of the **x:Name** property. The following sample shows how to change the **Text** property in the code:

Code Listing 10

```
MyText.Text = "Hello world!";
```

Events, on the other hand**,** are used to determine how the user or the application is interacting with your control. Every time something happens that involves the control, an event is raised and you'll be able to manage it in the code-behind. A very common event is **Click**, which is exposed by all the controls that offer direct interaction with the user, like the **Button** one. Every time the user presses the button (either with a click of a mouse or a tap of a finger), the **Click** event is raised. You will need to manage it with a specific method called an **event handler**. Visual Studio will help you to define this method in the proper way. After you write the name you want to assign to the event, Visual Studio will take care of creating the event handler in the code for you.

The following sample shows you how to define an event handler for the **Click** event of a **Button** control in XAML:

Code Listing 11

```
<Button Click="button_Click" />
```

Visual Studio will generate for you the following event handler:

Code Listing 12

```
private void button_Click(object sender, RoutedEventArgs e)
{
    MyText.Text = "Hello world!";
}
```

As you can see, event handlers are regular methods, but with a specific definition. They always include in the signature two parameters that, as developers, we can use to better handle the event. The first parameter is called **sender** and it's a reference to the object that invoked the event (in our sample, it will contain a reference to the **Button** control); the second parameter, instead, offers some properties that are useful to understand the event's context. We'll see more detailed sample usages of this parameter in the next chapters. Inside the event handler, you simply need to write the code that you want to execute when the event is raised. In the previous sample, we set the **Text** property of our **TextBlock** control every time the **Button** is pressed.

Visual Studio offers a feature called **IntelliSense**, which can autocomplete the code while you're writing it and offer some useful information about properties and events on the fly. IntelliSense also offers a useful visual reference to distinguish properties and events. The former are identified by a small wrench, the latter by a lightning icon.

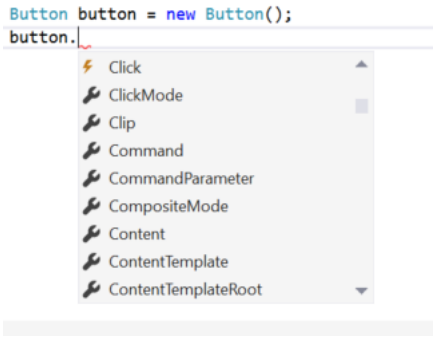


Figure 21: The different icons used to highlight properties and events.

Resources

If you have ever worked with web technologies like HTML, the resources concept will be familiar to you. Exactly like in the HTML world, you can share and reuse styles in multiple pages by using the CSS language. Resources can be used to define a control's style and behavior and to reuse it in different pages of the application.

Resources are defined thanks to a property called **`Resources`**, which is offered by any control. Since XAML is based on a hierarchical structure, every nested control will be able to use the resources defined by its parent. For example, the following sample shows how to define some resources that will be available to a grid control and to any other control nested within it:

Code Listing 13

```
<Grid>
    <Grid.Resources>
        <!-- insert your resources here -->
    </Grid.Resources>
</Grid>
```

However, resources are more often defined with two different scopes: the page one and the application one.

Page resources are defined within the page itself, thanks to the **`Resources`** property offered by the **`Page`** class. This way, all the controls included in the page will be able to access the resources. Here is a sample of a page resources definition:

Code Listing 14

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

```
    <Page.Resources>
        <!-- insert here your resources -->
    </Page.Resources>
</Page>
```

Application resources, instead, are defined using the **Resources** property of the **Application** class that is defined in the **App.xaml** file. This way, the resources will be available to any control placed in any page or user control of the application. Here is a sample definition:

Code Listing 15

```
<Application
    x:Class="Styles.MainPage "
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Styles">

    <Application.Resources>
        <!-- insert your resources here -->
    </Application.Resources>

</Application>
```

Like controls are univocally identified by the **x:Name** property, resources are identified with the **x:Key** property. To apply a resource to a control's property, you need to use a special XAML syntax called **markup extension**. It's a way to describe, directly in XAML, complex operations that, otherwise, would require writing some logic in code. There are many markup extensions available in XAML and we'll talk about some of them during this chapter.

The one that is used to apply a resource to a control is called **StaticResource**. Here is, for example, how to use it to apply a style to a **TextBlock** control:

Code Listing 16

```
<TextBlock Style="{StaticResource CustomStyle}" />
```

The resource is applied by including the **StaticResource** keyword inside braces, followed by the name of the resource (which is the value assigned to the **x:Key** property).

In some cases, especially if you have a lot of resources, the page or the application's definition can be become too crowded and hard to read. XAML offers you a way to better manage resources by declaring them in a dedicated file, in the same way you can define CSS styles in another file in HTML and not just inline.

In XAML, these external files are called **Resource Dictionaries**. Visual Studio offers a specific template to create such files: just right-click in Solution Explorer on your project and choose **Add -> New Item**. You'll find as one of the available templates a file type called **Resource Dicitonary**. Automatically, it will create a file with the following definition:

Code Listing 17

```
<ResourceDictionary
```

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

    <!-- here you can place all your resources -->

</ResourceDictionary>
```

Using this file is easy. You just have to include all your resources inside the **ResourceDictionary** tag, in exactly the same way you did when you added them as page or application resources. Then you need to include the Resource Dictionary file inside the main application by declaring it in the App.xaml file in the following way:

Code Listing 18

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

Resource dictionary files are added inside the **MergedDictionaries** property, offered by the **ResourceDictionary** class. In the previous sample, you can see that we've added just one file, but you can add as many as you want (for example, in case you want to split resources in different files, based on their type or use case).

Let's see now, in detail, which kind of resources the XAML framework offers to developers.

Styles

Styles in XAML are similar to CSS styles. Their purpose is to collect multiple property definitions in one style so that all the properties are automatically changed when they're applied to a control. This way, in case you change your mind and you want to edit one of the properties, you can do it just in one place (the style definition), instead of manually editing all the controls.

Here's what a style looks like:

Code Listing 19

```
<Style TargetType="TextBlock" x:Key="RedStyle">
    <Setter Property="Foreground" Value="Red" />
    <Setter Property="FontSize" Value="30" />
</Style>
```

Styles are identified by the **Style** control, which, other than the **x:Key** property we've already learned about, offers an attribute called **TargetType**, used to specify which kind of controls this style can be applied to. Inside the **Style** control, you can place as many **Setter** tags as you want. Each of them can change the value (using the **Value** attribute) of a specific property (defined by the **Property** attribute).

In the previous sample, we've defined a style which can be applied just to **TextBlock** controls and that changes two of its properties: the color (**Foreground**) and the text size (**FontSize**).

Styles offer a way to apply them not just to a specific control, but to any control whose type matches the one we've defined in the **TargetType** property. This way, you won't have to manually apply the style using the **StaticResource** property, as it will be automatically applied. These styles are called **implicit styles** and they are defined simply by omitting the **x:Key** property, like in the following sample:

Code Listing 20

```
<Style TargetType="TextBlock">
    <Setter Property="Foreground" Value="Red" />
    <Setter Property="FontSize" Value="30" />
</Style>
```

This way, all the **TextBlock** controls will automatically display the text using a bigger, red font. Implicit styles are applied based on the scope in which they've been defined. If they've been declared in a page, they will be automatically applied to all the controls in the page; otherwise, if they've been declared as application resources, they will be applied to all the controls in all the pages.

It's important to remember the hierarchical nature of XAML. When it comes to styles, this means that inner styles always win over outer styles. If, for example, you have defined a style at the application level that changes all the **TextBlock**'s font colors to red, but then you define another style at the page level that changes the color to blue, the page one will win over the application one.

Data templates

Data templates are special resources that can be applied to some controls to define the visual layout. They are often used in combination with controls used to display collections of items, like **ListView** or **GridView** (we'll talk about them in Chapter 3).

The data template simply contains the XAML used to render each item of the list. This way, the XAML will be automatically repeated and applied to every element of the list. Let's say that you want to display a list of people—here's what a data template for this purpose could look like:

Code Listing 21

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

For now, just ignore the **Binding** keyword. It's a new markup extension, which will be covered in detail later in the chapter. For the moment, it's important just to know that you'll be able to display, with this data template, every person in the collection's name and surname.

Data templates behave like any other resource. They can be defined inline inside the control, as page or application resources, or in a resource dictionary. Then, you can apply them using the **StaticResource** keyword. Typically, when you're dealing with controls to display collections, data templates are assigned to a property called **ItemTemplate**, which defines the template used for each item in the collection, like in the following sample:

Code Listing 22

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

Brushes

Brushes are XAML elements that are used to define how a control is filled. For example, when you set the background color of a **Button** control, you're using a brush. There are many kinds of brushes: the simplest one is called **SolidColorBrush** and it's used to express a color. Most of the time, you'll be able to apply this brush using the standard property's syntax. It's enough to assign the color's name to the required property, since the XAML runtime will take care of creating a **SolidColorBrush** for you under the hood. For example, here is how to create a **Rectangle** shape with a red background:

Code Listing 23

```
<Rectangle Width="200" Height="200" Fill="Red" />
```

However, there are also more complex brushes that can be expressed only with the extended syntax. For example, you can apply a gradient instead of a simple color by using a **LinearGradientBrush** or a **RadialGradiantBrush**. They both have the same purpose, but while the first one uses a line as a separator between the colors, the second one applies a circular effect.

Here is how you can apply a gradient brush to the same **Rectangle** control we've seen before:

Code Listing 24

```
<Rectangle Width="200" Height="200">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Blue" Offset="0" />
            <GradientStop Color="Red" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Inside a gradient brush, we can insert multiple **GradientStop** controls. Each of them defines one of the colors that will be applied and you can specify the location where the gradient stops with the **Offset** property. Optionally, you can also apply two properties to the **LinearGradientBrush** called **StartPoint** and **EndPoint** to define the points' coordinates where the gradient should start and end.

Finally, you can also apply an image as a brush by using an **ImageBrush** control, which also requires the extended syntax, as you can see in the following sample:

Code Listing 25

```
<Rectangle Width="200" Height="200">
    <Rectangle.Fill>
        <ImageBrush ImageSource="background.png" />
    </Rectangle.Fill>
</Rectangle>
```

Handling resources based on the theme

So far, we've seen just one way to apply resources to a control: by using the **`StaticResource`** keyword. However, the Universal Windows Platform offers another markup extension called **`ThemeResource`**, which can be used to automatically adapt a resource based on the device's theme.

Let's take a step back and see in detail how this feature works. Windows 10 supports the concept of theme, a set of resources applied globally to the entire operating system. Some themes are available just to meet the visual preferences of the user, like the **dark theme** (white text on dark backgrounds) and the **white theme** (dark text on white backgrounds). Some others, instead, offer people with visual handicaps a better user experience, like the **high contrast** one.

Developers need to keep in mind this feature when it comes to designing an application. Otherwise, there's the risk that it will only look proper with one of the themes, making it unusable with the others. Let's say that you have a **`TextBlock`** control in the page and you force the text's color to be white: if a user has applied the white theme on their device, they won't be able to read the text. Thanks to the **`ThemeResource`** keyword, you'll be able to define multiple resources with the same name. The system will automatically apply the one that works best for the current theme.

Here is sample code:

Code Listing 26

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.ThemeDictionaries>
            <ResourceDictionary x:Key="Dark">
                <SolidColorBrush Color="Red" x:Key="ApplicationTitle" />
            </ResourceDictionary>
            <ResourceDictionary x:Key="Light">
                <SolidColorBrush Color="Blue" x:Key="ApplicationTitle" />
            </ResourceDictionary>
        </ResourceDictionary.ThemeDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

As you can see, we've defined (as application resources) two resources with the same name. They're both **`SolidColorBrush`** items and they have the same **`x:Key`** value, **`ApplicationTitle`**. However, the two brushes have a different value. The first one sets the color to **`Red`**, the second one to **`Blue`**.

Both of them have been added inside a **ResourceDictionary** property called **ThemeDictionaries**. An important difference is that each **ResourceDictionary** has a unique identifier, assigned with the **x:Key** property. This identifier tells the system which theme the resources are referring to by using a specific naming convention:

- **Default**, which is applied as the default theme.
- **Dark**, which is applied when the dark theme is used.
- **Light**, which is applied when the light theme is used.
- **HighContrast**, which is applied when the high contrast theme is used.

Now you just need to apply your resource in the same way you did before, but using the **ThemeResource** markup extension instead of the **StaticResource** one. Here is sample code that shows how to apply the previous style to a **TextBlock** control:

Code Listing 27

```
<TextBlock Text="Title" Foreground="{ThemeResource ApplicationTitle}" />
```

One of the most useful advantages of this markup extension is that it's able to detect the theme change at runtime. This way, if the user changes the theme while the app is running, all the resources will be automatically adapted without having to restart it.

You can also force a theme for your application by applying the **RequestedTheme** property to the whole application (in the **App.xaml** file) or to a single page or control. This way, the resources will ignore the user's theme and only follow the rules defined by the forced theme. The following sample shows how to force the entire application to use the **Dark** theme:

Code Listing 28

```
<Application
    x:Class="Qwertee.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Qwertee"
    RequestedTheme="Dark">

</Application>
```

Animations

Animations are probably one of the most powerful resources in XAML. With a few lines of XAML code, you'll be able to animate virtually any control in the page. For instance, you can make a control disappear, move to another position, or change the size, and so on.

Animations are rendered using a control called **Storyboard**, which offers different types of animations:

- **DoubleAnimation** is used when you want to animate the control by changing a numeric property (like the **FontSize**).

- **ColorAnimation** is used when you want to animate the control by changing a color property.

- **PointAnimation** is used when you want to animate the control by changing its coordinates.

Before seeing in detail how an animation works, let's see sample code that defines a **DoubleAnimation**:

Code Listing 29

```
<Storyboard x:Name="Animation">
    <DoubleAnimation Storyboard.TargetName="MyShape"
                     Storyboard.TargetProperty="Opacity"
                     From="1.0"
                     To="0.0"
                     Duration="0:0:5" />
</Storyboard>
```

Please meet another XAML concept, called **attached properties.** They are special properties inherited from one control, but that can be applied to others. In this case, **TargetName** and **TargetProperty** are two attached properties. They are exposed by the **Storyboard** control, but they're applied to the **DoubleAnimation** one. Their purpose is to define where the animation will be applied. **TargetName** defines the name of the control, while **TargetProperty** defines the name of the property whose value will be changed during the animation.

In the previous sample, we're changing the **Opacity** property of a control identified by the name **MyShape**. The other three properties define the behavior of the animation. **From** and **To** are used to define the start and end values of the property, while **Duration** is used to express the animation's length. In this case, we're changing the control's **Opacity** value from 1.0 to 0.0. The animation will last 5 seconds. The result will be that, after 5 seconds, the control will disappear.

With the previous code, the animation is equally distributed according to the specified length. However, XAML also offers a way to change this behavior by using one of the controls that ends with the suffix **UsingKeyFrames**. The following sample shows another approach to define a **DoubleAnimation**:

Code Listing 30

```
<Storyboard x:Name="Animation">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="MyShape"
                                   Storyboard.TargetProperty="Opacity"
                                   Duration="0:0:10">
        <LinearDoubleKeyFrame KeyTime="0:0:3" Value="0.8" />
        <LinearDoubleKeyFrame KeyTime="0:0:8" Value="0.5" />
        <LinearDoubleKeyFrame KeyTime="0:0:10" Value="0" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

We're using the alternative version of the control, called **DoubleAnimationUsingKeyFrames**. The difference is that, this time, we exactly specify the animation timings using the **LinearDoubleKeyFrame** control and its **KeyTime** and **Value** attributes. In the sample, the result is the same (the control disappears after 10 seconds), but not with different timings. After 3 seconds, the **Opacity** property will be set to 0.8, after 8 seconds to 0.5, and after 10 seconds to 0, making the control disappear.

Easing animations

There are animations that are pleasant to see for the user, but that can be hard to implement. Let's say that you have a shape on the page and you want to simulate it falling towards the bottom of the screen. When the shape touches the bottom margin, it should bounce, like it's a ball. This kind of animation can be complex to define, since it requires taking into consideration laws of physics like acceleration and gravity.

The XAML frameworks offer built-in animations, called **easing animations**, which can be used to implement such behaviors without dealing with all the complexity behind them. Let's see how to implement the bouncing sample using one of these animations:

Code Listing 31

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

In this case, other than defining the animation in the regular way (in this sample, it's a **PointAnimation** that moves a shape from one position of the screen to another), we set the **EasingFunction** property with one of the many built-in easing animations available. In the previous sample, we're using a **BounceEase** control, which can be used to add a bouncing effect to the control. Every easing animation offers a set of specific properties to customize it. For example, the **BounceEase** one offers a property called **Bounces** to define how many bounces the control should perform at the end of the animation.

You can see a list of all the available easing functions in the MSDN documentation.

System animations

The Universal Windows Platform offers a built-in set of animations that covers many common scenarios (like fade-in or fade-out effects). You can identify them by the **ThemeAnimation** suffix. Using them is simple: you just add the control in a **Storyboard** tag. In the previous samples, we've seen how to manually apply a fade-out effect to a control by changing the **Opacity** property from 1 to 0. We can achieve the same result using a built-in animation called **FadeOutThemeAnimation**, like in the following sample:

Code Listing 32

```
<Storyboard x:Name="Fade" TargetName="MyShape">
    <FadeOutThemeAnimation />
</Storyboard>
```

You can see a list of all the available system animations here.

Controlling the animations

Animations are defined as resources, like we've seen for styles and data templates. The only difference is, instead of using the **x:Key** property to identify them, we need to use the **x:Name** one, in the same way we do for regular controls. Here is a sample of an animation defined as a page resource:

Code Listing 33

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

    <Page.Resources>
        <Storyboard x:Name="Animation">
            <DoubleAnimation Storyboard.TargetName="MyShape"
                             Storyboard.TargetProperty="Opacity"
                             From="1.0"
                             To="0.0"
                             Duration="0:0:5" />
        </Storyboard>
    </Page.Resources>

</Page>
```

Thanks to the unique identifier, we can control the animation in the code in the same way we can interact with the controls placed in the page. The **Storyboard** controls offer some methods to play, stop, or resume the animation. The following sample shows two event handlers, connected to the **Click** event exposed by two buttons, that are used to start (with the **Begin()** method) and stop (with the **Stop()** method) the animation:

Code Listing 34

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

Transitions

Transitions are not very different from animations but, instead of being executeable at any time, they are performed only when a specific event happens, like a page is loaded or an element in a collection is deleted. The Universal Windows Platform offers native support to transitions. Instead of manually defining the animation, we will simply need to define which one to use by using the **`Transitions`** property that is offered by any control. Unlike animations, we won't need to set up a **`Storyboard`**, since we don't control the execution.

Here is a sample of transitions usage:

Code Listing 35

```
<Button Content="Transition test">
    <Button.Transitions>
        <TransitionCollection>
            <EntranceThemeTransition />
        </TransitionCollection>
    </Button.Transitions>
</Button>
```

The **`Transitions`** property accepts a **`TransitionCollection`** element. Its purpose is to support multiple transition effects in case you want to manage multiple events (for example, you want to apply both an entrance and an exit transition). In this sample, we've applied just an entrance effect by using the **`EntranceThemeTransition`** control.

As usual, since we're working with XAML, transitions are propagated to every element nested inside a control. For example, if we had applied the **`EntranceThemeTransition`** to a **`Grid`** control, every other control placed inside it would inherit the entrance animation.

Another scenario in which transitions are often applied are collections. For example, you can have animation applied to every single item when the page is loaded, or when an item is removed or added to the list. This scenario is implemented in the same way we've seen for standard controls. The only difference is that, instead of using the **`Transitions`** property, we use the one called **`ItemContainerTransitions`**, which is supported by any control that can display collections of data. Transitions assigned to this property will be automatically applied to every item in the list. The following sample shows this behavior using an **`ItemsControl`** control:

Code Listing 36

```
<ItemsControl x:Name="People">
    <ItemsControl.ItemContainerTransitions>
        <TransitionCollection>
            <EntranceThemeTransition />
            <AddDeleteThemeTransition />
        </TransitionCollection>
    </ItemsControl.ItemContainerTransitions>
</ItemsControl>
```

Most of the advanced controls offer built-in support for transitions. For instance, the previous sample (a collection of items with a transition effect applied when the items are deleted, added,

or displayed in the page) is automatically implemented by controls like **GridView** or **ListView**, which will be covered in the next chapter.

Visual states

Most of the XAML controls support the concept of **visual state**. A control can assume many states and each of them can have a different visual representation. Let's take as an example a **Button** control: by default, it's displayed with light grey background and black text. If you move the mouse over it, a border will be applied to the button. Or, if you press it, its state changes again: the background becomes dark grey.

Visual states are an easy way to define the different states without having to rewrite the layout of the control from scratch for each state. Each control, in fact, has a base state plus a set of other visual states that are expressed by defining the differences from the original state. In the previous sample, the base template of a **Button** will contain the whole definition of the control. The visual state related to the "pressed" state, instead, will simply change the background and the text color of the original template.

Let's see an example by adding a couple of **TextBlock** controls to a page:

Code Listing 37

```
<Grid>
    <StackPanel>
     <TextBlock Text="Text 1" x:Name="FirstText" />
     <TextBlock Text="Text 2" Visibility="Collapsed" x:Name="SecondText" />
     <Button Content="Change state" Click="OnChangeVisualStateClicked" />
    </StackPanel>
</Grid>
```

You can see that the first **TextBlock** is visible when the page is loaded, while the second one is hidden, since the **Visibility** property is set to **Collapsed**. Our goal is to reverse this situation and to make the first **TextBlock** disappear, while making the second one visible. Here is how we can achieve this result without writing any code, just by using XAML and the visual states:

Code Listing 38

```
<Grid>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="Default" />
            <VisualState x:Name="ChangedState">
                <Storyboard>
                    <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="FirstText"

Storyboard.TargetProperty="Visibility">
                        <DiscreteObjectKeyFrame KeyTime="0"
Value="Collapsed" />
                    </ObjectAnimationUsingKeyFrames>
```

```
                    <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="SecondText"

Storyboard.TargetProperty="Visibility">
                        <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"
/>
                    </ObjectAnimationUsingKeyFrames>
                </Storyboard>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <StackPanel>
        <TextBlock Text="Text 1" x:Name="FirstText" />
        <TextBlock Text="Text 2" Visibility="Collapsed" x:Name="SecondText"
/>
        <Button Content="Change state" Click="OnChangeVisualStateClicked"
/>
    </StackPanel>
</Grid>
```

We've defined a **VisualStateManager**, which offers a property called **VisualStateGroup**. Inside it we can define the different visual states we want to manage in the page. In this sample, we've created two states: one called **Default** and one **ChangedState**. The first one includes an empty definition. It's the base state, which simply displays the controls as they've been defined in the page. The second state, instead, contains a **Storyboard** with a set of animations. The first one is applied to the first **TextBlock** and changes the **Visibility** property to **Collapsed** to hide it. The second one is applied to the second **TextBlock** and changes the **Visibility** property to **Visible**, so that it can be displayed.

Once we've defined the visual states, we need to trigger them according to our requirements. One way is to do it in code-behind, by using the **VisualStateManager** class, like in the following sample:

Code Listing 39

```
private void OnChangeVisualStateClicked(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToState(this, "ChangedState", true);
}
```

If you remember the previous XAML definition, we've inserted a **Button** control. When it's pressed, the previous event handler is invoked to trigger the visual state change. The goal is achieved by using the **GoToState()** method offered by the **VisualStateManager** class, which requires three parameters: the control that is going to change its state (typically, it's defined inside the same page, so it's enough to use the **this** keyword), the name of the state to apply, and a **Boolean** parameter, which tells the **VisualStateManager** to apply or not an animation when the state changes.

The one we've just seen is a very simple example, but visual states are very useful especially when you must deal with complex controls. You can redefine a control's visual states at anytime

using the Visual Studio designer or Blend (a XAML designer tool that is installed together with Visual Studio). It's enough to right-click on it and choose the option **Edit template -> Create a copy**. This way, the tool will generate a copy of all the default styles applied to the control, including one called **`Template`**, which contains a list of all the available visual states. If you try to perform this operation on a **`Button`** control, you'll find the control can assume many visual states, like **`Pressed`** or **`Disabled`**.

Visual states play a key role in implementing one of the most important requirements for a Universal Windows Platform app: adaptive layout. Thanks to visual states, we'll be able to define how the pages of our application will look based on the size of the screen, and to change the visual layout of the page based on the available space. For example, a page may show two controls side by side when the app is running on a big screen, but with a visual state, we can stack the two controls one below the other when the app is running on a smaller screen.

We'll learn more about the techniques to implement an adaptive layout and to support different screens and devices properly in the next book of the series, dedicated to creating user interfaces.

Data binding

Data binding is one of the most powerful XAML features and it's crucial to learn it if you want to get serious with Windows development. Data binding is a way to create a communication channel between the user interface and a data source (which can be a control or a property in the code). In addition, the XAML framework offers a notification mechanism (that will be detailed later) connected to data binding that can create a real-time channel. Every time something changes on one side of the channel, the other side is automatically notified and updated.

When you set up a binding, you create a communication channel that involves a **source** (the data source) and a **target** (the control in the user interface that is going to display the data to the user). As default behavior, the channel is created in **`OneWay`** mode. When the source changes, the target is automatically updated, but not the other way around. Binding is defined using a specific markup extension, called **`Binding`**, like in the following sample:

Code Listing 40

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

In this case, the source is specified inside the markup extension using the **`Path`** attribute (in the previous sample, it's a property called **`Name`**). The target is the property to which the binding is assigned (in this case, the **`Text`** one). Specifying the **`Path`** attribute is optional—the following code works in exactly the same way:

Code Listing 41

```
<TextBlock Text="{Binding FirstName}" />
```

However, binding also offers a way to create a two-way communication channel. For example, the **TextBox** control can be used not only to display text, but also to receive text as input from the user. In this case, we need not only for changes from code to be reflected in the user interface, but also for us to be able to access a text from code when the user inserts one. To support this scenario, we need to explicitly set the **Mode** property of the binding, like in the following sample:

Code Listing 42

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

There's also a third **Mode** option, **OneTime**. With it, the binding will be evaluated only when the page is loaded for the first time. It's useful when we want to connect data we are sure won't change during the execution of the application.

Almost all the XAML controls can use the data binding. Most of their properties, in fact, are defined as dependency properties, which are special properties that, other than offering the standard mechanism to read and write their value, support notifications propagation.

Let's see an example to better understand this concept. Look at the following code snippet:

Code Listing 43

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

In this sample, we've connected the **Text** property of the **TextBlock** control to the **Slider** one. We're still using the **Binding** markup extension, but with a different approach. Instead of just using the **Path** attribute, we added first the **ElementName** one. This way, we can refer to another control in the page. In this case, we refer to the **Value** property of the **Slider** control, which contains the slider's value. Both **Value** and **Text** are dependency properties, so they can propagate notifications when something changes. The result will be that every time the user moves the slider on the screen, the **TextBlock** will automatically update itself to display the slider's value.

Data binding with objects

One of the most powerful data binding features is the ability to connect visual controls to objects in our code. This approach is the base of many important XAML concepts and patterns, like Model-View-ViewModel. To explain it, however, we have first to introduce a new XAML property, called **DataContext**. Its purpose is to define the binding context of a control and, as many other XAML features are, it's hierarchical. As soon as you define a control's **DataContext**, every other nested control will get access to the same binding context.

Let's see a sample: we'll display the info about a person using the data binding approach, instead of manually setting the **Text** property of a **TextBlock** control. The info is stored in a class, which contains some basic info (name and surname):

Code Listing 44

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

Here is, instead, the XAML we're going to use to display such information on a page:

Code Listing 45

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

As you can see, name and surname are displayed using data binding. The two properties of the **Person** class are connected to the **TextBlock** control using the **Binding** markup expression. Let's see now how to create a **Person** object and how to display it using this new approach:

Code Listing 46

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

When the page is created, we define a new **Person** object and we set it as **DataContext** of the **Customer** control, which is the **StackPanel** that contains the **TextBlocks** used to display name and surname. By doing this, we've defined, as binding context of the **StackPanel**, the **Person** object we've just created. This way, we can access the **FirstName** and **Surname** properties using binding.

The INotifyPropertyChanged interface

The previous code has a flaw. Unlike the sample we saw with the **Slider** control, if you change one of the properties of the **Person** class at runtime, during the app's execution, the **TextBlock** controls won't update themselves to display the new values. This happens because **Name** and **Surname** are simple properties and not dependency properties. If you want to enable notification's propagation support, the Universal Windows Platform offers a specific interface, called **INotifyPropertyChanged**, which you should implement in your classes.

Let's see how the **Person** class definition we've previously seen changes to properly support this interface:

Code Listing 47

```
public class Person : INotifyPropertyChanged
{
    private string _firstName;
    private string _surname;
    public string FirstName
    {
        get { return _firstName; }
        set
        {
            _firstName = value;
            OnPropertyChanged();
        }
    }
    public string Surname
    {
        get { return _surname; }
        set
        {
            _surname = value;
            OnPropertyChanged();
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
{
 handler(this, new PropertyChangedEventArgs(propertyName));
}
    }
}
```

Thanks to the **INotifyPropertyChanged** interface, we are able to define an event called **PropertyChanged**, which is raised by the **OnPropertyChanged()** method. Every time we call it, we notify the user interface that the property's value has changed.

The second step is to change the properties definition. They can't be simple properties anymore, because we need to invoke the **OnPropertyChanged()** method every time the value changes. We do it in the setter of the property.

Now the notification's mechanism offered by data binding will work properly. If you change the person's name or surname while the app is running, you'll correctly see the new values in the user interface.

Data binding and collections

Data binding plays a key role when you must deal with collections of data. Every control that can display a collection implements a property called **ItemsSource**, which contains the data to be shown in the page.

We've already seen previously in this chapter how to define a **DataTemplate** and how to use it with collections. Whenever you assign a set of data to the **ItemsSource** property, under the hood you're setting the single item that belongs to the collection as **DataContext** of the **ItemTemplate**.

Let's see a sample with a **ListView** control, which uses the **DataTemplate** we've previously seen as **ItemTemplate**:

Code Listing 48

```
<ListView x:Name="People" >
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="First Name" />
                <TextBlock Text="{Binding Path=FirstName}" />
                <TextBlock Text="Surname" />
                <TextBlock Text="{Binding Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

And here is how we assign a collection of data to the **ListView** control in code:

Code Listing 49

```
public MainPage()
{
    InitializeComponent();
    List<Person> people = new List<Person>
    {
        new Person
        {
            FirstName = "Matteo",
            Surname = "Pagani"
        },
        new Person
        {
            FirstName = "Angela",
            Surname = "Olivieri"
        }
    };

    People.ItemsSource = people;
```

```
}
```

Since the collection is assigned as **ItemsSource** of the **ListView** control, the **DataContext** of the **ItemTemplate** becomes the single **Person** object. Consequently, we're able to display the values of the **FirstName** and **Surname** properties using binding.

Another important class when it comes to managing collections and data binding is the **ObservableCollection<T>** one. It behaves like a regular collection but, under the hood, it implements the **INotifyPropertyChanged** interface. Consequently, every time the collection changes (a new item is added or removed, the item order changes, etc.), the control connected to it will automatically visually reflect the new changes.

It's important to highlight that the **ObservableCollection<T>** class can notify you of only collection changes. If you want to be notified also when one of the properties of the items in the collection changes, you still need to manually implement the **INotifyPropertyChanged** interface in your class.

Converters

Sometimes it can happen that you have some data in your application but, when it comes to displaying it on the page, you don't want to show it as it is, you want to perform some changes first. A common example is when you must deal with a **DateTime** object. If you want to display a list of news articles, it's probably enough to display the date of the news and not the full representation with hours, minutes, seconds, and milliseconds.

Converters are special classes that can satisfy this requirement. They intercept the source data before it is sent to the target control. To properly work, these classes need to implement the **IValueConverter** interface, like in the following sample:

Code Listing 50

```
public class DateTimeConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
string language)
    {
        if (value != null)
        {
            DateTime date = (DateTime)value;
            //the following line formats the DateTime object to return just
the date, without the time information
            return date.ToString("d");
        }
        return string.Empty;
    }

    public object ConvertBack(object value, Type targetType, object
parameter, string
    language)
    {
        if (value != null)
```

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

By implementing the **IValueConverter** interface, you'll be forced to define two methods. **Convert()** is the one invoked when the source data is intercepted and needs to be modified before being sent to the target. The **ConvertBack()** method is invoked in the opposite scenario: when the target needs to send the data back to the source. This method is invoked only when you define a two-way binding, otherwise you'll need to implement just the **Convert()** method.

Both methods will receive some information as input parameters that are needed to perform the conversion. The most important one is **value**, which contains the source data. Since binding can be applied to any object, the **value**'s type is a generic **object**. You'll have to properly cast it to the type you're expecting, based on your scenario.

The previous sample refers to the **DateTime** scenario we've previously introduced. The **Convert()** method takes care of returning just the date, while the **ConvertBack()** method takes the input string and converts it back into a **DateTime** object.

Converters are managed like regular resources. They need to be defined in the **Resources** property offered by controls, pages, or the application itself. Then you can apply them using the **StaticResource** markup extension to the **Converter** attribute in the binding expression. The following sample shows how to declare the previous declared converter as a resource and how to apply it to a **TextBlock** control:

Code Listing 51

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

    <Page.Resources>
        <converters:DateTimeConverter x:Key="DateConverter" />
    </Page.Resources>

    <TextBlock Text="{Binding Path=BirthDate, Converter={StaticResource
DateConverter}}" />

</Page>
```

If you want to add a parameter (that will be assigned to the property called **parameter** included in the signature of the **Convert()** and **ConvertBack()** methods), you just need to add a **ConverterParameter** property to the markup extension, like in the following sample:

Code Listing 52

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

    <Page.Resources>
        <converters:DateTimeConverter x:Key="DateConverter" />
    </Page.Resources>

    <TextBlock Text="{Binding Path=BirthDate, Converter={StaticResource
DateConverter}, ConverterParameter=ShortDate}" />

</Page>
```

It's important to highlight that converters shouldn't be abused. They can have a negative impact on performance, since the converter's logic needs to be invoked every time the binding expression changes. In more complex cases, it's better to directly modify the original property or to define a new property in the class to hold the value to display.

The DataTemplateSelector

Sometimes converters are used not just to change the data before it's displayed, but also when you want to change the visual layout of a control based on the data (for example, you want to hide or display it or change one of its properties, like the color). However, due to the potential performance issues mentioned before, this approach isn't always the best solution, especially if you need to deeply change the layout based on data.

For these scenarios, the Universal Windows Platform offers a better approach called **DataTemplateSelector**, which is a special class that can return a different **DataTemplate** based on our requirements. This way, we won't have to create a layout with a lot of converters, we'll simply define two (or more) different templates. Based on our needs, data will be rendered using the proper one.

To see how this feature works, let's change the **Person** class we've seen before by adding a new property called **Sex**, which will tell us if the person is a male or a female:

Code Listing 53

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

```
    public char Sex { get; set; }
}
```

Our goal will be to display a list of people with a different template based on the sex. The background will be blue for a male person, and it will be pink if she's a female. To do this, we need to create a class that inherits from the **DataTemplateSelector** one, and will define the available **DataTemplate** objects and the condition that will be used to decide which one to apply. Here is a full sample:

Code Listing 54

```
public class PeopleTemplateSelector : DataTemplateSelector
{
    public DataTemplate MaleTemplate { get; set; }

    public DataTemplate FemaleTemplate { get; set; }

    protected override DataTemplate SelectTemplateCore(object item,
DependencyObject container)
    {
        Person person = item as Person;
        if (person != null)
        {
            if (person.Sex=='M')
            {
                return MaleTemplate;
            }
            else
            {
                return FemaleTemplate;
            }
        }
        return base.SelectTemplateCore(item, container);
    }
}
```

In our scenario, we're going to use two templates. Consequently, the class defines two different **DataTemplate** objects, one for the male template and one for the female one. By implementing the **DataTemplateSelector** class, we are forced to define the **SelectTemplateCore()** method, which is invoked at runtime when the binding is performed. It's the method that will tell the control that displays the data collection which template to use. For this purpose, the method receives, as input parameter, the current item of the collection as a generic **object**. The first step is to cast it to the type we're working with (in our case, the **Person** class). Then we can check the condition we're interested in and return the proper **DataTemplate**. In our sample, we check the value of the **Gender** property. If it's equal to **M**, we return the **MaleTemplate**, otherwise we return the **FemaleTemplate**.

So far, we've defined just the logic of the **PeopleTemplateSelector**. Now we need to define the visual layout by specifying how the two templates will look. To do this, we simply define two **DataTemplates** as resources, as we would have normally done:

Code Listing 55

```
<Page.Resources>
    <DataTemplate x:Key="MaleTemplate">
        <StackPanel Width="300" Background="LightBlue">
            <TextBlock Text="{Binding Path=FirstName}" />
            <TextBlock Text="{Binding Path=Surname}" />
        </StackPanel>
    </DataTemplate>

    <DataTemplate x:Key="FemaleTemplate">
        <StackPanel Width="300" Background="Pink">
            <TextBlock Text="{Binding Path=FirstName}" />
            <TextBlock Text="{Binding Path=Surname}" />
        </StackPanel>
    </DataTemplate>
</Page.Resources>
```

As you can see, the two templates are basically the same, except for the background color applied to the **StackPanel** control. Now we need to define the **PeopleTemplateSelector** object we've previously created as a resource, too:

Code Listing 56

```
<Page.Resources>
    <DataTemplate x:Key="MaleTemplate">
        <StackPanel Width="300" Background="LightBlue">
            <TextBlock Text="{Binding Path=FirstName}" />
            <TextBlock Text="{Binding Path=Surname}" />
        </StackPanel>
    </DataTemplate>
    <DataTemplate x:Key="FemaleTemplate">
        <StackPanel Width="300" Background="Pink">
            <TextBlock Text="{Binding Path=FirstName}" />
            <TextBlock Text="{Binding Path=Surname}" />
        </StackPanel>
    </DataTemplate>

    <local:PeopleTemplateSelector x:Key="PeopleTemplateSelector"
     MaleTemplate="{StaticResource MaleTemplate}"
     FemaleTemplate="{StaticResource FemaleTemplate}" />
</Page.Resources>
```

If you remember, in the **PeopleTemplateSelector** object we've defined a property for each **DataTemplate** we need to manage. Now we simply need to identify which **DataTemplate** to use for each property. Since they are just resources, we use the **StaticResource** markup extensions, as we would have done with any other resource.

The last step is to assign to the control we're going to use to display the data collection the **DataTemplateSelector**. We can do it by using the **ItemTemplateSelector** property exposed

by most of the collections' controls. The following sample shows how to do it using a **GridView** control:

Code Listing 57

```
<GridView ItemTemplateSelector="{StaticResource PeopleTemplateSelector}" />
```

As you can see, in this case we don't need to define the **ItemTemplate** property. The **PeopleTemplateSelector** object we've created will take care of assigning the proper **ItemTemplate** to every item, based on the logic we've written.

The new x:Bind approach

Data binding is extremely powerful, but it has a couple of downsides:

- It's evaluated at runtime, and as such it can have a negative impact on the performance of the application. Binding, in fact, is achieved using reflection, which means that the XAML rendering engine must go through the whole XAML tree to find the right control and apply the proper value.

- Being evaluated at runtime, you won't find errors until you execute the application. Do you remember the definition of the **Person** class we've previously seen, which contains the **FirstName** and **Surname** properties? Let's say that, at some point, you create the following **DataTemplate** to be used with a **ListView**:

Code Listing 58

```
<ListView x:Name="People" >
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="First Name" />
                <TextBlock Text="{Binding Path=FirstName}" />
                <TextBlock Text="Surname" />
                <TextBlock Text="{Binding Path=Surame}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

As I've highlighted in yellow, the sample code contains an error on purpose. The **Person** class has a property called **Surname**, but we called it **Surame** in the **DataTemplate**. However, since binding is evaluated at runtime, you won't get any errors when you'll compile the application in Visual Studio. You will notice the error only because, when the app is running, you'll see the surname field empty and a message in the Output Windows reporting something wrong with a binding.

To solve both problems, the Universal Windows Platform has introduced a new markup expression called **x:Bind** that, instead, gets compiled during the build process. As such, it's

much faster, and if you make any typo like in the previous example, you will immediately notice it, because the build process will fail.

From a code point of view, the two markup expressions work in the same way, so it's enough to replace **`Binding`** with **`x:Bind`** to start using it, like in the following sample:

Code Listing 59

```
<StackPanel x:Name="Customer">
    <TextBlock Text="First Name" />
    <TextBlock Text="{x:Bind Path=FirstName}" />
    <TextBlock Text="Surname" />
    <TextBlock Text="{x:Bind Path=Surname}" />
</StackPanel>
```

However, there are some important differences to keep in mind.

Different context

We've seen that binding is handled through the **`DataContext`** property. When we use the **`Binding`** markup expression, the XAML infrastructure will look for the property in the **`DataContext`** we've set for the control or, if there isn't one, will follow the hierarchical structure to reach the first one defined by a parent control.

Thus, for example, these properties could be declared in a completely different class of our project. This is one of the fundamental concepts behind the Model-View-ViewModel pattern. This pattern can remove the dependency between the presentation layer and the data layer simply because we can set a normal class (called ViewModel) as **`DataContext`** for the page, which will take care of defining the properties and the interaction logic. Since it's a normal class (unlike the code-behind, which instead has a strong dependency from the XAML page), we can easily implement processes like unit testing, code isolation, etc.

`x:Bind`, instead, since it gets compiled, will use the code-behind class as **`DataContext`**. As such, in the previous example, we will have to declare the **`FirstName`** and **`Surname`** property in the code-behind class and not in a separate class, like in the following sample:

Code Listing 60

```
public sealed partial class MainPage : Page
{
    public string FirstName { get; set; }

    public string Surname { get; set; }
    public MainPage()
    {
        this.InitializeComponent();
        FirstName = "Matteo";
        Surname = "Pagani";
    }
}
```

If you want to get the best of both worlds (leveraging **x:Bind** but, at the same time, also setting the **DataContext** so that you can define your properties in a separate class), the solution is to create a property in the code-behind class that gets populated with the **DataContext**'s value. This way, you'll be able to access all the properties offered by the **DataContext** using the **x:Bind** expression through this property.

Going through the details of the MVVM pattern would be out of context for this explanation, but let's see a real example with a basic implementation and say that your XAML page has the following definition:

Code Listing 61

```
<Page
    x:Class="SampleProject.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    DataContext="{StaticResource MainViewModel}"
    mc:Ignorable="d">

    <StackPanel>
        <TextBlock Text="First Name" />
        <TextBlock Text="{Binding Path=FirstName}" />
        <TextBlock Text="Surname" />
        <TextBlock Text="{Binding Path=Surname}" />
    </StackPanel>


</Page>
```

Since we have set a resource called **MainViewModel** (which is a class included in our project) as **DataContext** of the entire **Page**, we can access through standard binding the **FirstName** and **Surname** properties defined by this class. As such, the **MainViewModel** class could look like this:

Code Listing 62

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

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

However, as already mentioned, if we simply swap the **Binding** expression with the **x:Bind** one, we will get a compile-time exception, because the **DataContext** property will be ignored and the compiler will look for the **FirstName** and **Surname** properties in the code-behind of this

XAML page, which don't exist. So, we can create a property in the code-behind to hold a reference to the **MainViewModel** instance, like in the following sample:

Code Listing 63

```
public sealed partial class MainPage : Page
{
    public MainViewModel ViewModel { get; set; }
    public MainPage()
    {
        this.InitializeComponent();
        ViewModel = DataContext as MainViewModel;
    }
}
```

We simply create a property, a **MainViewModel** type, and in the page's constructor we store the value of the **DataContext** property of the page (we need to perform a cast first, since the **DataContext** property can be of any type, so it's a generic object).

Now, we can start to leverage **x:Bind** simply by using the **ViewModel** property we've just created, like in the following sample:

Code Listing 64

```
<Page
    x:Class="SampleProject.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    DataContext="{StaticResource MainViewModel}"
    mc:Ignorable="d">

    <StackPanel>
        <TextBlock Text="Name" />
        <TextBlock Text="{x:Bind Path=ViewModel.FirstName}" />
        <TextBlock Text="Surname" />
        <TextBlock Text="{x:Bind Path=ViewModel.Surname}" />
    </StackPanel>


</Page>
```

It's enough to attach the **ViewModel** prefix to the name of the property of the **MainViewModel** class we want to use with a dot.

Different binding mode

By default, when we create a binding using the standard **Binding** expression, the applied **Mode** is **OneWay**, which means that every change in the source will be reflected in the target, but not vice versa. If we apply this concept to our previous sample, it means that every time we change the value of the **Name** or **Surname** property and we have properly implemented the **INotifyPropertyChanged** interface, the **TextBlock** control will automatically update its visual layout to reflect the new value.

With the **x:Bind** markup expression, instead, the default value is **OneTime**. This means that the binding expression will be evaluated only when the page gets created and then other changes won't be propagated anymore. The goal of this mode is to save performance. During development, it often happens to create binding channels that continuously listen to changes, though, in the end, they are never really updated during the application's lifecycle. In case we really need our changes in the source to be propagated to the target, we need to explicitly specify the **Mode** property of the **x:Bind** expression to **OneWay**, like in the following sample:

Code Listing 65

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

    <StackPanel>
        <TextBlock Text="Name" />
        <TextBlock Text="{x:Bind Path=FirstName, Mode=OneWay}" />
        <TextBlock Text="Surname" />
        <TextBlock Text="{x:Bind Path=Surname, Mode=OneWay}" />
    </StackPanel>

</Page>
```

Handling DataTemplates

Another difference in using **x:Bind** comes when you need to create a **DataTemplate**. Since the template doesn't rely on a code-behind class, the compiler doesn't know where to look for the properties. Consequently, again, we can't simply swap the **Binding** expression with the **x:Bind** one; we also need to specify the data type we're going to express with the **DataTemplate**.

In the previous samples, we've seen how we create a **DataTemplate** that represents the content of a **Person** object:

Code Listing 66

```
<ListView x:Name="People" >
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="First Name" />
                <TextBlock Text="{Binding Path=FirstName}" />
                <TextBlock Text="Surname" />
                <TextBlock Text="{Binding Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Here is how we need to change it to use the new markup expression:

Code Listing 67

```
<ListView x:Name="People" >
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="local:Person">
            <StackPanel>
                <TextBlock Text="First Name" />
                <TextBlock Text="{x:Bind Path=FirstName}" />
                <TextBlock Text="Surname" />
                <TextBlock Text="{x:Bind Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

We have added the property **`x:DataType`** to the **`DataTemplate`**, with a reference to the base entity we need to represent with the template (since **`Person`** isn't a basic control of the Universal Windows Platform, we need to reference it using a custom namespace like we learned at the beginning of this chapter). With this change, we are now able to properly use the **`x:Bind`** expression instead of the **`Binding`** one.

New x:Bind features

So far, we've focused our attention on the main differences between using the standard **`Binding`** markup expression and the new **`x:Bind`** one. However, **`x:Bind`** doesn't just bring to the table better performance and fewer chances to make errors, but also adds a set of features that aren't options with standard binding. Let's see the most important ones.

Binding with events

We've seen that the traditional **`Binding`** expression can just be used to connect properties of a XAML control with properties declared in a class. **`x:Bind`** also allows you to connect methods by applying the same binding syntax to an event instead of a property. Here is, for example, how we can bind a method to the **`Click`** event exposed by the **`Button`** control:

Code Listing 68

```
<Button Click="{x:Bind Path=SayHello}" />
```

The Universal Windows Platform will expect to find a method called **SayHello** defined in the code-behind. There are two supported ways to declare this method:

- By making it parameterless (which is a feature not supported with standard event handler subscription), like in the following sample:

Code Listing 68

```
public void SayHello()
{
    Message = "Hello world";
}
```

- By matching the signature of the original event handler, as if we are subscribing to the event in the regular way:

Code Listing 69

```
public void SayHello(object sender, RoutedEventArgs e)
{
    Message = "Hello world";
}
```

Improving performances with x:Phase and x:DeferLoadStrategy attributes

x:Phase is a new attribute that can be used in combination with controls whose properties are rendered using the **x:Bind** expression. This attribute is useful when you're defining a **DataTemplate** that will be used with a collection control (like **ListView**) and you want to optimize performance. Let's say, for example, that you have declared the following **DataTemplate**, which will be used as **ItemTemplate** of a **ListView** control that can potentially display thousands of items:

Code Listing 70

```
<DataTemplate x:DataType="local:Person">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>

        <Image Grid.Column="0" Source="{x:Bind Path=Photo}" />
        <StackPanel Grid.Column="1">
            <TextBlock Text="{x:Bind Path=FirstName}" />
            <TextBlock Text="{x:Bind Path=Surname}" />
        </StackPanel>
    </Grid>
</DataTemplate>
```

As you can see, this template, other than just displaying some text data (name and surname), also displays a photo image, which can require more time to be rendered (especially if it's stored on a remote server and not locally). This could affect the overall performance of the list when the user is scrolling it, because each item isn't fully displayed until the image is downloaded and rendered.

To reduce this problem, we can leverage the **x:Phase** attribute to define the rendering order of the items in the template. Here is how we can change the previous **DataTemplate** to leverage this feature:

Code Listing 71

```
<DataTemplate x:DataType="local:Person">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>

        <Image Grid.Column="0" Source="{x:Bind Path=Photo}" x:Phase="2" />
        <StackPanel Grid.Column="1">
            <TextBlock Text="{x:Bind Path=FirstName}" x:Phase="0" />
            <TextBlock Text="{x:Bind Path=Surname}" x:Phase="1" />
        </StackPanel>
    </Grid>
</DataTemplate>
```

We have added the **x:Phase** attribute to each control inside the **DataTemplate** and, by simply using an ordinal number, we have defined the rendering order. In the previous sample, the name and surname (which are plain strings) will be rendered first, while the image (which takes more time to be rendered) will be displayed last.

To further improve performance, we can leverage another property called **x:DeferLoadStrategy**, which we can use to enable lazy loading. The previous example, in fact, will give a more pleasant user experience, but the rendering performances will be the same. This is because **x:Phase** affects the **Opacity** of the control, not the **Visibility**. The difference between the two properties is that, when you set the **Opacity** of a control to 0, it's still rendered in the XAML tree, but it isn't visible. The **Visibility** property, when set to **Collapsed**, prevents the control from being added at all in the XAML tree. The consequence is that, when you leverage just the **x:Phase** property, the element is still rendered in the XAML tree and therefore consumes memory.

By setting the **x:DeferLoadStrategy** to **Lazy**, instead, the **x:Phase** attribute will affect the **Visibility** property of the control, helping to save memory when the control isn't fully rendered yet.

Here is how the **DataTemplate**, updated to leverage this additional property, looks:

Code Listing 72

```
<DataTemplate x:DataType="local:Person">
```

```
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>

        <Image Grid.Column="0" Source="{x:Bind Path=Photo}"
               x:Phase="2" x:DeferLoadStrategy="Lazy"
               x:Name="Photo" />
        <StackPanel Grid.Column="1">
            <TextBlock Text="{x:Bind Path=FirstName}"
                       x:Phase="0" x:DeferLoadStrategy="Lazy"
                       x:Name="FirstName"/>
            <TextBlock Text="{x:Bind Path=Surname}" x:Phase="1"
                       x:DeferLoadStrategy="Lazy" x:Name="Surname" />
        </StackPanel>
    </Grid>
</DataTemplate>
```

There's just one important thing to highlight: to properly work, the **x:DeferLoadStrategy** attribute requires that each control is identified by the **x:Name** property. Otherwise, the XAML infrastructure won't be able to find them when it comes time to start the rendering.

Generally speaking, the **x:DeferLoadStrategy** can be used not just with **DataTemplates**, but with every control placed in a XAML page. When you set the **x:DeferLoadStrategy** to **Lazy**, the control and its children won't be rendered until someone tries to reference it in some way, like:

- Using binding.
- Using a **Storyboard** to start an animation.
- Using the **FindName()** method provided by the Universal Windows Platform in the code-behind class.

Look at the following sample XAML code:

Code Listing 73

```
<Grid x:Name="DeferredGrid" x:DeferLoadStrategy="Lazy">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Rectangle Height="100" Width="100" Fill="#F65314" Margin="0,0,4,4" />
```

```
    <Rectangle Height="100" Width="100" Fill="#7CBB00" Grid.Column="1"
Margin="4,0,0,4" />
    <Rectangle Height="100" Width="100" Fill="#00A1F1" Grid.Row="1"
Margin="0,4,4,0" />
    <Rectangle Height="100" Width="100" Fill="#FFBB00" Grid.Row="1"
Grid.Column="1" Margin="4,4,0,0" />
</Grid>
<Button x:Name="RealizeElements" Content="Realize Elements"
Click="RealizeElements_Click" />
</Grid>
```

Since the **Grid** identified by the name **DeferredGrid** has the **x:DeferLoadStrategy** attribute set to **Lazy**, none of the **Rectangle** controls defined inside it will be rendered in the XAML tree when the page is loaded. However, as soon as someone tries to get a reference to the **DeferredGrid** control, the **Visibility** will be automatically changed from **Collapsed** to **Visible**, forcing all the **Rectangle** controls to be rendered and added to the XAML tree.

In the sample code, you can see that there's a **Button** control called **RealizeElements**. When it's clicked, it will execute the following code:

Code Listing 74

```
private void RealizeElements_Click(object sender, RoutedEventArgs e)
{
    this.FindName("DeferredGrid"); // This will realize the deferred grid
}
```

This is an example of an operation that will trigger the control's rendering, since we're calling the **FindName()** method to get a reference to the **DeferredGrid** one.

This approach will prove very useful when, in the next book of the series, we learn the different techniques that we can use to implement adaptive layout experiences in your app. Thanks to the **x:DeferLoadStrategy** attribute, we can completely avoid loading controls that aren't displayed when there isn't enough space on the screen.

Casting

We've already seen many samples of casting in the previous paragraphs. Casting means converting an object from one type to another. Of course, for the operation to be successful, the two types should be compatible. For example, we can easily cast an **int** into a **double**, but we can't do the opposite because if the number is decimal, it can't be implicitly converted into an integer.

In some other cases, we work with properties of the generic **object** type (like we've seen with the **DataContext** one), since they can accept multiple types as value. This way, with cast, we can convert the generic **object** to the type we expect, like we've seen when we talked about converters.

Casting is achieved by prefixing the type between parentheses to the variable. For example, here is how we can convert an integer number to a floating point type:

Code Listing 75

```
int a = 1;
double b = (int) a;
```

Until the Anniversary Update, casting was available only in the code-behind world. With this new update, you can leverage it also in XAML with the **x:Bind** expression to automatically convert the source binding from one type to another without requiring an explicit converter.

Let's see a real example with a very common scenario: handling the visibility of a control. Most of the time, when you need to hide or display a control based on a condition in your logic, you have a mismatch between the C# code and the XAML. In C#, typically you express this condition using a **bool** property. In XAML, instead, the visibility of a control is handled by the **Visibility** property, which is an enumerator that accepts the values **Visible** or **Collapsed**. Therefore, in every project, you usually end up creating a converter that takes a **bool** value as input and returns the corresponding value of the **Visibility** enumerator.

Thanks to this new feature, we can instead simply apply a cast directly in XAML. For example, let's say that in code-behind you have a **bool** property that you use to determine if the Internet connection is available or not.

Code Listing 76

```
public sealed partial class MainPage : Page
{
    public bool IsInternetAvailable { get; set; }
}
```

Now you can simply use casting in XAML if you want to hide or display a control based on the value of this property. The following sample shows how to hide or display a **Button** in the XAML page based on the value of the **IsInternetAvailable** property:

Code Listing 77

```
<Button Content="This is a button"
        Visibility="{x:Bind ((Visibility)IsInternetAvailable)}" />
```

As you can see, the syntax is the same as we used in C#. After the **x:Bind** markup expression, we specify the property we want to connect, prefixed by the type we want to use for the conversion inside braces.

Invoking functions

Another feature that has been added in the Anniversary Update is **x:Bind** support to functions. Instead of binding a control's property with a simple property in code, we can bind it to a function that gets automatically evaluated every time one of the parameters changes.

Let's reuse the previous sample code where we added two properties in our code-behind class called **FirstName** and **Surname**, and then displayed it in the XAML page using two **TextBlock** controls and the **x:Bind** expression.

Now, let's say that we want to add a new **TextBlock** that displays the full name of the user (which means the name and the surname together in the same string). In the past, we would have had to create a new property in code-behind, perform the string concatenation in the code, and then assign the result to this new property.

Starting from the Anniversary Update, we can now create a method that performs the operation and call it directly in XAML when we perform the binding. Here is what our code-behind looks like:

Code Listing 78

```
public sealed partial class MainPage : Page
{
    public string FirstName { get; set; }

    public string Surname { get; set; }

    public MainPage()
    {
        this.InitializeComponent();
        FirstName = "Matteo";
        Surname = "Pagani";
    }

    public string ComposeFullName(string name, string surname)
    {
        string fullname = $"{name} {surname}";
        return fullname;
    }
}
```

We have added a new method called **ComposeFullName()**to the sample we saw previously in this chapter, which accepts as input parameters two strings. Inside this method, we simply leverage the new C# 6.0 feature to perform string concatenation (so we concatenate the name and the surname, adding a space in the middle) and we return the result.

Now we can invoke this new method directly in XAML when we set up the binding channel:

Code Listing 79

```
<TextBlock Text="{x:Bind ComposeFullName(FirstName, Surname)}" />
```

The benefit of this approach is that if, at some point during the app's execution, the values of the **FirstName** and **Surname** properties should change, the **ComposeFullName()** method will be automatically invoked again and the **TextBlock** will display the updated result of the function.

Asynchronous programming

Asynchronous programming is one of the most important concepts when it comes to developing modern applications. In the past, most applications were developed using a synchronous approach. Until the running operation was completed, the application was basically frozen and the user didn't have a chance to interact with it.

This approach doesn't play well with modern applications. No one would buy a smartphone, a tablet, or a console that doesn't allow to the user to answer a call, to reply to email, or to start a game until the current application has finished its task. The Universal Windows Platform offers two different ways to manage asynchronous programming: callbacks and the async and await pattern.

Callbacks

If you have worked in the past with other development platforms, you have probably used the callbacks approach. However, in the Universal Windows Platform, this approach is not widely used anymore, since most of the APIs rely on the async and await pattern. Still, there are some classes that are using this approach, especially when their purpose is to act as a listener to detect when something changes (for example, the geolocalization services use the callback approach to track a user's movement).

Callbacks are delegate methods that are invoked when an asynchronous operation is ended or has detected a change from the previous value. With this approach, the code that starts the operation and the code that manages it is handled by two different methods. Let's see some code, based on the example previously mentioned of the geolocalization services, which are managed using the **`Geolocator`** class:

Code Listing 80

```
private void OnStartGeolocator(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.PositionChanged += geolocator_PositionChanged;
    Debug.WriteLine("Finding the user's position…");
}

void geolocator_PositionChanged(Geolocator sender, PositionChangedEventArgs
args)
{
    Latitude.Text = args.Position.Coordinate.Latitude.ToString();
    Longitude.Text = args.Position.Coordinate.Longitude.ToString();
}
```

As you can see, we're using two different methods to properly track the user's position. The **`OnStartGeolocator()`** one takes care of initialing the **`GeoLocator`** class and subscribing to the **`PositionChanged`** event; the real tracking is done by the **`geolocator_PositionChanged()`** event handler, which is invoked every time the user's position changes.

Typically, the event handler used to manage the callback receives two parameters as input. The first one is called **sender** and it's the object that triggered the event; the second one contains some useful parameters to understand what's going on. In the previous sample, you can see that the second parameter's type is **PositionChangedEventArgs** and it contains a **Position** property, with the coordinates of the user's position.

The code we've just written is asynchronous. The message in the Visual Studio's Output Windows (printed using the **Debug.Writeline()** method) is displayed immediately, right after the tracking has started. Only when a new position is detected will the callback's method will be executed.

The async and await pattern

The callback approach has the downside of making the code harder to understand and manage for the developer. Unlike synchronous code, the execution flow isn't linear, but jumps from one method to another. The async and await pattern has been introduced in C# 5.0 to solve this problem; the Universal Windows Platform heavily relies on this approach. Every API defining an operation that can require more than 50 milliseconds to be completed has been implemented using it.

When we use the async and await pattern, we write sequential code like it was synchronous. The compiler will execute one statement after the other. Under the hood, the compiler will add a bookmark every time you start an asynchronous operation and then will quit the current method. This way, the UI thread will be released and the application will continue to be fast and responsive. Once the asynchronous operation is terminated, the compiler will resume the execution from the previously set bookmark.

The async and await pattern is heavily based on the **Task** class, which is the base type returned by every asynchronous operation. A method can return two different types:

- **Task**, when it's a **void** method that doesn't return any value to the caller, but simply performs some operations.
- **Task<T>**, when the method returns a value to the caller. In this case, the compiler will wait until the operation is completed and then it will return the result (a **T** type) to the caller.

Let's see a real sample code by using the same **Geolocator** class we previously used to explain the callback approach. In this case, we won't subscribe to keep track of the user's movement, but we will ask for a single position, using a method called **GetGeopositionAsync()** (notice the **async** suffix).

Code Listing 81

```
private async void OnGetPositionClicked(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    Geoposition geoposition = await geolocator.GetGeopositionAsync();
    Latitude.Text = geoposition.Coordinate.Latitude.ToString();
    Longitude.Text = geoposition.Coordinate.Longitude.ToString();
```

```
}
```

We can see the two key features required to properly use an asynchronous method. The first one is that the method signature needs to contain the **async** keyword. Then we are able to add the **await** prefix before calling the asynchronous method (in our case, the **GetGeopositionAsync()**). Thanks to this keyword, the runtime will wait until the operation is completed before moving on. The result is that, until the geolocation services have returned the user's position, the application won't display the user's coordinates on the page.

As you can see, this code is much simpler to read and write and it's completely asynchronous. The operation will be executed on a different thread than the one that manages the user interface, keeping the application fast and responsive.

It's important to highlight, as a rule, that every asynchronous operation needs to return a **Task** or a **Task<T>** object. If you declare an asynchronous method that simply returns **void**, the behavior could be unpredictable. The only exception is when you're dealing with event handlers (like in the previous sample). Since they are "fire and forget" methods (you don't need to wait until, for example, the **Click** event on a **Button** is completed to execute the operation), you can mark them as **async void**.

For example, if the previous sample code had been a simple method instead of an event handler, this would have been the proper way to write it:

Code Listing 82

```
private async Task GetPosition()
{
    Geolocator geolocator = new Geolocator();
    Geoposition geoposition = await geolocator.GetGeopositionAsync();
    Latitude.Text = geoposition.Coordinate.Latitude.ToString();
    Longitude.Text = geoposition.Coordinate.Longitude.ToString();
}
```

As you can see, the method now returns a **Task** object, and consequently we can invoke it using the **await** prefix:

Code Listing 83

```
private async Task GetUserPosition()
{
    await GetPosition();
}
```

Or, if you wanted to write a method that simply returns the detected position to the caller, instead of directly setting the values of the two **TextBlock** controls, the method would have looked like this:

Code Listing 84

```
private async Task<Geoposition> GetPosition()
{
    Geolocator geolocator = new Geolocator();
```

```
    Geoposition geoposition = await geolocator.GetGeopositionAsync();
    return geoposition;
}
```

The difference this time is that, instead of returning just a generic **Task**, it returns a **Task<Geoposition>** object. Then, you would have been able to update the **TextBox** controls directly from the calling method, like in the following example:

Code Listing 85

```
private async void OnGetPositionClicked(object sender, RoutedEventArgs e)
{
    Geoposition geoposition = await GetPosition();
    Latitude.Text = geoposition.Coordinate.Latitude.ToString();
    Longitude.Text = geoposition.Coordinate.Longitude.ToString();

}
```

The dispatcher

When you're working with asynchronous code, typically the code is executed on multiple threads. This way, the UI thread is free to keep the interface fast and responsive. However, sometimes you need to interact with a page's control from a secondary thread. The problem is that, if you try to do it, the application will crash with the following exception:

The application called an interface that was marshalled for a different thread.
(Exception from HRESULT: 0x8001010E (RPC_E_WRONG_THREAD))

The issue occurs because you can't interact with the user interface from a background thread. If you're using the async and await pattern, however, you don't have to deal with this problem. The pattern automatically takes care of returning the result from the secondary thread to the main one. In fact, as you can see in the previous sample, we didn't do anything special to display the user's position on the screen. We've simply retrieved the coordinates using the **GetGeopositionAsync()** method and we've assigned the results to the **Text** property of a couple of **TextBlock** controls.

However, you don't always have the chance to use the async and await pattern. Let's consider the previous sample about using the callback approach:

Code Listing 86

```
private void OnStartGeolocator(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.PositionChanged += geolocator_PositionChanged;
    Debug.WriteLine("Finding the user's position…");
}

void geolocator_PositionChanged(Geolocator sender, PositionChangedEventArgs
args)
```

```
{
    Latitude.Text = args.Position.Coordinate.Latitude.ToString();
    Longitude.Text = args.Position.Coordinate.Longitude.ToString();
}
```

The previous code will generate an exception at runtime. The callback's method, in fact, is executed on a background thread, while the **TextBlock** controls we're trying to update are managed by the UI thread.

For these situations, the Universal Windows Platform offers a class called **Dispatcher**, which takes care of forwarding the operation to the UI thread. Here is the proper way to define the previous sample:

Code Listing 87

```
private void OnStartGeolocator(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.PositionChanged += geolocator_PositionChanged;
    Debug.WriteLine("Finding the user's position…");
}

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

The operations that need to be executed on the UI thread (in our case, assigning the user's position to the **Text** property of a **TextBlock** control) are wrapped inside an anonymous method, which is passed as parameter of the **RunAsync()** method exposed by the **Dispatcher** class. The first parameter represents the priority of the operation: the suggested one is usually **Normal**. This way, the whole callback's method will be executed on a background thread but the operations performed by the **RunAsync()** method will be executed on the UI thread. It's important to forward with the **Dispatcher** just the operations that really need to interact with the page. If, for example, we had done additional operations to the user's position before displaying the coordinates on the page (like converting the coordinates into a civic address), we ought to have performed them outside the dispatcher.

Handling multiple SDK versions

We've already seen this concept in the first chapter of this book. Since Windows 10 has introduced the concept of Windows as a Service, a Windows 10 machine can have multiple

versions of the Universal Windows Platform, since every major update introduces a new SDK version, with an expanded set of APIs and features.

Now, when you create a new Universal project in Visual Studio 2015, you'll be prompted with the following dialog:

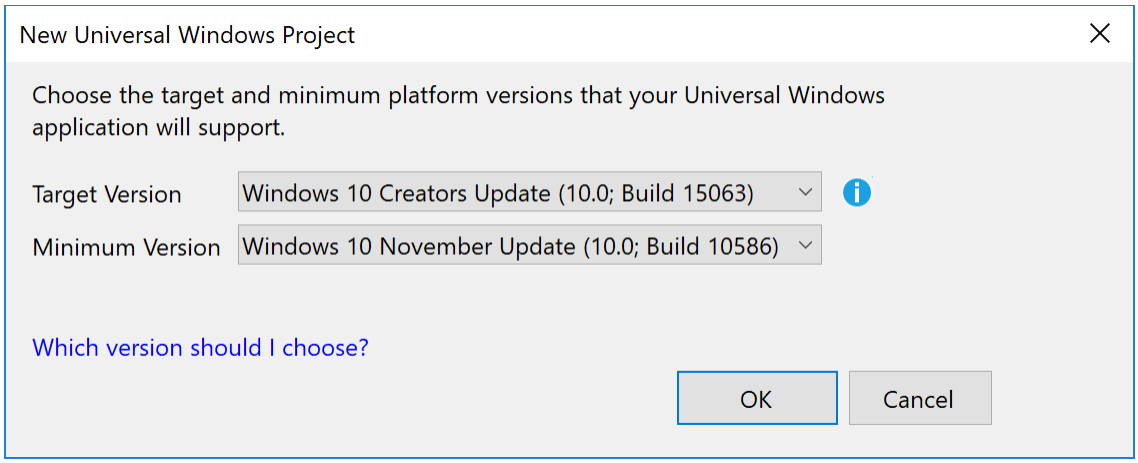


Figure 22: The dialog displayed when you create a new Universal Windows Platform app in Visual Studio 2017.

Target version refers to the set of APIs you want to leverage in your application. For example, if you set as target **Windows 10 Creators Update (10.0; build 15063)**, it means that you'll also be able to get access to the new APIs added in this version.

Minimum version refers to the minimum Windows 10 version you want to support. If you set as minimum **Windows 10 (10.0; build 10586)**, the application will run also on devices with Windows 10 November Update. Old devices that are still using the original Windows 10 version (build 10240) won't be able to install this app, either from the Store or by manually side loading it.

Choosing the best combination depends on the type of project you're working on. For example, let's say that your application relies heavily on a feature called **Applications extensions**, which means that your application can act as container for extensions that can expand it. You can think of them like plugins that can add new features to your application. This feature has been added in the Anniversary Update and, without it, your application wouldn't make much sense. In this case, the best choice is to limit the distribution to users who already have the Anniversary Update. Therefore, you'll set both the **Target** and **Minimum** version to SDK 14393.

On the other side, let's say that you have another application that relies on the Composition APIs, which are a set of classes and methods that we will see in the next book of the series. The set helps to create a better user interface by adding powerful and performant animations. These APIs have been introduced in Windows 10, but they have been further expanded in both the November and the Anniversary Updates. However, animations are hardly a core feature, so it wouldn't be a wise choice to limit the number of users who can download your application just to leverage some additional animations. In a scenario like this, it makes more sense to set the **Minimum version** to SDK 10586 (so that we can provide all the core features to a broader set of users) and the **Target version** to SDK 15063 so that, by leveraging the API detection

approach described in the previous chapter, we can still display the new animations to the users with an updated device.

One important thing to highlight is that the API detection approach works only in C# and not with XAML. This means that, if you're planning to take advantage of some Anniversary Update features managed in XAML, you can't filter them based on the version of the operating system. In this case, you're forced to set the **Minimum version** and the **Target version** to the same SDK if you want to leverage them. A real example of this scenario is the **`x:Bind`** markup expression. As we've seen in this chapter, there are some features (like casting and functions support) that have been added in the Anniversary Update. Even if you set as **Target version** the 14393 SDK, until the **Minimum version** is set to a prior version, you'll always get a build error when you compile the project if you try to leverage one of these new features.

Chapter 3 Creating the User Interface: the Controls

Creating the user interface is probably the most challenging difference compared to a Windows Store application for Windows 8.1 and Windows Phone 8.1. In the previous version of the platform, we could share most of the business logic, but in the end, we were working with two different projects, so it was easy to handle the differences between devices, because we were creating two completely different sets of XAML pages. Of course, this approach was easier to handle, but also more time-consuming, because it required us to create two completely different XAML pages, even if there were probably many similarities between them.

In this chapter, we're going to explore the various controls the Universal Windows Platform includes to create the user interface. Most of these controls are already optimized for the concept of "Universal" and they may show different behaviors and layouts based on the device the app is running on. For example, controls based on the swipe gesture (like **`FlipView`** or **`Hub`**) automatically display a set of buttons on the left and right of the screen when the app is running on a device with a mouse and keyboard to make it easier for the user to move to the next or previous item.

Layout controls

Layout controls are special controls that, most of the time, don't really display anything on the page. Their purpose is to define the layout of the page, and they usually act as a container of other controls.

StackPanel

The **`StackPanel`** control can be used to place the nested controls one below the other, which will automatically adapt to fit all the available space.

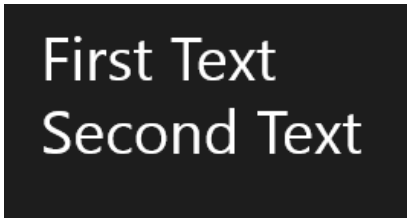


Figure 23: The StackPanel control.

You can also choose to align the nested controls horizontally by setting the **`Orientation`** property to **`Horizontal`**, like in the following sample:

Code Listing 88

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

Grid

The **Grid** control is used to create table layouts where nested controls are organized in rows and columns. Here is a sample code:

Code Listing 89

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*" />
        <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>

    <TextBlock Text="Text placed in the first row of the second column"
Grid.Row="0" Grid.Column="1" />
</Grid>
```

The layout is defined using the **RowDefinitions** and **ColumnDefinitions** properties. Each of them can contain one or more **RowDefinition** or **ColumnDefinition** elements, based on the number of rows and columns you want to create. For every table element, you can specify a fixed height or width, or use relative values like ***** to create proportional sizes, or **Auto**, which adapts the row or column to its content. Using relative values is a very common pattern when designing the user interface of a Universal Windows Platform application. This way, the layout can automatically adapt to the screen size and to the resolution of the device.

In the previous code, you can see an example of all three scenarios:

- The first row has a fixed height of 50 pixels, no matter what the size of the content.
- The second row is set to **Auto**: the row's height will always fit the row's content.
- The two columns are using a proportional approach. We're ideally splitting the width of the screen in three parts: the first column will use 1/3 of the space (**1***), the other one the 2/3 left (**2***).

To define the cell where a control is placed, you need to use two special attached properties, called **Grid.Row** and **Grid.Column**, which can be added to any control. These properties simply contain the row and column numbers where the control is placed, starting from 0 as base index (so, if you've defined two columns, they will be identified by the indexes 0 and 1).

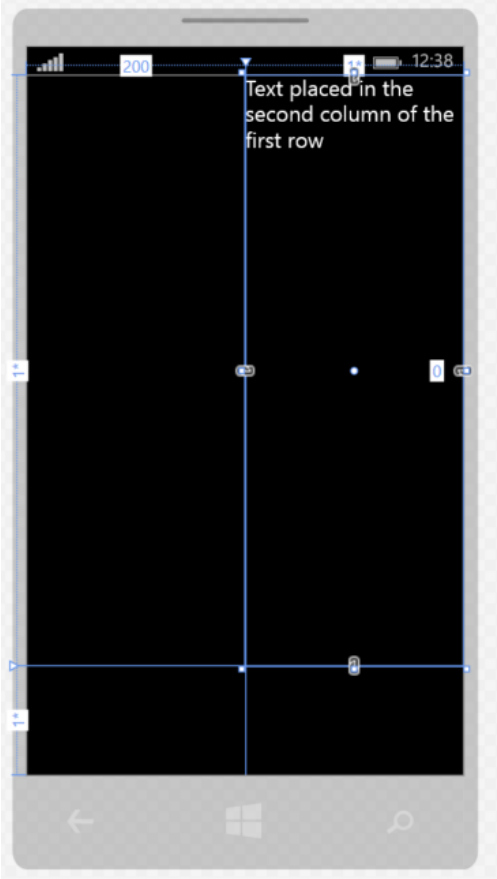


Figure 24: The Grid control.

Canvas

The **Canvas** control gives maximum flexibility to the developer, since it uses a fixed placement. Using the attached properties **Canvas.Top** and **Canvas.Left**, which can be applied to any control, you can set the exact distance (in pixels) where the control should be placed, starting from the top left.

However, you need to be careful using this control. Since it provides a fixed placement, it can't automatically adapt the content to the screen's size and resolution, making hard to create a layout that is pleasant to see both on a small tablet and on a big monitor. Since this is the case, unless you're dealing with a very specific scenario, it's strongly advised not to use it in a Universal Windows Platform application.

Code Listing 90

```
<Canvas Width="640" Height="480" Background="White">
    <Rectangle Canvas.Left="30" Canvas.Top="30" Fill="Red" Width="200"
Height="200" />
</Canvas>
```

The previous samples show a **Rectangle** control placed at the coordinates 30,30 starting from the top left.

VariableSizedWrapGrid

The **VariableSizedWrapGrid** control can automatically split the layout into multiple rows and columns. Unlike in the **Grid** control, you won't have to manually specify the number of rows and columns to create, you'll just have to set the maximum number of items to place in a row or in a column and its size.

Here's sample code to better understand how it works:

Code Listing 91

```
<VariableSizedWrapGrid MaximumRowsOrColumns="2" ItemHeight="200"
ItemWidth="200">
    <Rectangle Fill="Red" />
    <Rectangle Fill="Blue" />
    <Rectangle Fill="Orange" />
    <Rectangle Fill="Green" />
    <Rectangle Fill="Brown" />
</VariableSizedWrapGrid>
```

As you can see, we've set the size of a single item (using the **ItemHeight** and **ItemWidth** properties) and the maximum number of rows and columns to create (using the **MaximumRowsOrColumns** property). Inside the **VariableSizedWrapGrid** control, we've placed five **Rectangle** controls. The result will be that, since we've set two as the maximum number of items, they will be automatically split into two rows.

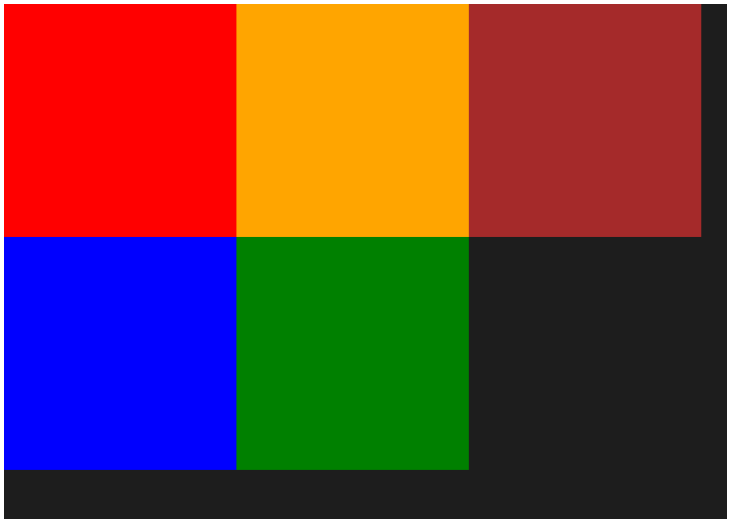


Figure 25: A set of Rectangle controls placed inside a VariableSizedWrapGrid control.

As default behavior, the nested items are split into rows. As you can see from the previous figure, the control has aligned the five rectangles into two rows. If you want to change this behavior and split items based on columns, it's enough to set the **`Orientation`** property to **`Horizontal`**, like in the following sample:

Code Listing 92

```
<VariableSizedWrapGrid MaximumRowsOrColumns="2" ItemHeight="200"
ItemWidth="200" Orientation="Horizontal">
    <Rectangle Fill="Red" />
    <Rectangle Fill="Blue" />
    <Rectangle Fill="Orange" />
    <Rectangle Fill="Green" />
    <Rectangle Fill="Brown" />
</VariableSizedWrapGrid>
```

ScrollViewer

The **`ScrollViewer`** control acts as a container like the previous controls, but it doesn't try to arrange the layout of the nested controls. This means that it needs to be used in combination with other layout controls.

The **`ScrollViewer`** control is used when you need to display content that takes more space than the screen's size. The following sample shows how to manage a long text that doesn't fit the screen. Thanks to the **`ScrollViewer`** control, the user will be able to scroll the text down to keep reading it.

Code Listing 93

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

Border

The **`Border`** control, as the name says, can wrap nested controls inside a border. By using the **`BorderThickness`** and **`BorderBrush`** properties, you can set the border's thickness and color. The following sample shows an **`Image`** control wrapped inside a red border, with a 5-pixel thickness:

Code Listing 94

```
<Border BorderThickness="5" BorderBrush="Red">
    <Image Source="/Assets/Image.jpg"/>
</Border>
```

As default behavior, the **BorderThickness** value is applied to every side of the border. However, you can also customize it by specifying a different thickness for each side (in the following order: left, top, right, bottom), like in the following sample:

Code Listing 95

```
<Border BorderThickness="10, 15, 20, 15" BorderBrush="Red">
    <Image Source="/Assets/Image.jpg"/>
</Border>
```

Figure 26: A Border control with a different thickness for each side.

With the goal of improving performance and reducing the number of controls you must add in the XAML tree, the November Update has added built-in support for borders for many layout controls like **StackPanel** or **Grid**. This way, for example, if you want to apply a border to a **StackPanel**, you don't have to embed it into a **Border** control anymore like in the previous sample, you can leverage the **BorderBrush** and **BorderThickness** properties exposed directly by the control, like in the following sample:

Code Listing 96

```
<StackPanel BorderBrush="Red" BorderThickness="2">
    <TextBlock Text="Text 1" />
    <TextBlock Text="Text 2" />
</StackPanel>
```

RelativePanel

All the controls we've seen so far should be familiar to you if you already have some previous development experience with XAML based technologies, like WPF, Silverlight, or Windows

Store apps. The Universal Windows Platform has added a new layout control, which was born specifically to support adaptive layout scenarios, since it makes it easier to reposition controls placed on a page.

The control is called **RelativePanel** and takes its name from the fact that, for every child control, we can specify:

- The relationship between the control and the panel itself.
- The relationship between one control and the other.

The children controls are simply placed inside the **RelativePanel** one. Then you can leverage a set of attached properties to specify, for each control, its position in relation to the panel or other controls. Since they're attached properties, you can simply use them with any control by using the syntax **RelativePanel.NameOfTheProperty**.

Let's see an example of both approaches.

Relationships with the panel

The **RelativePanel** control allows you to define the position of the children controls relative to the panel itself. For example, you can specify that a control should always been aligned to the bottom or to the right side of the panel. This approach makes it easier to create adaptive layout experiences since, no matter what the size of the screen, the child control will always respect the relationship we've established. Let's see a sample code:

Code Listing 97

```
<RelativePanel>
    <Button Content="Button 1"
            RelativePanel.AlignBottomWithPanel="True"
            RelativePanel.AlignLeftWithPanel="True" />
    <Button Content="Button 2"
            RelativePanel.AlignBottomWithPanel="True"
            RelativePanel.AlignHorizontalCenterWithPanel="True" />
    <Button Content="Button 3"
            RelativePanel.AlignBottomWithPanel="True"
            RelativePanel.AlignRightWithPanel="True" />
</RelativePanel>
```

The **RelativePanel** control offers an attached property (**bool** type) that follows the naming convention **Align*Position*WithPanel**, where position is one of the different sides of the panel. In the previous code, you can see different samples. When the property **RelativePanel.AlignBottomWithPanel** is set to **True**, the control will be placed at the bottom side of the panel; when the **RelativePanel.AlignHorizontalCenterWithPanel** property is set to **True**, the control will be horizontally aligned in the center of the panel.

The advantage of this approach is that, since the position of the controls is relative to the panel, it doesn't matter if the device the app is running on has a big or a small screen, the controls will always stick to the chosen position.

The following images show the previous XAML code in action:

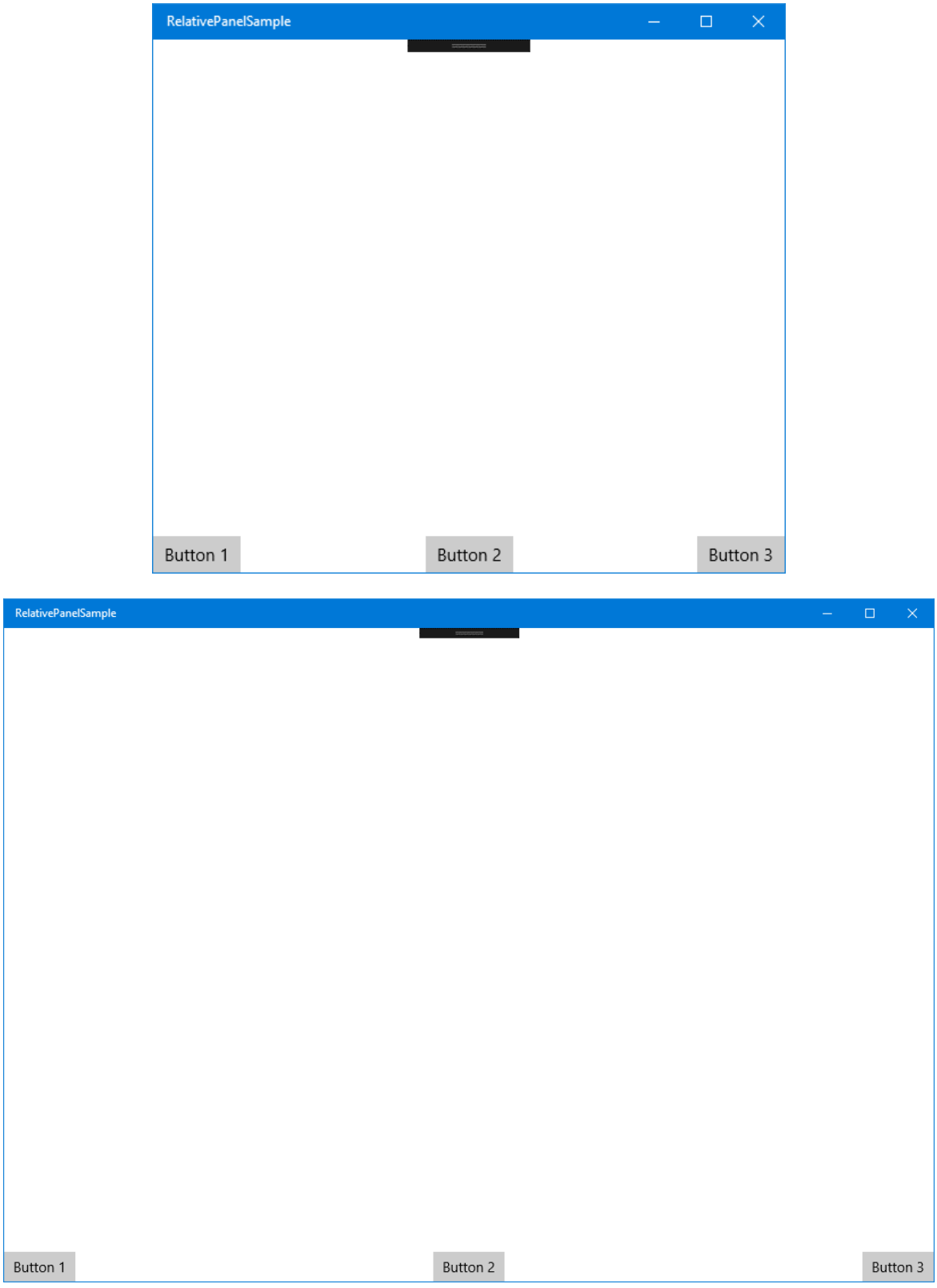


Figure 27: The controls inside the RelativePanel are positioned based on the relation with the panel itself.

As you can see from the two images, despite the application running in two windows with different sizes, the three buttons are always placed in the same position: one on the left, one in the middle, and one on the right. Additionally, all of them are placed on the bottom of the screen. If we are on a desktop and we start resizing the window, the three buttons will continue to respect the same position.

Relationships with other controls

Another powerful feature offered by the **RelativePanel** is the ability to define the positions of children controls in relation to other controls, using the name as a qualifier to identify the relation. Let's see an example first:

Code Listing 98

```
<RelativePanel>
    <Rectangle x:Name="Rectangle1"
               RelativePanel.AlignHorizontalCenterWithPanel="True"
               Width="200" Height="200"
               Fill="Red" />
    <Rectangle x:Name="Rectangle2" Fill="Yellow"
               Width="200" Height="200"
               RelativePanel.RightOf="Rectangle1" />
    <Rectangle Fill="Green"
               x:Name="Rectangle3"
               Width="200" Height="200"
               RelativePanel.AlignHorizontalCenterWith="Rectangle1"
               RelativePanel.Below="Rectangle1" />
</RelativePanel>
```

As you can see, we're exploring two other different types of attached properties offered by the **RelativePanel** control. The first one is **AlignPositionWith**, which allows you to align a control in relation to another control's position. In this sample, the **Rectangle** control identified by the name **Rectangle3** is aligned horizontally based on the position of the control with name **Rectangle1**.

The other kind of supported relationship is related to the control's position instead of the alignment. We can leverage attached properties like **RelativePanel.Below**, **RelativePanel.RightOf**, etc., to define that a control should be rendered in a specific place, based on the position of another control. In the previous sample, the **Rectangle** identified by the name **Rectangle2** is placed to the right of the **Rectangle** identified by the name **Rectangle1**; additionally, the **Rectangle3** control is placed below **Rectangle1**.

The image below shows the result of this code:

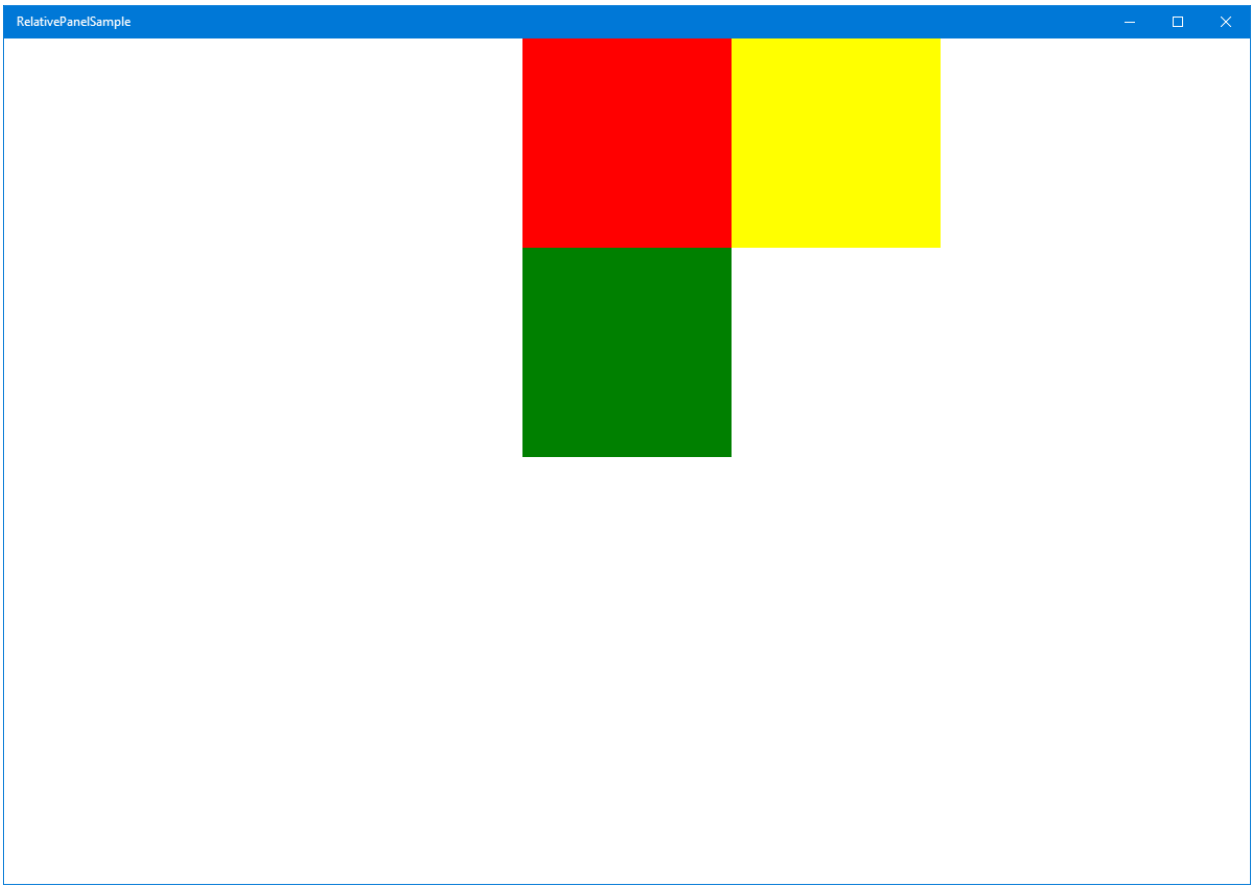


Figure 28: The controls inside the RelativePanel are positioned based on relationships with the other controls.

You'll notice, also, that you have the freedom to mix both kinds of approaches. In the previous sample code, we're leveraging the relationships both between the panel and the children controls (**`Rectangle1`** is horizontally centered in the panel) and among the controls themselves (**`Rectangle2`** is placed to the right of **`Rectangle1`**).

The freedom you have to change relationships among the controls inside a **`RelativePanel`** makes it a key control when it comes to adapting the layout of the application to different screens. As we're going to see in detail when I talk more deeply about adaptive layout in the second book of the series, one of the most-used techniques is called **reposition**, which means that we move the position of the controls based on the size of the screen. For example, we could decide that, when the screen is small, the three rectangles should be displayed one below the other instead of in the current layout. This goal is very easy to achieve with the **`RelativePanel`** control, since it's enough to change the relationships among the children (for example, every **`Rectangle`** could leverage the attached property **`RelativePanel.Below`** so that we can place all of them one below the other).

Output controls

In this category, we'll see all the controls used to display information to the user, like texts, images, etc.

TextBlock

The **TextBlock** control is most widely used to display text on a page. The most important property is called **Text** and contains the text that will be displayed. In addition, you can customize the text's look and feel, thanks to properties like **FontSize** (to change the dimensions) or **FontStyle** (to change the font's type). Another property often used is called **TextWrapping**. When it's set to **Wrap**, it makes sure that the text is wrapped in multiple lines in case it's too long.

Code Listing 99

```
<TextBlock Text="This is a long text" TextWrapping="Wrap" />
```

Another interesting property is called **TextTrimming**. When this feature is enabled, if the text is too long, it will be automatically trimmed and ellipses will be added at the end, so the user can understand that the text is cut off. You can apply two types of trimming: **CharacterEllipsis** (the text is cut at character level) or **WordEllipsis** (the text is cut at word level).

Code Listing 100

```
<TextBlock Text="This is a trimmed text" TextTrimming="CharacterEllipsis"
/>
```

You also have the chance to apply different styles to various parts of the text without having to split it into multiple **TextBlock** controls, thanks to the **Run** control. It's enough to add one or more **Run** controls inside a **TextBlock** control; each of them will contain a part of the text. You can also use the **LineBreak** control if you want to manually split the text into multiple lines.

Code Listing 101

```
<TextBlock>
    <Run Text="First line" />
    <LineBreak />
    <Run Text="Second line in bold" FontWeight="Bold" />
</TextBlock>
```

RichTextBlock

The **RichTextBlock** control is a more powerful version of **TextBlock**, since it offers more flexibility to the developer. You'll be able to create complex layouts by:

- Splitting the text into multiple paragraphs using the **Paragraph** control.

- Changing the style of a text portion using tags likes **Bold**, **Italic**, or **Underline**.
- Adding other XAML controls using the **InlineUIContainer** control.

Let's see a sample XAML code:

Code Listing 102

```
<RichTextBlock>
    <Paragraph>
        <Bold>This is a bold text</Bold>
    </Paragraph>
    <Paragraph>
        <Italic>This is an italic text</Italic>
    </Paragraph>
    <Paragraph>
        <Underline>This is an underlined text</Underline>
    </Paragraph>
    <Paragraph>
        <InlineUIContainer>
            <Image Source="/Assets/image.jpg" Width="200" />
        </InlineUIContainer>
    </Paragraph>
</RichTextBlock>
```

The **RichTextBlock** control contains multiple **Paragraph** controls, each one with some text that is formatted in different ways. In addition, we've used the **InlineUIContainer** control to display an image at the bottom of the text.

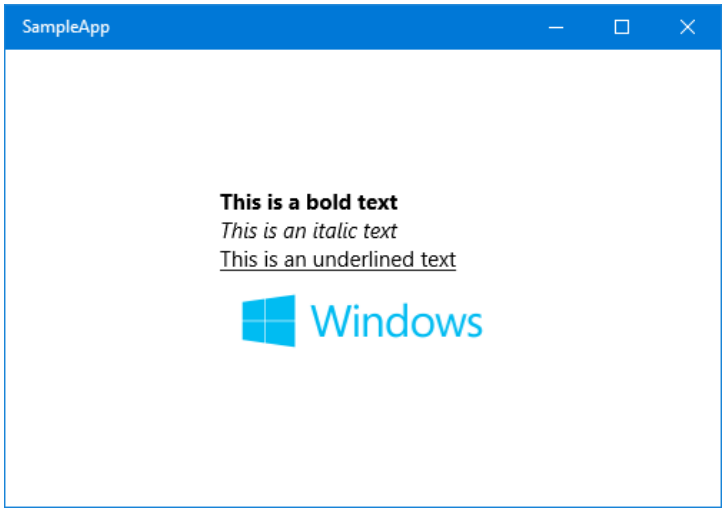


Figure 29: The RichTextBlock control.

Image

The **Image** control is used, as the name says, to display an image on a page. The image path is set using the **Source** property, which supports different kinds of sources. The image can be stored on a remote address, in the application's package, or in the local storage. We'll see in detail in the next book of the series the different ways we can access the various kinds of storages using a URL address.

The following sample code shows an **Image** control that displays a remote image:

Code Listing 103

```
<Image Source="http://www.website.com/image.png" />
```

The **Image** control also offers built-in support for the crop feature, so that you can easily display just a small part of the image. This feature is implemented using the **Clip** property, which accepts a **RectangleGeometry** control, defining the crop area, like in the following sample:

Code Listing 104

```
<Image Source="http://www.website.com/image.png">
    <Image.Clip>
        <RectangleGeometry Rect="0, 0, 100, 100" />
    </Image.Clip>
</Image>
```

The crop is defined with four values. The first two identify the X and Y coordinates of the area's starting point (0,0 represents the top left corner of the image) while the other two represent the size of the crop area (in the sample, the rectangle's size is 100x100).

Another important property is called **Stretch**, which is used to define how the image will fill the available space:

- **Uniform**: the default value, the image is resized to fit the control's size by keeping the original ratio, so that it doesn't look distorted.
- **Fill**: the image is stretched to use all the available space, even if it means ignoring the ratio and creating a distorted effect if the control has a different size.
- **UniformToFill**: a combination of the previous modes, the image is automatically cropped to create a new image that keeps the same ratio of the original one but still fills all the available space.
- **None**: the image is displayed using the original size, regardless of the size of the control.

Input controls

In this category, we'll see in detail all the main controls that are used to receive input from the user.

TextBox and PasswordBox

The **TextBox** control is the simplest available to get text input from the user. It works in a similar way to the **TextBlock** control, except that the **Text** property isn't used just to set the text to display, but also to grab the text inserted by the user.

A useful property offered by the control is called **PlaceholderText**. It defines a placeholder text that can be displayed inside the box as a hint for the users, so they can better understand what kind of input we're expecting. As soon as the user starts to type text into the box, the placeholder text will disappear.

Another important scenario to keep in mind is that Universal Windows Platforms apps can be used on a device with a touch screen. In this case, instead of a real keyboard, the user will insert the text using a virtual keyboard. As developers, we have the chance to customize the virtual keyboard so that it's optimized for the kind of input we're asking for from the user. This customization is achieved using the **InputScope** property, which can assume many values, like:

- **URL** to display a keyboard optimized to enter web addresses.
- **Number** to display a keyboard optimized to enter numbers.
- **EmailSmtpAddress** to display a keyboard optimized to enter an email address.

We can also enable or disable the autocorrection feature (using the **IsSpellCheckEnabled** property) or the text prediction one, which suggests words to users while they're typing on the keyboard (using the **IsTextPredictionEnabled** property).

Here is a sample code to define a **TextBox** control:

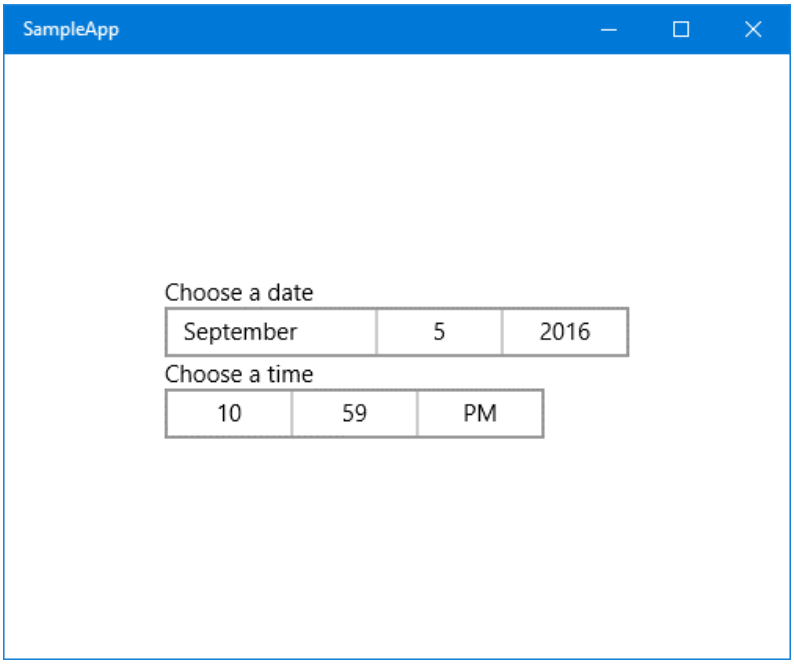
Code Listing 105

```
<TextBox IsSpellCheckEnabled="True" IsTextPredictionEnabled="True"
PlaceholderText="Placeholder Text" Text="Input text" />
```

The Universal Windows Platform also offers a custom **TextBox** control called **PasswordBox**. It works exactly like a **TextBox** but, by default, it replaces the inserted text with dots so that other people who are looking at the screen can't read it. As you can imagine, it's perfect for scenarios where we're asking for sensitive data, like passwords or credit card numbers. Since the text is automatically hidden, the **PasswordBox** control doesn't offer as many ways to customize it as the **TextBox** control. The only important available option is called **IsPasswordRevealButtonEnabled**; when it's set to **True**, it adds a button at the end of the box that, when it's pressed, temporarily displays the text clearly so the user can check that they inserted the correct password.

DatePicker and TimePicker

The **`DatePicker`** and **`TimePicker`** controls are used to manage dates and times in an application.



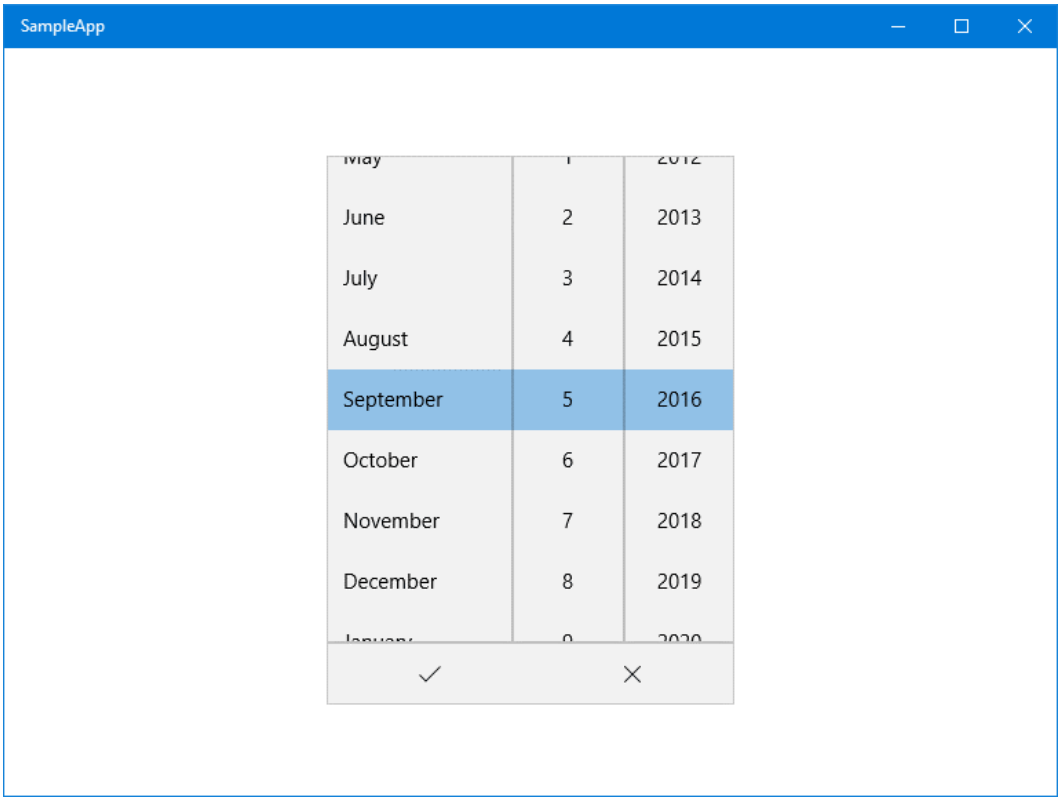


Figure 30: The top image shows both the DatePicker and the TimePicker controls. The bottom image shows the expanded DatePicker.

As default behavior, the **DatePicker** control displays all the date fields: the day, the month, and the year. It's possible to hide one of them by using the properties called **YearVisible**, **MonthVisible**, or **DayVisible**. If one or more fields is hidden, the control will automatically return the current value to the application (so, for example, if you hide the year, the control will return the current year).

Code Listing 106

```
<DatePicker x:Name="Date" YearVisible="False" />
```

You can also define the date range by using the **MinYear** and **MaxYear** properties. This can be useful for many scenarios, for example, if you're asking the user's birthday; in this case, it would be useless to display years prior to 1900 or later than the current year. However, these properties can't be set in XAML, only in code-behind, since their type is **DateTimeOffset**, which can't be expressed in XAML.

Code Listing 107

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DateTime minYear = new DateTime(1900, 1, 1);
    Date.MinYear = new DateTimeOffset(minYear);
    Date.MaxYear = new DateTimeOffset(DateTime.Now);
}
```

If you want to retrieve the date selected by the user in the code-behind, you need to use the **Date** property, like in the following sample, which shows the date using a pop-up message:

Code Listing 108

```
private async void OnGetDateClicked(object sender, RoutedEventArgs e)
{
    DateTimeOffset selectedDate = Birthday.Date;
    MessageDialog dialog = new MessageDialog(selectedDate.ToString());
    await dialog.ShowAsync();
}
```

The **TimePicker** control works in a similar way but, unlike the **DatePicker** one, it's composed of only two elements: hours and minutes. You can customize the time range using the **MinuteIncrement** property. This way, instead of displaying all the possible values for the minute field (from 0 to 59), it will show only the specified range.

Let's look at, for example, the following implementation:

Code Listing 109

```
<TimePicker MinuteIncrement="15" />
```

By using the previous XAML code, the minute drop-down will display just the values 00, 15, 30, and 45.

The value selected by the user in the **TimePicker** control is stored in the **Time** property using a **TimeSpan** object. The following sample shows a method that displays the selected time converted into hours to the user using a pop-up message:

Code Listing 110

```
private async void OnGetDateClicked(object sender, RoutedEventArgs e)
{
    TimeSpan timeSpan = StartTime.Time;
    MessageDialog dialog = new
MessageDialog(timeSpan.TotalHours.ToString());
    await dialog.ShowAsync();
}
```

Both controls also support setting a header, which is displayed above the control, by leveraging the **Header** property.

CalendarDatePicker and CalendarView

CalendarDatePicker and **CalendarView** are two new controls added in Windows 10 that make it easier for developers to work with dates.

The goal of the **CalendarDatePicker** is the same as the **DatePicker** we've previously seen. It allows the user to choose a date. However, in this case, it offers a more pleasant user experience. We can't limit the user selection to just the subset of info that comprise a date (like just the day and the month) because it displays a full calendar view, like the one you see when you click on the current date and time in the Windows 10 taskbar.

By default, the control displays just a placeholder that invites the user to select a date (which can be customized using the **Placeholder** property). Once the user taps on it, the control displays the full calendar below the placeholder, as you can see in the following image.

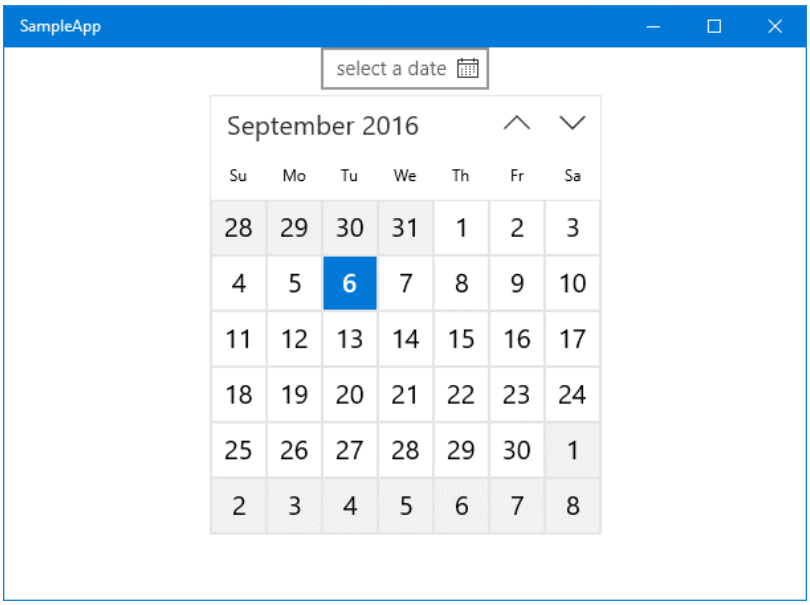


Figure 31: The CalendarDatePicker control.

The control can be customized in multiple ways and even deeply restyled thanks to the **`CalendarViewStyle`** property. A couple of additional interesting options to customize it are:

- **`DisplayMode`**: can be set to **`Year`**, **`Decade`**, or **`Month`**. By default, the calendar picker opens, displaying the current month with all its days. By changing this property to another value, you can change the startup experience. For example, by setting it to **`Year`**, the calendar will open and display a list of the latest years, and then will switch to the monthly view only once the user has chosen a specific year.

- **`IsTodayHighlighted`**: a **`bool`** property that, when set to **`True`**, automatically highlights the current day in the calendar. This property is enabled by default and you can see an example in the previous image.

- **`IsOutOfScopeEnabled:`** another bool property enabled by default. When it's set to **`True`**, all the items out of scope from the current selection are displayed with a different background. Also in this case, the previous image shows a real example. Since September is the selected month, all the days that belong to the previous (August) or the next (October) months are displayed with a gray background instead of a white one.

From an interaction point of view, the **`CalendarDatePicker`** control works in a very similar way to the **`DatePickerOne`**. You can restrict the displayed date range by leveraging the **`MinDate`** and **`MaxDate`** properties and, from code-behind, you can get the selected date using the **`Date`** property.

Here is a sample XAML code on how to insert it into your pages:

Code Listing 111

```
<CalendarDatePicker  DisplayMode="Month"  IsTodayHighlighted="True"
IsOutOfScopeEnabled="True" />
```

The **CalendarView** control, from a user experience point of view, offers the same UI as the **CalendarDatePicker** one, with many customization options in common (like the opportunity to highlight the current day or to change the background color of out-of-scope days).

However, it isn't meant to be just a simple selector. In fact, there are some major differences:

- By default, the **CalendarDatePicker** displays just a selector and the full calendar is displayed only when the user taps on it. The **CalendarView**, instead, always shows the full calendar.
- The **CalendarView** offers a deep way to customize every part of the calendar, like the style of the out-of-scope days, of the current day, etc.
- By leveraging the **SelectionMode** property, you can allow users to select multiple dates instead of just one.

Due to these differences, the way you retrieve the selected date is also different:

- Since the user can choose multiple dates, they are all stored in a property called **SelectedDates**, which is a collection. In cases of single selection, it will contain just one element.
- If you want to intercept when the user has selected one or more dates, you can leverage the **SelectedDatesChanged** event.

The following sample shows how to declare in XAML a **CalendarView** control that invokes the **SelectedDatesChanged** event every time the user clicks on a date:

Code Listing 112

```
<CalendarView
x:Name="Calendar"
SelectedDatesChanged="OnSelectedDates" />
```

Here is, instead, how the definition of the event handler looks:

Code Listing 113

```
private async void OnSelectedDates(CalendarView sender,
CalendarViewSelectedDatesChangedEventArgs args)
{
    DateTimeOffset selectedDate = Calendar.SelectedDates.FirstOrDefault();
    MessageDialog dialog = new MessageDialog(selectedDate.ToString("d"));
    await dialog.ShowAsync();
}
```

By leveraging the **SelectedDates** property exposed by the **CalendarView** control, we simply retrieve the first result in the collection (by default, the control is configured to support single date selection) and we display it to the user with a dialog using the **MessageDialog** class.

Another important feature offered by **CalendarView** control is that we can customize the look and feel of each day based on our logic. For example, if we are writing a birthday reminder application, we could highlight in a different way the days in which one of our friends has a birthday. To reach this goal, we need to leverage another event offered by the control called **CalendarViewDayItemChanging**, which is triggered every time the XAML tree is rendering one of the visible calendar's cells.

Code Listing 114

```
<CalendarView
x:Name="Calendar"
CalendarViewDayItemChanging="OnCalendarViewDayItemChanging" />
```

Here is what the event handler looks like:

Code Listing 115

```
private void OnCalendarViewDayItemChanging(CalendarView sender,
CalendarViewDayItemChangingEventArgs args)
{
    if (args.Item.Date.Date == new DateTimeOffset(new DateTime(2016, 9,
10)))
    {
        args.Item.Background = new SolidColorBrush(Colors.Red);
    }
}
```

As you'll notice, the event handler includes a parameter of the **CalendarViewDayItemChangingEventArgs** type, and contains a property called **Item**, a **CalendarViewDayItem** type. This object represents the cell that is being rendered and, as such, contains all the information about it, both from a visual (like the background or the font size) and a logic (like the represented date) point of view.

This being the case, we can combine both kinds of information to achieve our goal. In the previous example, we use the **Date** property to check if the current rendered date is a special one that we need to handle (for example, it's the birthday of a friend) and then we can leverage the UI properties to change its look and feel (in this sample, we leverage **Background** to change the color to red using a **SolidColorBrush**).

The following image shows you the result, assuming that 10 Sept., 2016, is the special date we want to highlight. Of course, a real application would have a more complex logic, so we would probably have to handle not just one date, but a list of them, and they would be stored in a more complex infrastructure, like a database or a REST service.

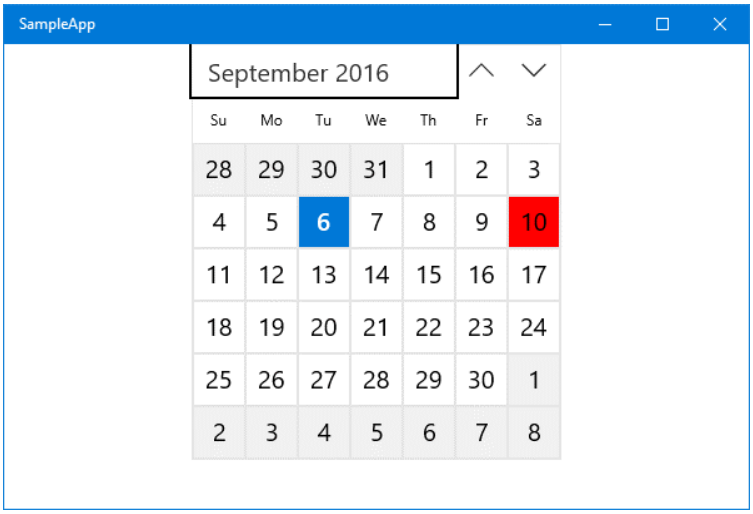


Figure 32: The CalendarView controls shows a date with a different look and feel than the others.

Button and ToggleButton

The **Button** control is the simplest one available to interact with a user. When it's pressed, it raises the **Click** event, which you can manage in code-behind. The button's content is defined by the **Content** property, which can contain a simple text, like in the following sample:

Code Listing 116

```
<Button Content="Click me" Click="OnButtonClicked" />
```

However, the **Content** property can also be expressed with the extended syntax. In this case, you can use any XAML control to define the button's layout. The following sample shows how to define a **Button** control using an image:

Code Listing 117

```
<Button Click="OnButtonClicked">
    <Button.Content>
        <Image Source="image.png" />
    </Button.Content>
</Button>
```

The Universal Windows Platform also offers another kind of button control, called **ToggleButton**. It works and behaves like a regular button, but it's able to manage two different kind of states: enabled and disabled. Consequently, the control also offers a property called **IsChecked**, of the **bool** type. When the button is enabled, its value is **true**; otherwise, the value is **false**.

The following sample code shows a simple usage of the control:

Code Listing 118

```
<ToggleButton Content="This is a CheckedButton" IsChecked="True" />
```

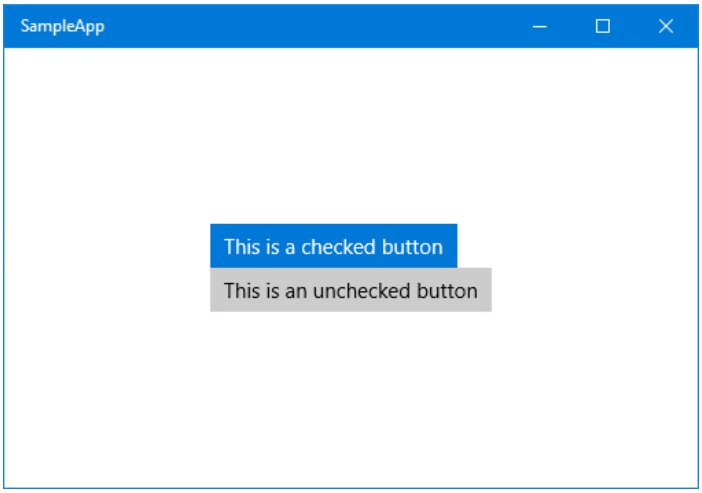


Figure 33: The two different states that a ToggleButton control can assume.

RadioButton and CheckBox

The **RadioButton** and **CheckBox** controls are very similar and are both used to offer multiple choices to the user. They share many features, like:

- The text displayed near the control is set using the **Content** property.
- To check if the control has been selected, you need to use the **IsChecked** property.
- If you want to intercept when the user taps on the control, you can use the **Checked** and **Unchecked** events.

The biggest difference between the **RadioButton** and **CheckBox** controls is that the former can be defined as part of a group. In this case, the user will be able to enable only one of them. If they try to enable another one, the previously enabled control will be automatically deactivated. The **CheckBox** control doesn't apply this logic: the user can enable as many of them as they like.

The **RadioButton**'s grouping is achieved using the **Group** property. By assigning to a set of **RadioButton** controls the same value, the user will be able to enable just one of them at a time. Here is a sample on how to use both controls:

Code Listing 119

```
<StackPanel>
    <CheckBox Content="First option" />
    <CheckBox Content="Second option" />
    <CheckBox Content="Third option" />

    <RadioButton Content="First option" GroupName="Options" />
```

```
    <RadioButton Content="Second option" GroupName="Options" />
    <RadioButton Content="Third option" GroupName="Options" />
</StackPanel>
```

Inking

One of the most innovative features of Windows 10, which has been expanded release by release, is inking support. Most of the 2-in-1 devices available on the market (like the Surface Pro) support digital pens so that the user can write on the screen of her device like she would do on a sheet of paper. The Universal Windows Platform includes a set of controls and APIs to add inking support in your application.

The basic one is called **InkCanvas** and it's simply a canvas where the user, using a digital pen, can write and draw anything. Adding it in a page is simple, as you can see in the following sample:

Code Listing 120

```
<StackPanel>
    <InkCanvas x:Name="Ink" Width="1200" Height="500"  />
</StackPanel>
```

The **InkCanvas** control includes a property called **InkPresenter**, which is fundamental to configure the behavior of the canvas. By default, the **InkCanvas** control works only with a digital pen (like the one that comes with the Surface Pro). If you try to draw something with your finger or with a mouse, nothing will happen. However, you can change this behavior thanks to the **InputDeviceTypes** property of the **InkPresenter** object, like in the following sample:

Code Listing 121

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Ink.InkPresenter.InputDeviceTypes = CoreInputDeviceTypes.Mouse
        | CoreInputDeviceTypes.Pen | CoreInputDeviceTypes.Touch;
}
```

With the previous code, when the page is loaded, we allow the user to draw on the canvas with any supported method: pen, mouse, and touch screen. However, as we're going to see in the next sections, the **InkPresenter** is much more than just a way to configure the **InkCanvas** control. It's the primary access to interact with the canvas and to perform advanced operations.

Storing the content of the canvas in an image

One of the **InkPresenter** features allows you to save the content of an **InkCanvas** control into a GIF image. The **InkPresenter** object, in fact, grants access to all the available strokes, which is everything that has been drawn into the canvas. To achieve this, we have added a button to the page whose **Click** event is connected to the following event handler:

Code Listing 122

```
private async void OnSaveImage(object sender, RoutedEventArgs e)
{
    IReadOnlyList<InkStroke> strokes =
Ink.InkPresenter.StrokeContainer.GetStrokes();
    if (strokes.Count > 0)
    {
        FileSavePicker savePicker = new FileSavePicker();
        savePicker.SuggestedStartLocation =
            PickerLocationId.PicturesLibrary;
        savePicker.FileTypeChoices.Add(
            "GIF",
            new List<string> { ".gif" });
        savePicker.DefaultFileExtension = ".gif";
        savePicker.SuggestedFileName = "InkSample";

        // Show the file picker.
        StorageFile file = await savePicker.PickSaveFileAsync();

        if (file != null)
        {
            IRandomAccessStream stream = await
file.OpenAsync(FileAccessMode.ReadWrite);
            // Write the ink strokes to the output stream.
            using (IOutputStream outputStream =
stream.GetOutputStreamAt(0))
            {
                await
Ink.InkPresenter.StrokeContainer.SaveAsync(outputStream);
                await outputStream.FlushAsync();
            }
            stream.Dispose();
        }
    }
}
```

The first step is to check if the user has actually drawn something in the canvas. We do this by using the **`InkPresenter`** object and by calling the **`GetStrokes()`** method of the **`StrokesContainer`** property. This method will return a collection of all the strokes drawn in the canvas. We move on with the saving procedure only if the collection actually contains at least one element.

The rest of the code leverages the Storage APIs to ask the user where they want to save the image file and to effectively write the content of the canvas in the file's stream. In this chapter, I won't describe in detail the usage of the storage APIs, like **`FileSavePicker`** or **`StorageFile`**, but they will be the main topic of one of the chapters of the second book. For this demo, you just need to know two important things:

- The **`FileSavePicker`** API allows the user to choose a folder in which to save a file. With this approach, the user will be able to choose a folder and a name to which he can save

the content of the canvas. The default options (configured using the **SuggestedStartLocation** and **SuggestedFileName** properties) are to create a file called **InkSample.gif** inside the Pictures library of the computer.

- The **StorageFile** class, the **OpenAsync()** method, and the **IRandomAccessStream** interface are used to get write access to the stream of the file selected by the user, so that we can store the content of the canvas.

We again use the **InkPresenter** object when we are ready to write the data into the file selected by the user. The **StrokeContainer** property offers another method called **SaveAsync()**, which requires as parameter the stream to write the data to.

At the end of the operation, we will have on our device a GIF file with the content of the **InkCanvas** control, no matter if it was a drawing, a text, etc.

Recognizing text and shapes

The **InkCanvas** control can be used to write not just drawings, but also text and shapes. The Creators Update has added a new class called **InkAnalyzer** to perform ink recognition, so that the control can interpret the handwriting of the user and decode it into text or shapes.

To accomplish this task, I've added to the page a couple of additional controls: a **Button** to perform the recognition and a couple of **TextBlock** controls to display the number of identified results and the list of them.

Code Listing 123

```
<StackPanel Margin="12">
    <InkCanvas x:Name="Ink" Width="1200" Height="500"  />
    <Button Content="Recognize" Click="OnRecognizeInk" />
    <TextBlock x:Name="Results" />
</StackPanel>
```

Here is the code that is invoked when the user clicks on the **Button** control:

Code Listing 124

```
private async void OnRecognizeInk(object sender, RoutedEventArgs e)
{
    InkAnalyzer analyzer = new InkAnalyzer();
    var strokes = Ink.InkPresenter.StrokeContainer.GetStrokes();
    if (strokes.Count > 0)
    {
        analyzer.AddDataForStrokes(strokes);
        var result = await analyzer.AnalyzeAsync();
        if (result.Status == InkAnalysisStatus.Updated)
        {
            Results.Text = analyzer.AnalysisRoot.RecognizedText;
        }
    }
}
```

Recognition is performed by the **`InkAnalyzer`** class, which belongs to the **`Windows.UI.Input.Inking.Analysis`** namespace. First you need to load, inside the object, the strokes you want to recognize, by calling the **`AddDataForStrokes()`** method. In the previous sample, we can get the collection of all the strokes drawn by the user by calling the **`GetStrokes()`** method on the **`StrokeContainer`** collection of the **`InkPresenter`** object.

Then we call the **`AnalyzeAsync()`** method and, if we get in the **`Status`** property of the result that something has changed (thanks to the **`InkAnalysysStatus`** enumerator), we can access to the **`AnalysisRoot.RecognizedText`** property to get the text that has been identified.

The most interesting thing of this method is that it can recognize not just texts, but also shapes, as you can see from the following image:

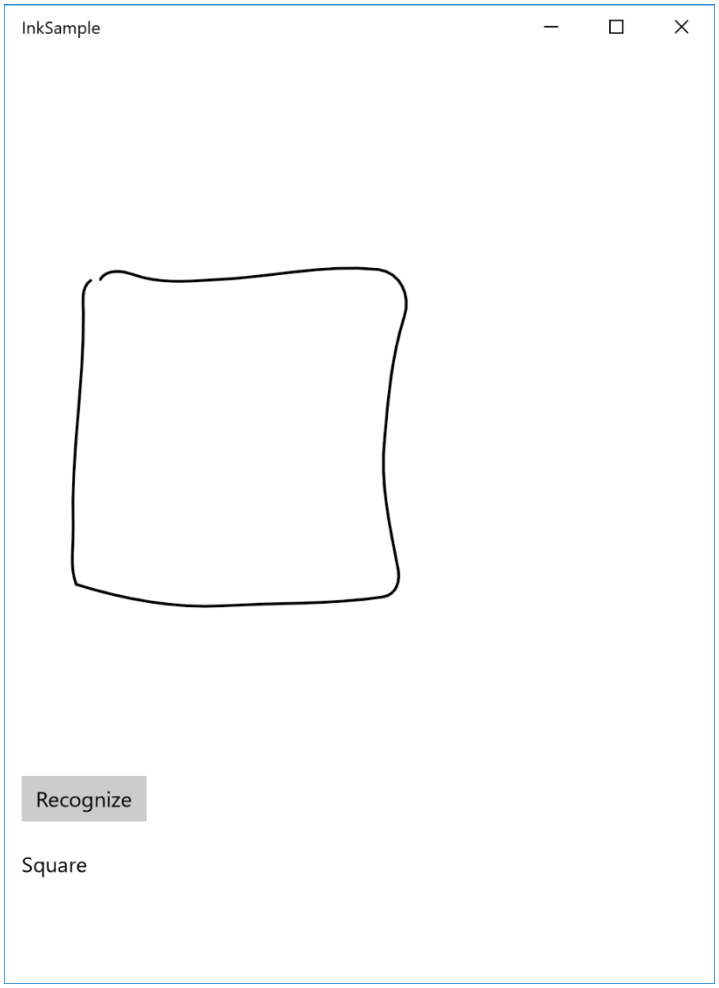


Figure 34: An InkCanvas with a square and, below, the name of the shape identified using the InkAnalysis class

Empowering the canvas with the InkToolbar

The Anniversary Update has added a new control that makes it easier to create a richer experience for the user by providing a set of tools to perform advanced operations with the

InkCanvas control, like changing the stroke color, using a ruler, etc. It's a very powerful control, yet very simple to add, as you can see in the following sample:

Code Listing 125

```
<StackPanel>
    <InkToolbar TargetInkCanvas="{x:Bind Ink}" />
    <InkCanvas x:Name="Ink" Width="1200" Height="500"  />
</StackPanel>
```

As you can see, it's enough to add an **InkToolbar** control in the page and, by using the **x:Bind** markup expression we've learned in the previous chapter, we specify the name of the **InkCanvas** control we want to connect. That's all! Now the users will see a toolbar that will allow them to customize the drawing experience, like:

- Changing the color and the thickness of the pen.
- Using different kinds of pencils.
- Drawing straight lines using a drawer, which can be moved around the canvas using your fingers.

The following image shows the **InkToolbar** placed at the top of the canvas:

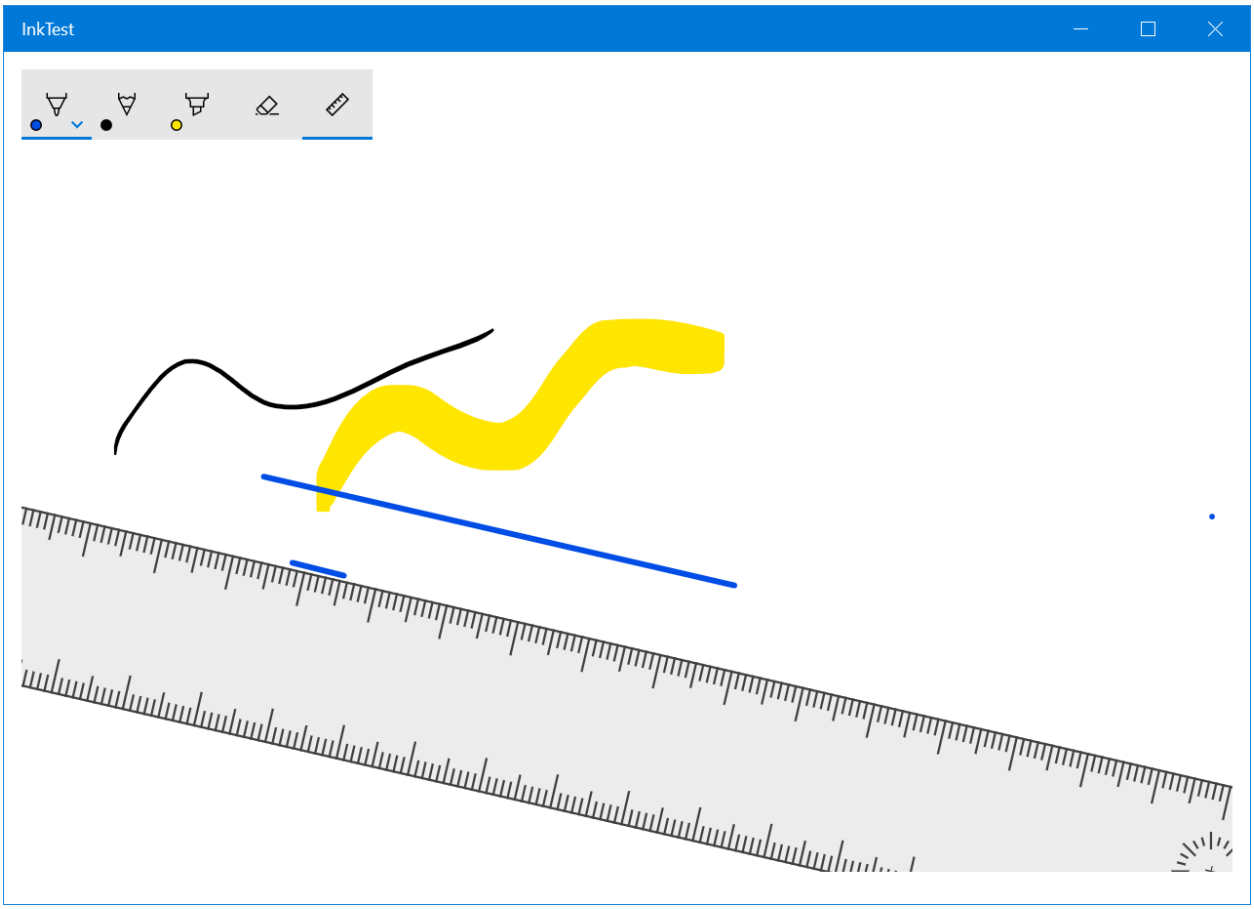


Figure 35: The InkToolbar control allows advanced customization of the drawing experience.

Show the operation status

A very common requirement when you develop an application is to show to the user the status of an operation in progress. We need to notify them that something is going on and, until the operation is finished, the application may not be fully ready to use. The Universal Windows Platform includes two controls to achieve this goal.

ProgressRing

The **ProgressRing** control is used for loading operations that are preventing the user from interacting with the application. Until the operation is finished, there isn't anything else for the user to do, so they just need to wait (for example, a news reader is loading the latest news and, until the operation is completed, the user doesn't have any content to interact with). The control simply displays a spinning ring, which will notify the user that an operation is in progress.

Using this control is very simple. The animation is controlled by the **IsActive** property, which is a **bool** type. When its value is **true**, the progress ring will spin; when it's **false**, the progress ring will be hidden. Typically, you're going to show it before the operation starts; then, you'll hide it once the job is finished. Here is a sample XAML declaration of this control:

Code Listing 126

```
<ProgressRing x:Name="Progress" />
```

Here is, instead, how you ideally manage a loading operation in code (for example, downloading some data from Internet) using a **ProgressRing** control:

Code Listing 127

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Progress.IsActive = true;
    //start loading the data
    Progress.IsActive = false;
}
```

ProgressBar

The **ProgressBar** control, unlike the **ProgressRing**, can be used in scenarios where the operation happens in the background and the user can keep interacting with the application even while the operation is in progress. The **ProgressBar** control is rendered with a bar that is filled using the **Value** property. You can assign a numeric value from 0 (empty bar) to 100 (full bar). This feature also makes the control useful for operations where you can estimate its status, like downloading a file from the Internet.

For example, during a download operation you can determine how many bytes have been downloaded of the total file size and calculate the percentage to set in the **Value** property. This way, the user can be continuously updated on the download status.

Otherwise, the **ProgressBar** control can be used to notify the user that an operation is in progress without displaying the exact status. By setting the **IsIndeterminate** property to **true**, the bar will be replaced by a series of dots that will continuously move from the left to the right of the screen. This way, you'll achieve a user experience closer to the one offered by the **ProgressRing** control:

Code Listing 128

```
<ProgressBar x:Name="Progress" IsIndeterminate="True" />
```

Displaying collections of data

One of the most common requirements in mobile applications is to display a collection of data. We've already seen in Chapter 2 how this scenario can be easily implemented using binding and data templates. In this section, we'll see the most important controls available in the Universal Windows Platform to display collections.

GridView and ListView

GridView and **ListView** are the two controls most used to display data collections in Universal Windows Platform apps, since they're both able to provide a look and feel that is consistent with the platform's guidelines. Both controls offer the same features and properties. The main difference is that the **GridView** control can be scrolled horizontally, using a grid structure; the **ListView** control, instead, is rendered using a traditional list that the user can scroll from top to bottom.

Showing flat collections

The simplest way to use these controls is to display flat data collections. In this case, they behave like every other standard control to display lists, which means:

- You need to define a **DataTemplate** for the **ItemTemplate** property, which defines the layout used to render each element of the collection.
- You need to assign the data collection you want to display to the **ItemsSource** property.

You need to manage the item selected by the user using one of the ways that will be detailed later. The following XAML code shows a sample definition of a flat collection displayed with a **ListView** control:

Code Listing 129

```
<ListView x:Name="List" SelectionChanged="List_OnSelectionChanged">
    <ListView.ItemTemplate>
        <DataTemplate>
```

```
            <StackPanel>
                <TextBlock Text="{Binding Path=Name}" />
                <TextBlock Text="{Binding Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

The **ItemTemplate** of this list is configured to display a collection of objects with two properties called **Name** and **Surname**. Here is a sample definition of this object:

Code Listing 130

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

And here is how, in the code-behind, we create a sample collection of **Person** objects and we assign it to the **ListView** control:

Code Listing 131

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    List<Person> people = new List<Person>
    {
        new Person
        {
            Name = "Matteo",
            Surname = "Pagani"
        },
        new Person
        {
            Name = "Angela",
            Surname = "Olivieri"
        }
    };

    List.ItemsSource = people;
}
```

The **GridView** control is widely used in Universal Windows Platform apps because it makes it easier to create adaptive layout experiences. The control is able, in fact, to automatically implement a **reflow** experience, which means that the items are automatically moved to new rows or moved back to the original row based on the size of the screen.

This approach is clearly visible when you launch your application on a desktop. As soon as you start resizing the window, you will notice that the items will move back and forth based on the available space, helping you to always deliver a great user experience, no matter which device is running the app.

Showing grouped collections

One of the most interesting features offered by **ListView** or **GridView** controls is grouped collections support. You'll be able to display a collection of data grouped in different categories. The user will be able, other than just scrolling through the list, also to quickly jump from one category to another.

Many native Windows 10 applications take advantage of this approach. For example, if you open the People application, you will notice that contacts are grouped in different categories based on the initial letter of the name. By tapping on a letter, a new view that displays all the available letters is opened, so that the user can quickly jump from one group of contacts to another.

Let's see a real example by changing the **Person** class we've previously defined a bit:

Code Listing 132

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

We've added a new property called **City**; we'll use it to group our collection of people by the city where they live.

To achieve our goal. we need to introduce a new class offered by the Windows Runtime, called **CollectionViewSource**, which acts as a proxy between the data collection and the control that will display it (in our case, a **GridView** or a **ListView** control). Instead of connecting the **ItemsSource** property of the control directly to our collection, we'll connect it to this proxy object, which offers many advanced features, like automatic grouping support.

The **CollectionViewSource** can be defined in XAML as a regular resource. The following sample shows a **CollectionViewSource** object defined as page resource:

Code Listing 133

```
<Page.Resources>
    <CollectionViewSource x:Name="People" IsSourceGrouped="True" />
</Page.Resources>
```

As you can see, thanks to the **IsSourceGrouped** property, we can easily specify that the data stored in this collection will be grouped. The next step is to connect our data to this proxy class. The procedure is like the one we've seen for a flat list, except that this time we need to specify

the group criteria. We can easily do this thanks to LINQ and to the **GroupBy()** extension. Here is an example:

Code Listing 134

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    List<Person> people = new List<Person>
    {
        new Person
        {
            Name = "Matteo",
            Surname = "Pagani",
            City = "Como"
        },
        new Person
        {
            Name = "Ugo",
            Surname = "Lattanzi",
            City = "Milan"
        },
        new Person
        {
            Name = "Roberto",
            Surname = "Freato",
            City = "Milan"
        },
        new Person
        {
            Name = "Massimo",
            Surname = "Bonanni",
            City = "Rome"
        }
    };
    var groups = people.GroupBy(x => x.City);
    People.Source = groups;
}
```

We've grouped the collection by the property called **City** by applying the **GroupBy()** extension method to the original collection. Then, we've assigned the resulting grouped collection to the **Source** property of the **CollectionViewSource** object we previously defined as resource in the page.

We're done working on the code. However, our goal is not achieved yet, since we need to define the visual layout of the collection. By default, in fact, neither the **GridView** nor the **ListView** control know how to visually display the groups. The **ItemTemplate** property defines what a single item will look like, but not how the entire group will be rendered. Consequently, we need to apply some changes to the XAML definition of the control.

The following sample shows a **GridView** control configured to display our grouped collection:

Code Listing 135

```
<GridView ItemsSource="{Binding Source={StaticResource People}}">
    <GridView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding Path=Name}" />
                <TextBlock Text="{Binding Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </GridView.ItemTemplate>
    <GridView.GroupStyle>
        <GroupStyle HidesIfEmpty="True">
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <Border Background="LightGray">
                        <TextBlock Text="{Binding Key}" Foreground="Black"
Margin="10" FontSize="22"/>
                    </Border>
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
        </GroupStyle>
    </GridView.GroupStyle>
</GridView>
```

The first thing you'll notice is that we're setting the **`ItemsSource`** property in a different way. Since the **`CollectionViewSource`** object has been defined as resource, we assign it using the **`StaticResource`** markup extension.

Then we need to set up the different visual styles of the list:

- The first one shouldn't be a surprise: as we have done to manage a flat list, we define the **`ItemTemplate`**, the template used to render every single item of the collection. We're using the same one we've previously seen that displays the name and the surname of the person.

- The second one is the **`GroupStyle`** element, which defines the behavior of the group. By setting the **`HidesIfEmpty`** property to **`True`**, we make sure that a group isn't displayed if there are no elements in it. In our case, we don't want the Milan group to be displayed if there are no people from Milan in the collection.

- The **`GroupStyle`** control offers an important property called **`HeaderTemplate`**. It's the template used to render the group header displayed at the beginning of each group. This goal is achieved by adding a **`TextBlock`** control connected to the **`Key`** field. What is it? When we've grouped the collection using the **`GroupBy()`** method, we've basically created another collection with one element for each group (the user's city). Each element contains the list of items that belong to the group (the people who live in that city) and a key (the **`Key`** property) with the name of the group. With this **`DataTemplate`**, we'll simply display a text with the name of the city, followed by the list of people who live there.

The following image shows how the previous code is rendered in a real application.

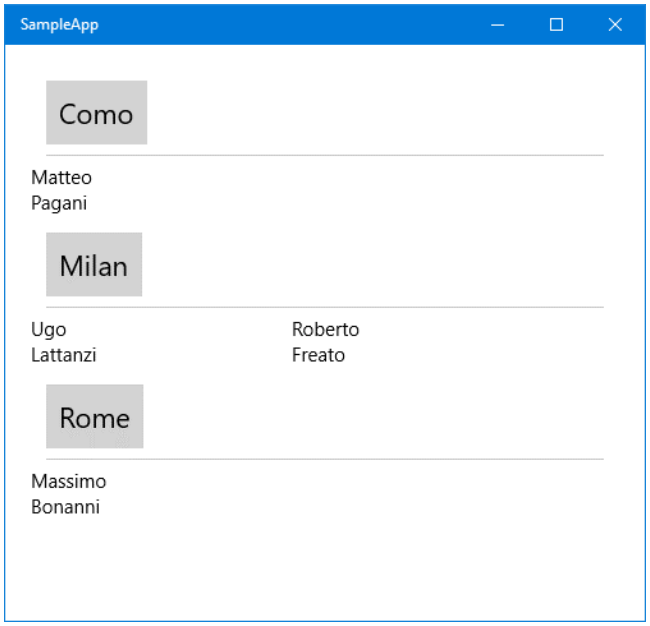


Figure 36: A grouped GridView.

As mentioned previously, the **`ListView`** control works in exactly the same way. By simply changing the previous code by replacing all the references to the **`GridView`** control, we will be able to turn the grid into a traditional vertical list, like in the following image.

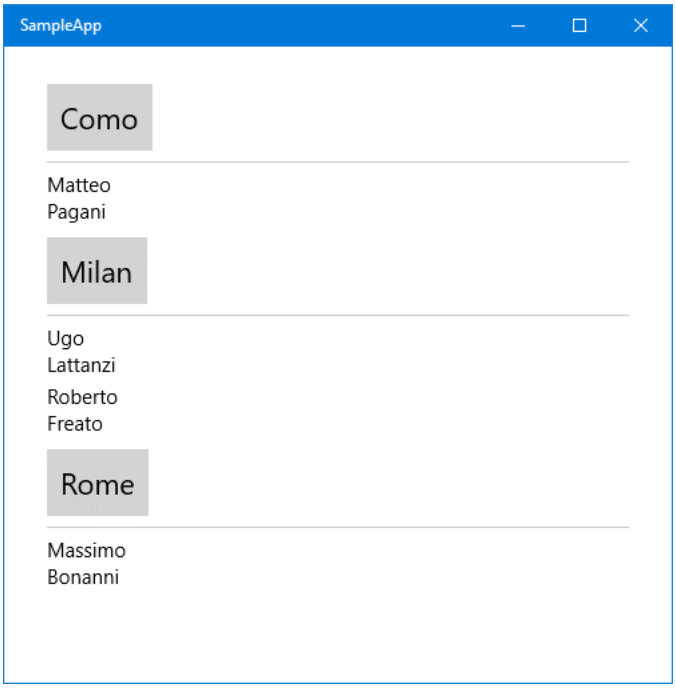


Figure 37: A grouped ListView

As you can see, the orientation difference is applied not to the groups, but to the elements inside them. In both cases, the groups are displayed vertically, one below the other. However, the elements inside (each **Person** object) are placed horizontally or vertically, based on the control's type.

If we also want to change the orientation of the groups, we need to change the **ItemsPanel** of the control, the panel used to render each group. For this scenario, we can leverage the **ItemWrapGrid** panel, which allows us to automatically split the groups into multiple rows and columns, based on the kind of orientation we want to leverage.

Here is what our updated XAML code looks like:

Code Listing 136

```
<GridView ItemsSource="{Binding Source={StaticResource People}}">
    <GridView.ItemTemplate>
        <DataTemplate>
            <StackPanel Width="200">
                <TextBlock Text="{Binding Path=Name}" />
                <TextBlock Text="{Binding Path=Surname}" />
            </StackPanel>
        </DataTemplate>
    </GridView.ItemTemplate>
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <ItemsWrapGrid MaximumRowsOrColumns="3" />
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>

    <GridView.GroupStyle>
        <GroupStyle HidesIfEmpty="True">
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <Border Background="LightGray">
                        <TextBlock Text="{Binding Key}" Foreground="Black"
Margin="10" FontSize="22"/>
                    </Border>
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
        </GroupStyle>
    </GridView.GroupStyle>
</GridView>
```

Highlighted in yellow, you can see the new property we have added. We have set the **ItemsPanel** property of the **GridView** control by leveraging as template the **ItemsWrapGrid** control. With the **MaximumRowsOrColumns** number we can specify the maximum number of rows or columns we want to display before the control starts splitting the groups into a new row or column.

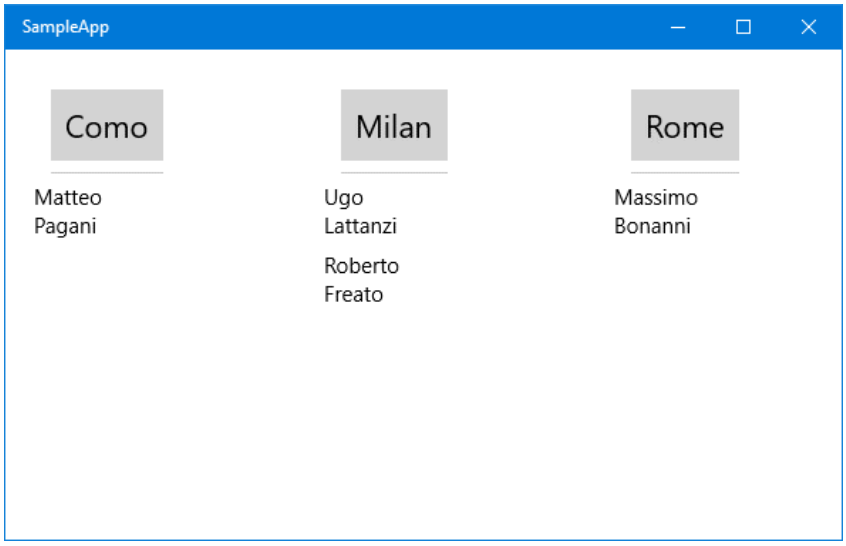


Figure 38: A GridView using an ItemsWrapGrid as the panel's template.

Handling the item selection

The most common requirement when you work with a collection of data is to manage selection. The user taps on an item and we want to detect which has been selected so that we can perform additional operations (like redirecting the user to a detail page). There are four ways to manage selection and you can choose your favorite one by setting the **`SelectionMode`** property:

- **`Single`**: the default mode, the user can select only one item, which is highlighted until another item is selected.
- **`Multiple`**: this mode allows the user to select multiple items. Every time they tap on an item, it's automatically added to the list of selected items.
- **`Extended`**: a combination of the previous two modes. The single tap triggers a standard selection, while a right-click with the mouse will add it to the list of selected items.
- **`None`**: disables the selection. No items can be selected and the **`SelectionChanged`** event is not triggered.

Except for the last one, these modes always trigger the **`SelectionChanged`** event when the user taps on one item. The only difference is that, in the case of single selection, we can use just the **`SelectedItem`** property of the control, which contains the selected item. Otherwise, in cases of multiple selection, we can use the **`SelectedItems`** property, a collection that contains all the flagged items.

The following sample shows how to use the **`SelectedItem`** property when the **`SelectionChanged`** event is triggered to display the name of the selected person with a pop-up:

Code Listing 137

```
private async void List_OnSelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    Person selectedPerson = List.SelectedItem as Person;
    if (selectedPerson != null)
    {
        MessageDialog dialog = new MessageDialog(selectedPerson.Name);
        await dialog.ShowAsync();
    }
}
```

It's important to highlight that, since the **GridView** or **ListView** controls can display any collection of data, the **SelectedItem** property is a generic **object**. Before accessing its properties, we need to cast it to the type we're expecting (in our case, it's a collection of **Person** objects).

The following sample shows a similar scenario, but with multiple selection enabled. In this case, we display to the user the number of items they selected in the list.

Code Listing 138

```
private async void List_OnSelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    int selectedItems = List.SelectedItems.Count;
    MessageDialog dialog = new MessageDialog(selectedItems.ToString());
    await dialog.ShowAsync();
}
```

The **GridView** and **ListView** controls also offer an alternative way to manage the selection, which is useful when we don't have to perform any special operation on the selected item, but just want, for example, to redirect the user to another page when the item is clicked. In this mode, the items are treated like buttons. When you tap on one of them, an event called **ItemClick** will be triggered and it will contain, as parameter, the item that has been selected.

To enable this mode, you'll have to set the **IsItemClickEnabled** property to **True** and then subscribe to the **ItemClick** event. Usually, when you enable this mode, it's also useful to set the **SelectionMode** to **None**, to avoid overlapping the two selection modes and to avoid having both the **ItemClick** and the **SelectionChanged** events triggered.

Managing the **ItemClick** event in the code-behind is easy, as you can see in the following sample:

Code Listing 139

```
private async void List_OnItemClick(object sender, ItemClickEventArgs e)
{
    Person person = e.ClickedItem as Person;
    if (person != null)
```

```
    {
        MessageDialog dialog = new MessageDialog(person.Name);
        await dialog.ShowAsync();
    }
}
```

The event handler's parameter (an **ItemClickEventArgs** type) contains a property called **ClickedItem**, with a reference to the selected item. As usual, since the control can display any type of data, the **ClickedItem** property contains a generic object, so we need to perform a cast before using it.

Semantic Zoom

The Semantic Zoom is a feature that is part of the native experience delivered by Windows and that offers to the users two different ways to browse a collection of data. There's a traditional one, with all the details (like the one we've just seen talking about the **GridView** and **ListView** controls), and a "high level" one, which gives the users a glance at all the available groups, allowing them to quickly jump from one to another.

The Universal Windows Platform's **SemanticZoom** control is easy to understand:

Code Listing 140

```
<SemanticZoom>
    <SemanticZoom.ZoomedInView>
        <!-- standard visualization -->
    </SemanticZoom.ZoomedInView>
    <SemanticZoom.ZoomedOutView>
        <!-- groups visualization -->
    </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

The **SemanticZoom** control can manage two different statuses, represented by two specific properties. **ZoomedInView** defines the layout that displays the traditional list with all the details. **ZoomedOutView** defines the layout that displays all the groups. Inside these properties, you can't define any arbitrary XAML. Only controls that support the **ISemanticZoomInformation** interface can properly work with this feature. The Windows Runtime offers three native controls that support this interface: **GridView**, **ListView**, and **Hub**.

When it comes to managing the **ZoomedInView**, it's no different from what we've learned about the **GridView** or the **ListView** controls. In this view, we need to manage the traditional list, so we're going to create a collection of data, define it as a **CollectionViewSource** object and connect it to the **ItemsSource** property of the control. The **ZoomedInView** property will contain a block of XAML code like the following one:

Code Listing 141

```
<Page.Resources>
```

```
    <CollectionViewSource x:Name="People" IsSourceGrouped="True" />
</Page.Resources>

<SemanticZoom>
    <SemanticZoom.ZoomedInView>
        <GridView ItemsSource="{Binding Source={StaticResource People}}">
            ...
        </GridView>
    </SemanticZoom.ZoomedInView>
</SemanticZoom>
```

As you can see, the **GridView** control is simply connected to the **CollectionViewSource** object, which has been defined as a page resource.

The **ZoomedOutView**, however, needs to be managed in a different way. When it's enabled, we don't need to display all the items in the collection, but just the groups the data is divided into. To achieve our goal, the **CollectionViewSource** offers a property called **CollectionGroups**, which contains all the groups the data is split into. Do you remember the sample we saw previously with a list of people grouped by the city they live in? In our case, the **CollectionGroups** property will contain just the list of cities.

Here is how a sample definition of the **ZoomedOutView** property looks:

Code Listing 142

```
<SemanticZoom>
    <SemanticZoom.ZoomedOutView>
        <ListView ItemsSource="{Binding Source={StaticResource People},
Path=CollectionGroups}">
            ...
        </ListView>
    </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

We're using, again, a **ListView** control to display the list of groups but, instead of binding the **ItemsSource** property directly with the **CollectionViewSource** object, we bind it to the specific **CollectionGroups** property.

The last step is to define the visual layout of the **ZoomedOutView** mode. We're using a standard **ListView** control, so we simply need to define the **ItemTemplate** property with a proper **DataTemplate**, like in the following sample:

Code Listing 143

```
<SemanticZoom>
    <SemanticZoom.ZoomedOutView>
        <ListView ItemsSource="{Binding Source={StaticResource
People},Path=CollectionGroups}">
            <ListView.ItemTemplate>
```

```
                <DataTemplate>
                    <Border Background="LightGray" Width="300" Padding="5">
                        <TextBlock Text="{Binding Group.Key}"
Foreground="Black" TextAlignment="Center" FontSize="22" />
                    </Border>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </SemanticZoom.ZoomedOutView>
    <SemanticZoom.ZoomedInView>
        <GridView ItemsSource="{Binding Source={StaticResource People}}">
            ...
        </GridView>
    </SemanticZoom.ZoomedInView>
</SemanticZoom>
```

It's a standard **DataTemplate**: the only thing to highlight is that the **TextBlock** control is connected to a property called **Group.Key**, which contains the name of the group that we want to display (in our case, the city).

The following image shows what happens in the app when you zoom out from the standard list:

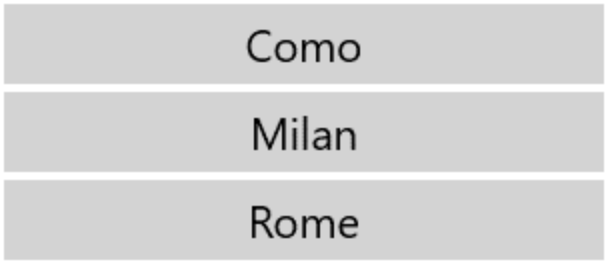


Figure 39: The group list displayed using the SemanticZoom control.

FlipView

The **FlipView** control offers another way to display a collection of items that is useful when you want to focus the user's attention on the selected item, like in a photo gallery. In fact, when you use the **FlipView** control, only the selected item is visible and it occupies all the available space. The user needs to swipe to the left or right (or use the buttons that are displayed on both sides of the screen when the app is running on a device with mouse and keyboard) to display the other items in the collection.

Except for this difference, the **FlipView** control behaves like a **GridView** or **ListView** control:

- You need to set the **ItemTemplate** property to define the layout of the selected item.
- You need to assign the collection of items to the **ItemsSource** property.

You can subscribe to the **SelectionChanged** event to be notified every time the user swipes to the left or right to display another item. If you need to discover which item is currently displayed, you can use the **SelectedItem** property.

The following XAML code shows a sample definition of a **FlipView** control used to display an images gallery:

Code Listing 144

```
<FlipView x:Name="Images">
    <FlipView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Image Source="{Binding Image}" Stretch="UniformToFill"/>
                <Border Background="#A5000000" Height="80"
VerticalAlignment="Bottom">
                    <TextBlock Text="{Binding Title}" FontFamily="Segoe UI"
FontSize="26" Foreground="#CCFFFFFF" Padding="15,20"/>
                </Border>
            </Grid>
        </DataTemplate>
    </FlipView.ItemTemplate>
</FlipView>
```

Navigation controls

This category includes all the controls that can be used to define the navigation experience of your application, meaning the way the user will be able to move across the different sections and pages of your application. One navigation approach isn't better than another, as it all depends on your scenario and the kind of content you want to display.

Hub

The **Hub** control is often used to define the main page of an application, and it's composed of different sections placed one next to the other. The user can swipe to the left or right of the screen to see the previous or next section. To help users understand the sections concept, a section doesn't take up the whole space on the page: the right margin is used to display a glimpse of the next section, so that the user can understand that there's more content to see and discover.

Typically, the **Hub** control isn't used to contain huge amounts of data, but to provide a subset of it and to give quick access to the different sections of the applications. For example, in a news reader application you won't use the **Hub** control to display all the available news; this task can be assigned to a specific page of the application. However, a section of the **Hub** control could display just the most recent news and then provide a link to see all of them.

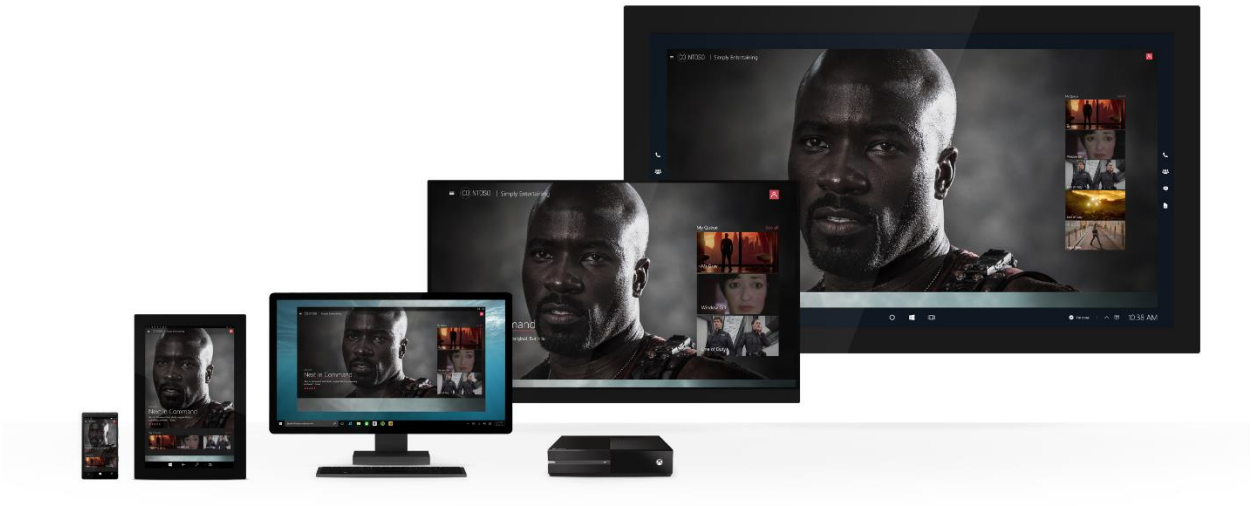


Figure 40: The Hub control displayed on different kinds of Windows 10 devices.

Here is a sample **Hub** control's definition:

Code Listing 145

```
<Hub Header="Page title">
    <HubSection Header="First section">
        <DataTemplate>
            <Image Source="/Assets/image.png" />
        </DataTemplate>
    </HubSection>
    <HubSection Header="Second section">
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="Some content" />
            </StackPanel>
        </DataTemplate>
    </HubSection>
</Hub>
```

The **Hub** control can include a title displayed at the top. It's set using the **Header** property, which accepts a simple string. You can also fully customize the header by defining a new template thanks to the **HeaderTemplate** property.

As we've previously mentioned, the **Hub** control is split into different sections. Each of them is identified by the **HubSection** control. Every section has unique features:

- The **HubSection** control offers a property called **Header**, which contains the section's title and is displayed at the top of every section.
- The content of the section is defined using a **DataTemplate**. It's important to highlight that, despite this behavior, the **Hub** control isn't able to display a collection of data. You'll notice, in fact, that the **ItemsSource** property is missing.

Using a **DataTemplate** to define the section's look and feel provides some challenges, compared to defining the layout of a simple page. In fact, since it's a **DataTemplate**, we can't simply assign a name to a control using the **x:Name** property and then access to it in code-behind. The controls, in fact, aren't part of the page, but are included in the **DataTemplate** connected to the **HubSection**.

Let's see a real example to better understand the issue. Take as an example the following XAML code:

Code Listing 146

```
<Hub>
    <HubSection Header="First section">
        <DataTemplate>
            <TextBlock x:Name="Name" />
        </DataTemplate>
    </HubSection>
</Hub>
```

Normally, if you wanted to update the **TextBlock**'s content from code, you would write an event handler like the following one:

Code Listing 147

```
private void OnButtonClicked(object sender, RoutedEventArgs e)
{
    Name.Text = "Matteo";
}
```

However, this code won't compile since the **TextBlock** control identified by the keyword **Name** is defined inside a **DataTemplate**, so it can't be directly accessed. The solution is to use binding, as we've learned in Chapter 2.

The previous **Hub** control definition needs to be changed in the following way:

Code Listing 148

```
<Hub x:Name="MainHub">
    <HubSection Header="First section">
        <DataTemplate>
            <TextBlock Text="{Binding Name}" />
        </DataTemplate>
    </HubSection>
</Hub>
```

We need also to change the code in code-behind to properly set the control's **DataContext** so that it can find a property called **Name** to resolve the binding expression:

Code Listing 149

```
public MainPageView()
{
    this.InitializeComponent();
    Person person = new Person();
    person.Name = "Matteo";
    MainHub.DataContext = person;
}
```

Another feature supported by the **Hub** control is interactive headers. When we enable it, a link labelled **See more** is made visible near the header that the users can tap on, so that we can perform additional operations or navigations. For example, a section could display just a couple pieces of news but, by tapping on the header, the users can be redirected to another page of the application where they can read all the available news.

To enable this feature, you'll need to set the **IsHeaderInteractive** property to **true** on every **HubSection** control you want to manage this way. Then, you need to implement the **SectionHeaderClick** event that is offered directly by the **Hub** control.

The following sample shows a **Hub** control where the first section has been configured to support interactive headers:

Code Listing 150

```
<Hub SectionHeaderClick="Hub_OnSectionHeaderClick">
    <HubSection Header="First section" IsHeaderInteractive="True">
        <DataTemplate>
            <Image Source="/Assets/image.jpg" />
        </DataTemplate>
    </HubSection>
    <HubSection Header="Second section">
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="Some content" />
            </StackPanel>
        </DataTemplate>
    </HubSection>
</Hub>
```

The following code shows how to manage the **SectionHeaderClick** event, so that we can be notified every time the user taps on a header:

Code Listing 151

```
private async void Hub_OnSectionHeaderClick(object sender,
HubSectionHeaderClickEventArgs e)
{
    MessageDialog dialog = new MessageDialog(e.Section.Header.ToString());
    await dialog.ShowAsync();
```

```
}
```

The event handler offers a parameter (a **HubSectionHeaderClickEventArgs** type) containing a property called **Header**, the **HubSection** control that triggered the event. You can use it to determine which section has been tapped and perform the proper navigation. In the previous sample, we just display the header of the selected section with a pop-up message.

The **Hub** control was introduced in Windows 8, but Windows 10 has added a new feature. You can change the orientation by using the **Orientation** property. By default, the **Hub** control spans horizontally, but you can change this property to **Vertical**, so all the sections will be displayed one below the other instead of one after the other.

The Store app, for example, uses this approach to display the various categories of apps in the main page, as you can see in the following image:

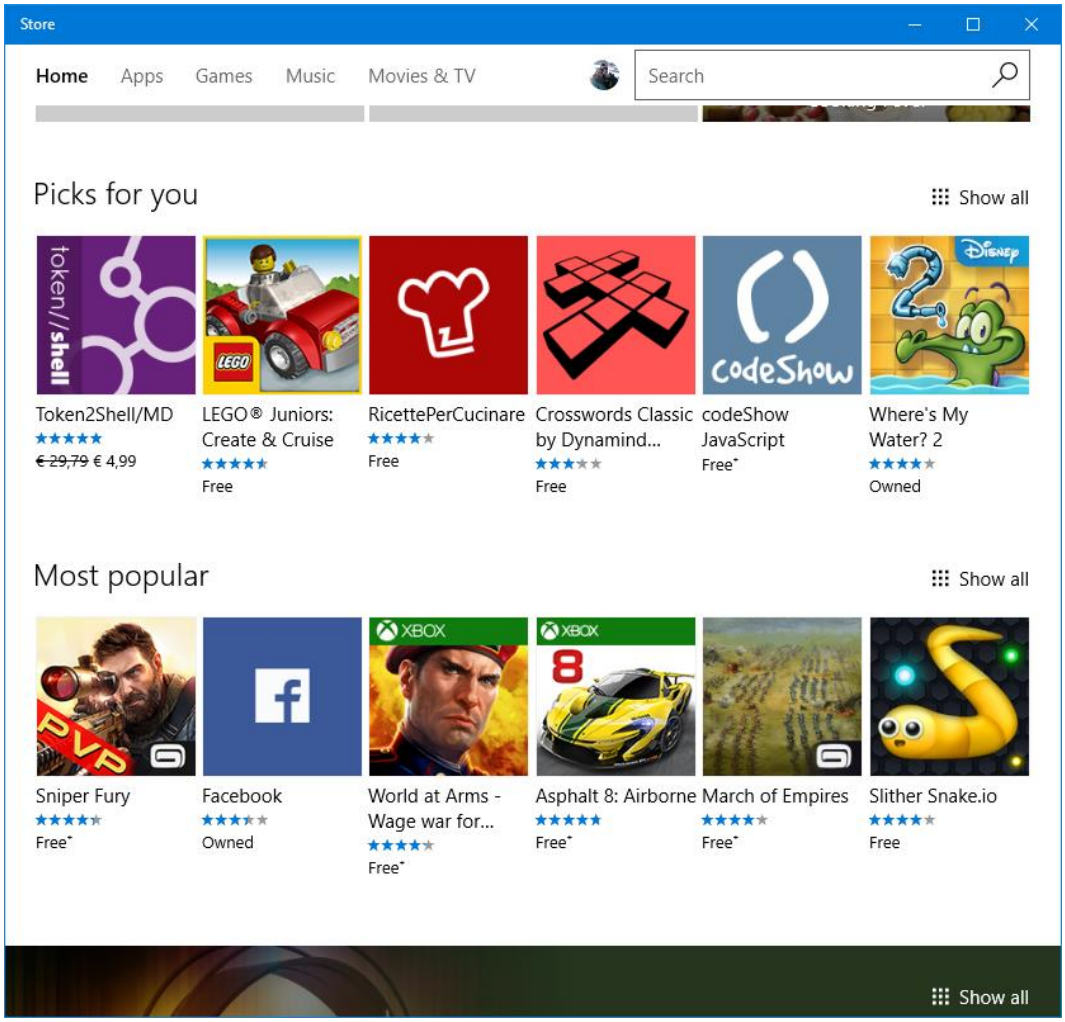


Figure 41: The Hub control configured to use a vertical orientation.

In the previous image, **Picks for you** and **Most popular** are two **HubSection**s of a **Hub** control and they are displayed one below the other. You 'll notice, also, that the **IsHeaderInteractive**

property has been set to **`True`**. You can see the **Show all** link at the end of each section that redirects the user to another page of the app listing all the apps in the selected category.

Pivot

The **`Pivot`** control (which, before Windows 10, was available only on the phone) offers a user experience like the one delivered by the tab control in other mobile platforms. The **`Pivot`** is split into different sections that the user can see by swiping to the left or right of the screen (or by clicking on the arrows at the two sides if they're visible; this is another control that delivers an optimized experience if the app is running on a device with a mouse and keyboard). In this case, however, every section will fit the entire size of the page. To help the user understand that there are other sections, a top bar will display the names of the other sections, with the current one highlighted in a different color.

The **`Pivot`** control is typically used in two different scenarios:

- You need to show the user the same type of information, but refer to different contexts. The MSN News application by Microsoft is a good example. All the pivot's sections display the same type of information (news), but filtered by different categories (politics, financial, science, etc.).

- You need to show the user different information types, but they're related to the same context. The built-in People application is a good example. When you tap on a contact, you can see all their details, like phone number, photos posted on social networks, the latest interactions, etc. All this information is stored in different sections of the page.

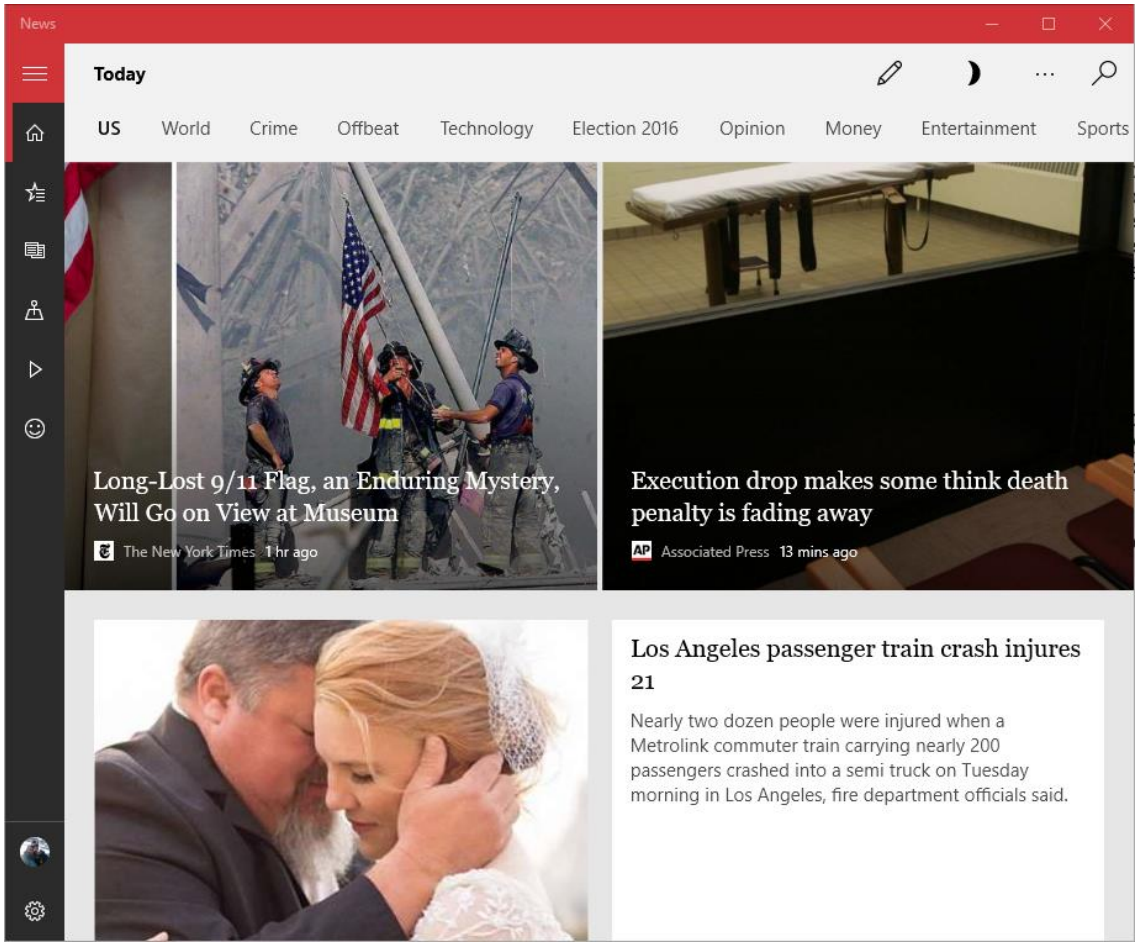


Figure 42: The MSN News app uses a Pivot control (at the top) to allow users to switch from one news category to another.

The following sample shows how to use the **`Pivot`** control in a XAML page:

Code Listing 152

```
<Pivot Title="Page title">
    <PivotItem Header="First header">
        <StackPanel>
            <TextBlock Text="Some content" />
        </StackPanel>
    </PivotItem>
    <PivotItem Header="Second section">
        <StackPanel>
            <TextBlock Text="Some other content" />
        </StackPanel>
    </PivotItem>
</Pivot>
```

The **`Pivot`** control has a property called **`Title`**, which defines the title of the page. Every section, on the other hand, is identified by a **`PivotItem`** control, which is nested inside the main

Pivot one. Each of them can have its own title, which is the header displayed in the top bar. The property to set is called **Header**.

Except for these features, the **PivotItem** control acts as a simple container. You can place inside it any other XAML control you want and it will be rendered in the page when the current section is active.

The **Pivot** control can also be useful to create guided procedures (like a configuration wizard). It offers, in fact, a property called **IsLocked** that prevents the user from moving to another section when it's set to **true**. This way you can unlock the next section only when the user has fulfilled all the required fields in the current one.

The **Pivot** control can also be heavily customized to make it more like a tab control. For example, the built-in Alarms & Clock app in Windows 10 uses this approach to define the different sections, as you can see in the following image.

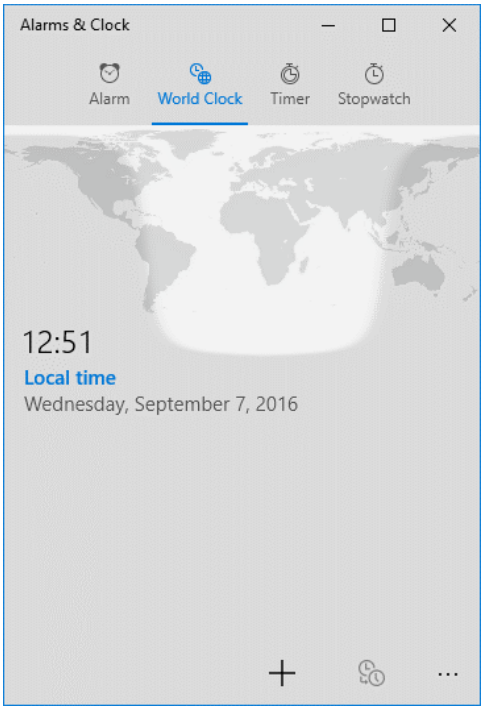


Figure 43: The Alarms & Clock app uses a customized Pivot to create a tabs effect.

However, this kind of customization isn't built in to the control; you'll need to make many changes to its template. You can find a working sample in the official Windows 10 repository on GitHub.

SplitView

The **SplitView** control is another new feature added in Windows 10, and it allows you to create "hamburger menu" experiences in your application easily. What is a hamburger menu? It's a panel that, typically, is placed on the left side of the screen and can be activated anytime by

tapping a button. This button is usually identified by three lines, one below the other, which reminds you of a hamburger.

The panel is typically a way to give the user quick access to different sections of the applications. Unlike other navigations approaches (like the ones you can implement with the **`Hub`** or the **`Pivot`** controls), a hamburger menu is available across every page of the application. Consequently, no matter which page the user is visiting, they'll always be able to quickly jump to another section of the app.

Many Windows 10 built-in apps leverage this control. The following image shows Groove Music, the Windows 10 music player, which uses a hamburger menu to give quick access to the various sections of the application, like Albums, Artists, Songs, Settings, etc.

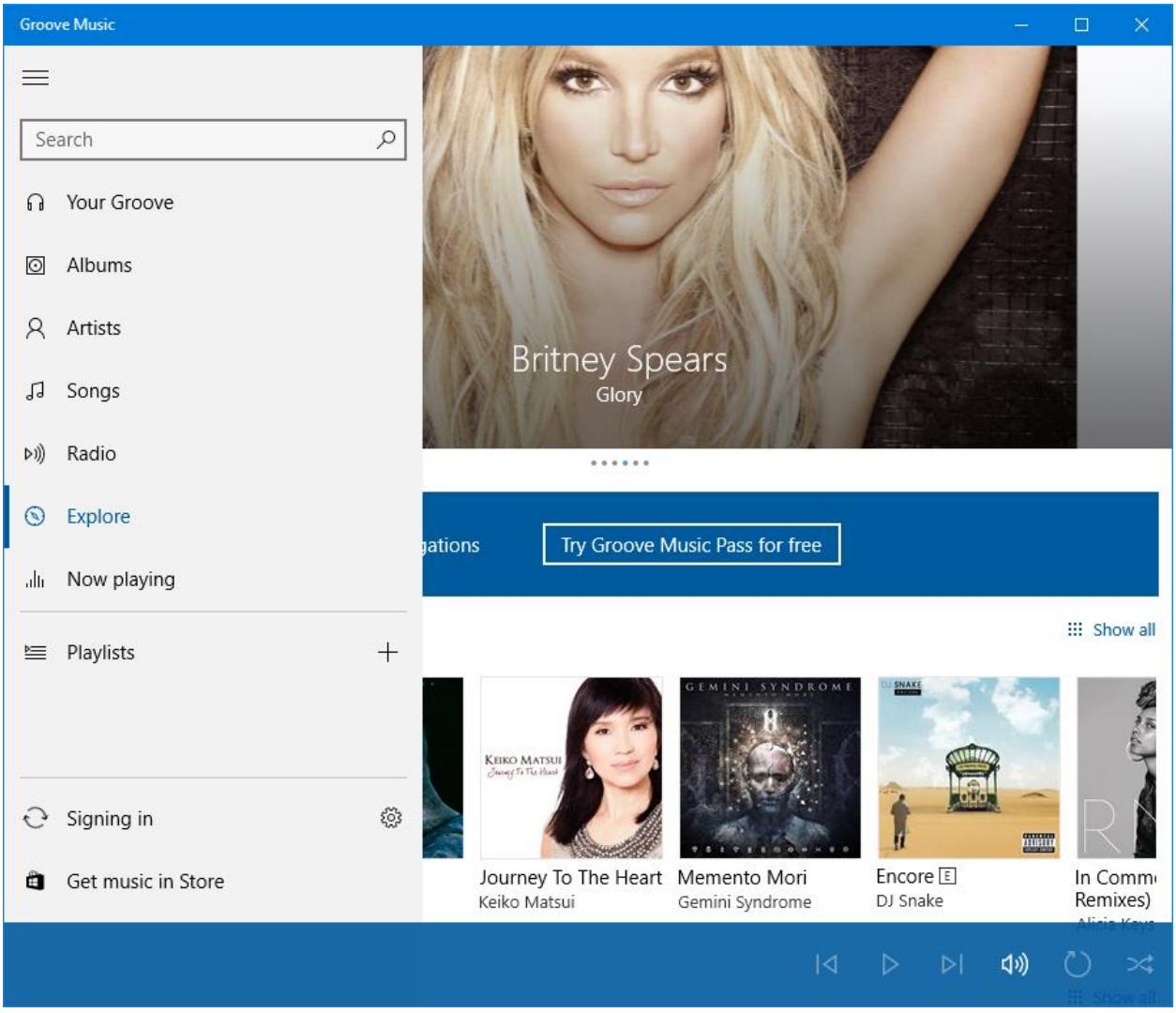


Figure 44: The hamburger menu implemented in the Groove Music app.

However, by default, the **`SplitView`** control doesn't offer any built-in feature to implement these kinds of experiences. In fact, it just offers a way to split a page into a panel and the main

content. Inside each part, you can define your own layout by leveraging any XAML control. Here is a very simple definition of the **SplitView** control:

Code Listing 153

```
<SplitView DisplayMode="Inline" PanePlacement="Left" IsPaneOpen="True">
    <SplitView.Pane>
        <TextBlock Text="This is the panel" FontSize="24"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />
    </SplitView.Pane>

    <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center"
               FontSize="24"
               Text="This is the content" />
</SplitView>
```

The content of the panel is included inside a property called **Pane**, while the content of the page (in the previous sample, made by a **TextBlock**) is included directly as children of the **SplitView** control. In the sample, you can see some of the most important properties offered by the control:

- **DisplayMode** defines the way the panel is displayed.
- **Inline** means that the panel, when it's open, takes space from the content area.
- **CompactInline** means that, when the panel is closed, a small portion is still visible. When it's opened, it behaves like the **Inline** property, so it takes space from the content area.
- **Overlay** means that the panel, when it's open, overlaps the content area, without stealing space.
- **CompactOverlay** means that, when the panel is closed, a small portion is still visibile. When it's opened, it behaves like the **Overlay** property, so it overlaps the content area.
- **PanePlacement** can be used to position the panel on the left or right sides of the screen.
- **IsPaneOpen** defines if the panel is visible or not. The outcome of this property is strictly connected to the **DisplayMode**. With regular display modes, if this property is set to **False**, it won't be visible at all. With compact display modes, a small portion will be visible and expanded only when the property is set to **True**.

The following image shows how the previous XAML code is rendered:

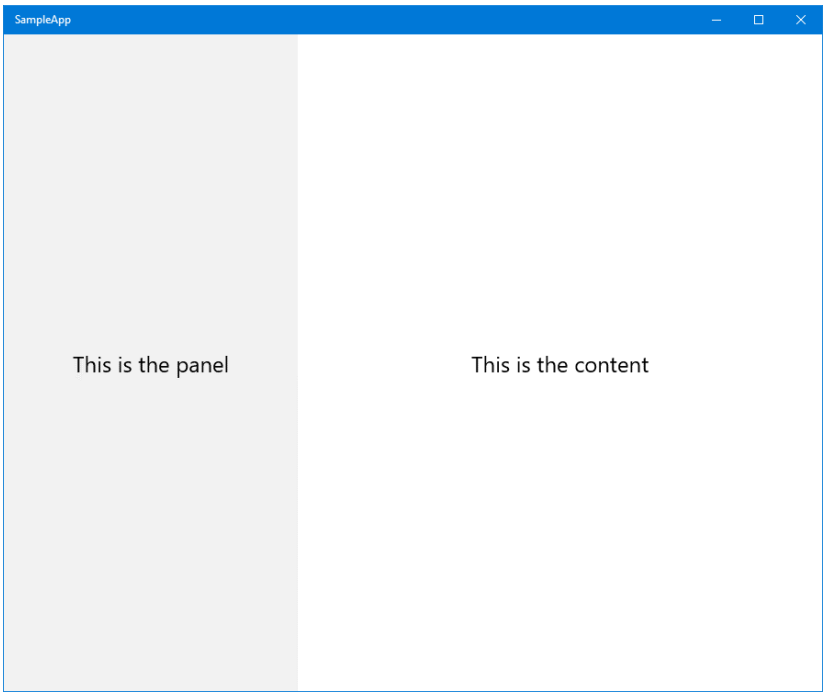


Figure 45: The SplitView control.

The approach provided by the **`SplitView`** control has some pros and some cons. The biggest advantage is that it's very flexible. As you can see, you can define the panel with any XAML control, you can customize the position, you can customize the behavior, etc. The downside is that, if you want to provide a standard hamburger menu experience like the one we've seen in the Groove Music app, you must do everything on your own. The framework doesn't provide built-in controls to create the hamburger button (the one placed at the top to show/hide the panel), or to define the various sections, etc.

As a result, there are many third-party libraries that introduced their own implementation of the hamburger menu based on the **`SplitView`** control, which makes it easier to recreate the user experience of the built-in Windows 10 applications.

At the time of writing, the most useful implementations you can leverage in your applications are:

- The **`HamburgerMenu`** control in the **UWP Community Toolkit**, an open-source project created by Microsoft with the goal of expanding the features offered by the Universal Windows Platform by offering additional controls and services. Being an open-source project, it allows every member of the community to support and improve it. Additionally, being a third-party library and not something embedded into the platform, it can be updated much more frequently than the Universal Windows Platform. The starting page of the toolkit is here. From there, you can access the documentation, the GitHub repository, and a sample app (published on the Store) where you can try out the various controls. Specifically, you can find documentation and sample code about the **`HamburgerMenu`** control on this page.

- The **HamburgerMenu** control in **Template 10**, another open-source project created by Microsoft with the goal of providing better-structured and easier-to-use templates as starting points to create Universal Windows Platform apps. Being an open-source project, it's available on GitHub. The repository contains many samples and, additionally, you can install an extension for Visual Studio 2015 that adds a set of new templates in the Universal category of the Windows section. One of them contains a sample implementation of the **HamburgerMenu** control and so is a great starting point if you're planning to leverage this navigation approach in your application. You can read the full documentation here.

Additionally, both toolkits allow you to solve one of the challenges with the **SplitView** control. By default, a Universal Windows Platform app is embedded into a **Frame**, which is a container of all the pages of the application. Thanks to the **Frame** class, you can leverage a set of APIs to move from one page to the other. The problem in using the **SplitView** control is that, since by default you leverage a single empty **Frame** in the app, you need to find a way to have the same menu applied across every page and, at the same time, keep the navigation flow consistent. Both toolkits offer a way to turn the **Frame** from an empty container to a container with the **HamburgerMenu** control. This way, all the other pages of your application will be hosted inside this container and they will share the same menu.

You'll see more details in the second book of the series, where we're going to discuss in the detail how to handle navigation in Universal Windows Platform apps.

I won't describe in detail in this chapter how to use the two mentioned **HamburgerMenu** controls. It would be out of the scope of this book, since they're part of an open-source project that is continuously evolving and, therefore, this information may quickly become outdated.

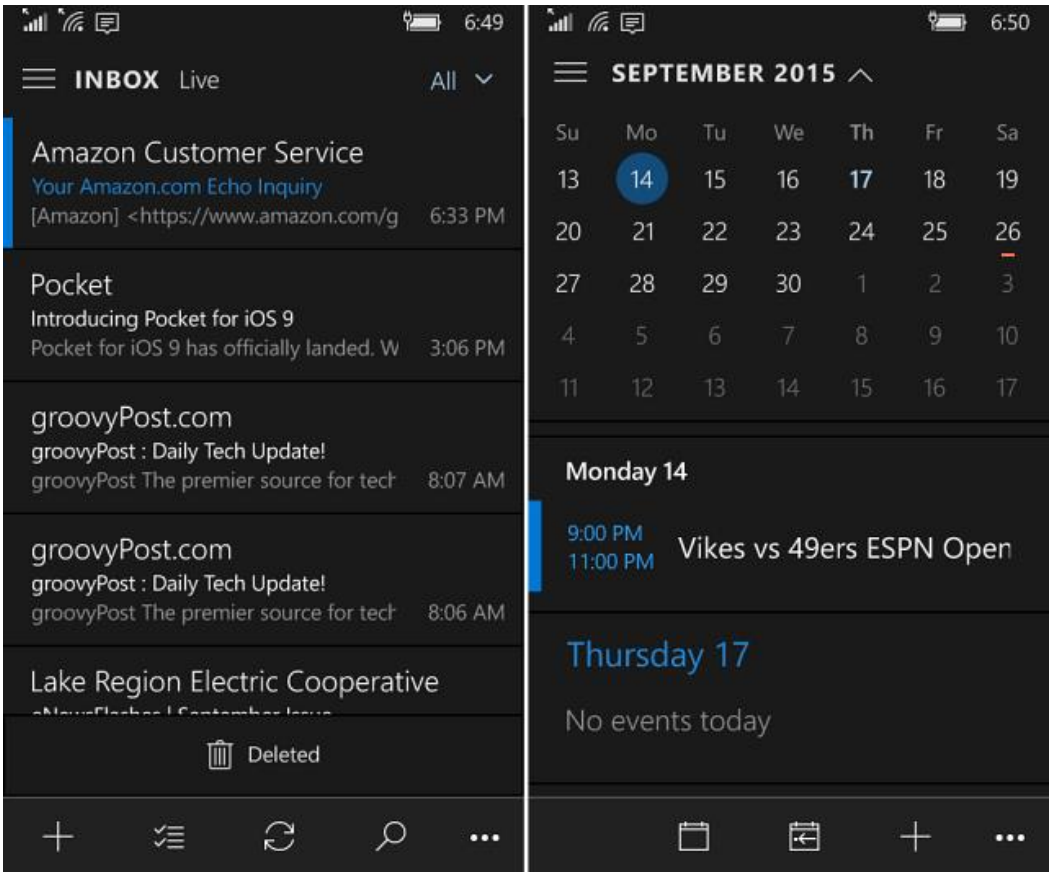
Managing the application bar: CommandBar

The application bar is one of the key controls in Windows and it's widely used by many applications. It provides a bar, which can contain different interactive elements (like buttons) that typically provide a set of options to interact with the content that is currently being displayed.

For example, the built-in Mail & Calendar app leverages an application bar when you're reading mail to provide the most-used functionalities, like Reply, Reply All, Forward, etc., or when you're using the Calendar, to give a quick way to create a new appointment, to move to the current date, etc.

To create an application bar, the Universal Windows Platform offers a control called **CommandBar**, which is defined at page level and can be placed in two positions:

- At the top of the screen, by leveraging the **TopAppBar** property of the **Page** class.
- At the bottom of the screen, by leveraging the **BottomAppBar** property of the **Page** class.


6:49
INBOX Live
All
Amazon Customer Service
Your Amazon.com Echo Inquiry
[Amazon] <https://www.amazon.com/g
6:33 PM
Pocket
Introducing Pocket for iOS 9
Pocket for iOS 9 has officially landed. W
3:06 PM
groovyPost.com
groovyPost : Daily Tech Update!
groovyPost The premier source for tech
8:07 AM
groovyPost.com
groovyPost : Daily Tech Update!
groovyPost The premier source for tech
8:06 AM
Lake Region Electric Cooperative
Deleted
6:50
SEPTEMBER 2015
Su Mo Tu We Th Fr Sa
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
Monday 14
9:00 PM
11:00 PM
Vikes vs 49ers ESPN Open
Thursday 17
No events today

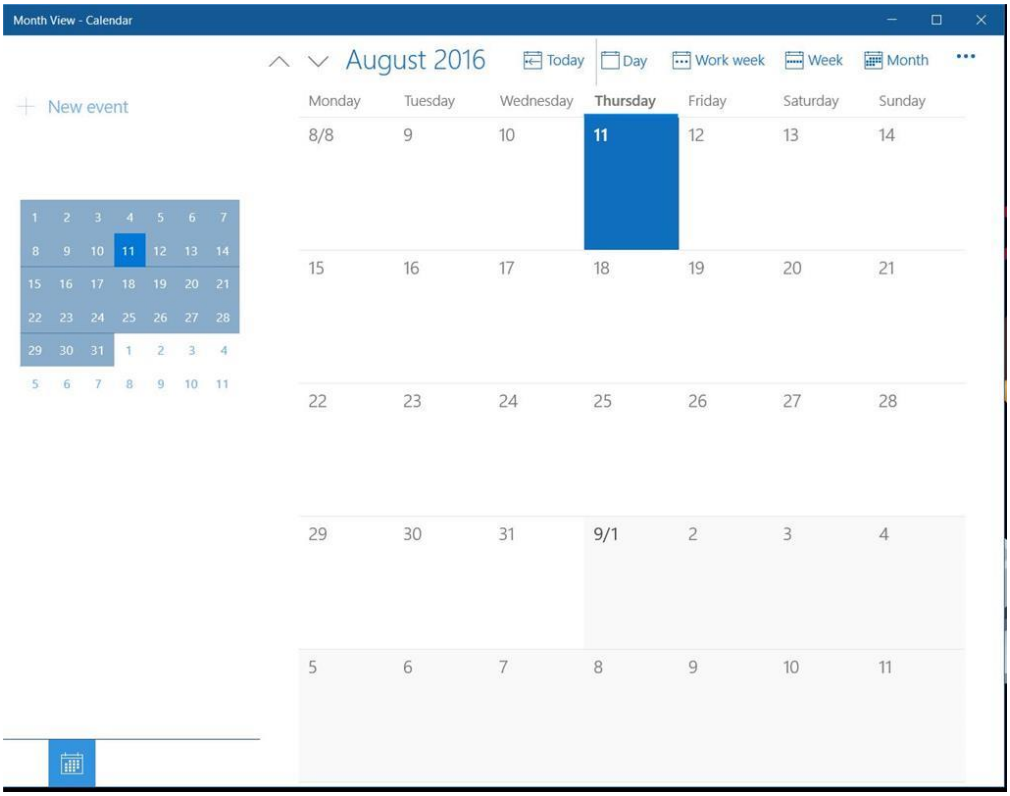

Month View - Calendar
August 2016
Today
Day
Work week
Week
Month
New event
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
8/8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 9/1 2 3 4
5 6 7 8 9 10 11

Figure 46: The Mail & Calendar app leverages both approaches based on the platform. On a mobile device, the application bar is placed at the bottom (top image), on the desktop it is placed at the top (bottom image).

Regardless of the position, a **CommandBar** offers two kinds of commands:

- **PrimaryCommands** are the most important ones and they are always visible. They are displayed with an icon and a label, which contains a brief description of the function.
- **SecondaryCommands** are related to secondary functions. By default, they're hidden, displayed only when the user expands the application bar by tapping on the three dots placed at the right side. They don't have an icon, but just a textual description.

The following sample defines a **CommandBar** placed at the bottom of the page:

Code Listing 154

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

    <Page.BottomAppBar>
        <CommandBar>
            <CommandBar.PrimaryCommands>
                <!-- primary commands -->
            </CommandBar.PrimaryCommands>
            <CommandBar.SecondaryCommands>
                <!-- secondary commands -->
            </CommandBar.SecondaryCommands>
        </CommandBar>
    </Page.BottomAppBar>

    <Grid>
        <!-- page content -->
    </Grid>
</Page>
```

The **CommandBar** supports three different types of commands:

- A button, which is identified by the **AppBarButton** class.
- A toggle button (which can maintain the on/off state), identified by the **AppBarToggleButton** class.
- A separator, which is useful to group the commands in different sections, identified by the **AppBarSeparator**.

The buttons' visual layout is defined thanks to two important properties: **Label** and **Icon**. Label is the text that is displayed below the button. **Icon** is the image displayed inside the button and it works in a different way than a standard **Image** control. By default, you can't simply specify an image path, but one of the symbols that belongs to the Segoe UI font family. You can find a list of all the available symbols in the MSDN documentation.

The following sample shows an **AppBarButton** control with a **Save** icon:

Code Listing 155

```
<Page.BottomAppBar>
    <CommandBar>
        <CommandBar.PrimaryCommands>
            <AppBarButton Label="Save" Icon="Save" />
        </CommandBar.PrimaryCommands>
    </CommandBar>
</Page.BottomAppBar>
```

However, if you can't find the right icon for you, there's still a way to use your own images as icons by using the extended syntax to define the **Icon** property, like in the following sample:

Code Listing 156

```
<Page.BottomAppBar>
    <CommandBar>
        <CommandBar.PrimaryCommands>
            <AppBarButton Label="Save">
                <AppBarButton.Icon>
                    <BitmapIcon UriSource="/Assets/image.png" />
                </AppBarButton.Icon>
            </AppBarButton>
        </CommandBar.PrimaryCommands>
    </CommandBar>
</Page.BottomAppBar>
```

When it comes to managing the user interaction with the application bar's commands, there are no differences from a regular **Button** control. They expose, in fact, a **Click** event that you can subscribe in code-behind to perform some operations when the button is clicked. The following code shows a complete **CommandBar** sample:

Code Listing 157

```
<Page.BottomAppBar>
    <CommandBar>
        <CommandBar.PrimaryCommands>
            <AppBarButton Label="refresh" Click="OnButton1Clicked"
Icon="Refresh" />
            <AppBarSeparator />
```

```
            <AppBarToggleButton Label="add" Click="OnButton2Clicked"
Icon="Favorite" />
        </CommandBar.PrimaryCommands>
        <CommandBar.SecondaryCommands>
            <AppBarButton Label="save" Click="OnButton3Clicked" Icon="Save"
/>
        </CommandBar.SecondaryCommands>
    </CommandBar>
</Page.BottomAppBar>
```

The Anniversary Update has added a new feature in the **CommandBar** control to make it easier to create adaptive layouts. We've seen, so far, that when you add items to the bar, you must split them between primary and secondary commands. However, this approach has a downside when it comes to creating a user interface that can adapt to multiple devices and screen sizes. No matter how much space your application can use, secondary commands will always be hidden unless the user taps on the three dots.

The Anniversary Update has added a new property called **IsDynamicOverflowEnabled**. When it's set to **True**, you can start adding items inside the **CommandBar** control without categorizing them as primary or secondary commands. Then, automatically, the **CommandBar** will try to render most of them as primary and, if there isn't space left anymore, it will move them to secondary.

You can influence which commands are less important for your scenario by assigning a numeric value to the **DynamicOverflowOrder** property offered by each control that you can place inside the **CommandBar**. Let's look at the following sample implementation:

Code Listing 158

```
<Page.BottomAppBar>
    <CommandBar IsDynamicOverflowEnabled="True">
        <AppBarButton Label="Favorite" Icon="Favorite" />
        <AppBarButton Label="Pin" Icon="Pin" />
        <AppBarButton Label="Website" Icon="World" />
        <AppBarButton Label="Buy" Icon="Shop" DynamicOverflowOrder="1" />
        <AppBarButton Label="Trailer" Icon="Play" DynamicOverflowOrder="2"
/>
        <AppBarButton Label="Similar shows" Icon="People"
DynamicOverflowOrder="3" />
    </CommandBar>
</Page.BottomAppBar>
```

As you can see, this **CommandBar** contains six **AppBarButton** controls. Three of them have the the **DynamicOverflowOrder** property set to a numeric value, which means that, as soon as the window starts to shrink, they will be the first to be moved to secondary commands, based on the order we've defined (so the Buy button will be the first to be moved, followed by the Trailer and Similar Shows ones).

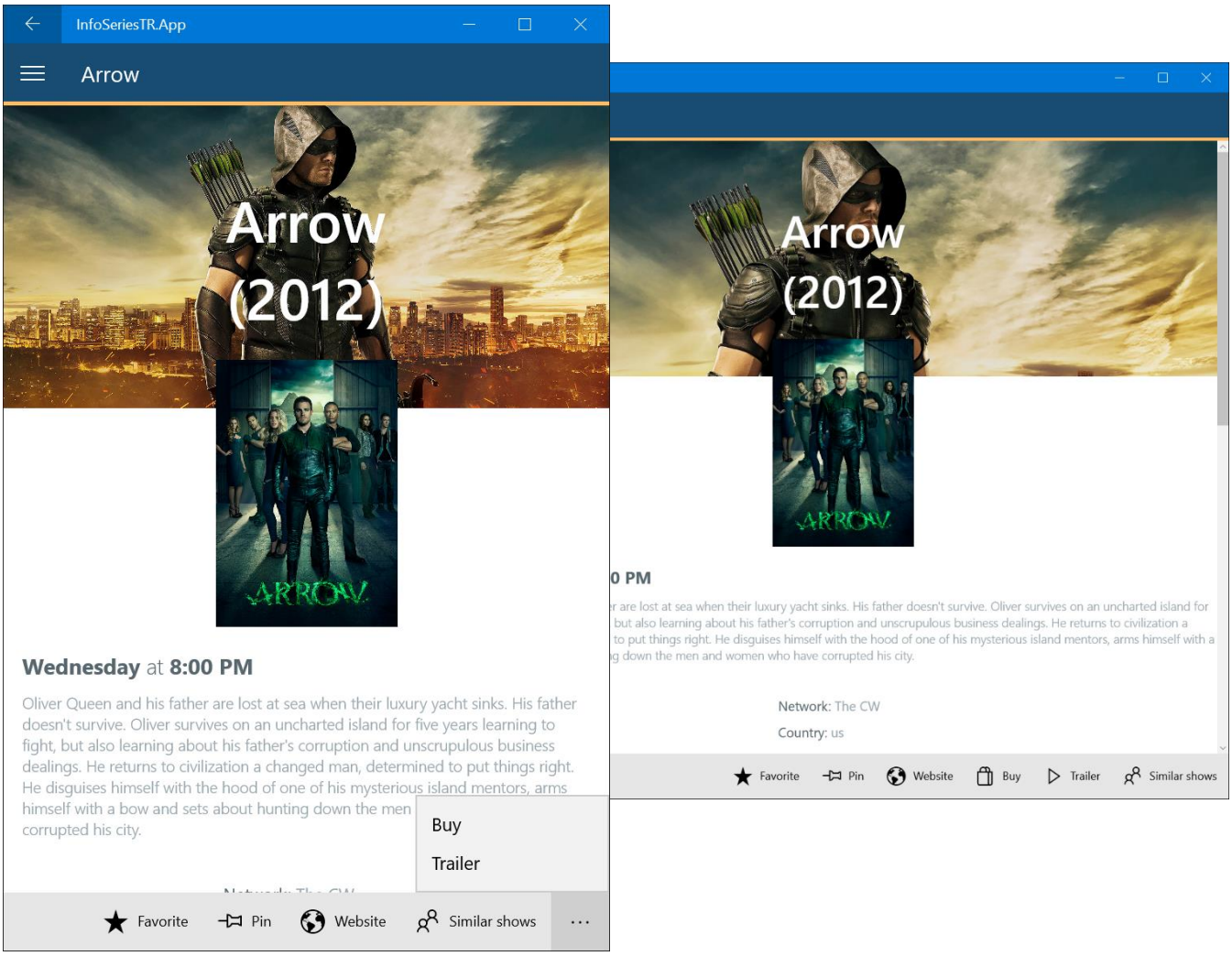


Figure 47: The two images show the same page, but displayed at two different screen sizes. Automatically, some of the buttons in the application bar have been moved to secondary commands in the smaller window.

StatusBar (Windows Mobile only)

The **`StatusBar`** is a control currently available only on the mobile platform, since it's the system tray placed at the top where the operating system shows some important information, like the time, the signal strength, etc. To get access to this control, you first need to add a reference to the **Windows Mobile Extensions for Mobile** library that you can find when you right-click on your project, choose **Add reference,** and move to the section **Universal Windows -> Extensions**. The Universal Windows Platform allows developers to interact with this bar to perform additional operations.

However, since it's a control available only for mobile, you can't interact with it directly from the XAML. We need to manage it in code-behind, by using the **`StatusBar`** class (part of the

Windows.UI.ViewManagement namespace), which offers a method called **GetForCurrentView()** that returns a reference to the bar.

However, remember that you need to leverage the capability detection approach described in Chapter 1 before using any of these APIs. In fact, if you tried to call one of the following APIs on a desktop or on a console, you would get an exception, since this control isn't implemented on these platforms.

As such, most of the time you're going to write code like the following:

Code Listing 159

```
private void CustomizeSystemTray()
{
    var api = "Windows.UI.ViewManagement.StatusBar";
    if (ApiInformation.IsTypePresent(api))
    {
        StatusBar statusBar = StatusBar.GetForCurrentView();
        //do something with the status bar
    }
}
```

Here are some of the main operations you can do once you have a reference to the current bar.

Hiding and showing the status bar

As a developer, you're able to hide the status bar so that the application can use all the available space on the screen. This feature can be useful also if your application is using a custom theme, which would look bad mixed together with the top bar, since by default it keeps the same theme color of the phone (black or white).

However, you must be cautious in using this feature. Hiding the status bar means that some important information will be hidden unless the user decides to pull down the Action Center from the top of the screen. Consequently, for example, if your application uses the data connection heavily, it's not a good idea to hide the status bar because the user won't be able to tell immediately if issues are being caused by a network problem (like she doesn't have cellular signal).

To hide the status bar, it's enough to call the **HideAsync()** method, like in the following sample:

Code Listing 160

```
private async void OnChangeSystemTrayClicked(object sender, RoutedEventArgs
e)
{
    StatusBar statusBar = StatusBar.GetForCurrentView();
    await statusBar.HideAsync();
}
```

If you want to display it again, just call the **ShowAsync()** method:

Code Listing 161

```
private async void OnChangeSystemTrayClicked(object sender, RoutedEventArgs
e)
{
    StatusBar statusBar = StatusBar.GetForCurrentView();
    await statusBar.ShowAsync();
}
```

Changing the look and feel

If you don't want to hide the bar, but you still don't want your application to look bad because the status bar color doesn't mix well with your theme, the **StatusBar** control offers more choices.

One approach is to change the bar's opacity. This way, you'll be able to make it transparent, so that the various indicators (time, signal strength, etc.) will still be visible, but they will be displayed over your application's theme, without interfering with it. To achieve this, you need to set the **BackgroundOpacity** property with the opacity value you prefer, like in the following sample:

Code Listing 162

```
private void OnChangeSystemTrayClicked(object sender, RoutedEventArgs e)
{
    StatusBar statusBar = StatusBar.GetForCurrentView();
    statusBar.BackgroundOpacity = 0.4;
}
```

Another approach that needs to be used in combination with the **BackgroundOpacity** property is to change the background color by setting the **BackgroundColor** property, like in the following sample:

Code Listing 163

```
private void OnChangeSystemTrayClicked(object sender, RoutedEventArgs e)
{
    StatusBar statusBar = StatusBar.GetForCurrentView();
    statusBar.BackgroundOpacity = 1.0;
    statusBar.BackgroundColor = Colors.Red;
}
```

Showing a progress bar

Previously in this chapter, we learned how to use controls such as **ProgressBar** and **ProgressRing** to display the status of a running operation to the user. However, there's another

alternative on Windows 10. Instead of showing a progress bar inside the page, you can display it in the status bar at the top of the screen. In this case, you need to use the **ProgressIndicator** control, which, like the **ProgressBar**, supports two kind of animations. The default is an indeterminate one, which is used to keep track of operations for which you aren't able to determine the exact duration. Otherwise, if you want to display a standard progress bar, you can fill the **ProgressValue** property with a value between **0** (empty bar) and **1** (full bar).

Regardless of the behavior you choose, you can also add a text using the **Text** property, which is displayed below the progress bar, like in the following sample:

Code Listing 164

```
private async void OnChangeSystemTrayClicked(object sender, RoutedEventArgs
e)
{
    StatusBar statusBar = StatusBar.GetForCurrentView();
    statusBar.ProgressIndicator.Text = "Loading...";
    await statusBar.ProgressIndicator.ShowAsync();
}
```

Once you've configured the **ProgessIndicator** property exposed by the **StatusBar** control, you can display it by calling the **ShowAsync()** method or hide it by using the **HideAsync()** one.

Showing a dialog

Another common requirement in an application is to show a dialog message, for example, to ask the user for a confirmation that he wants to proceed with an operation.

There are two kinds of dialogs that can be leveraged in a Universal Windows Platform app: dialogs and flyouts.

Dialogs

Dialogs have a modal approach, they are usually displayed in the middle of the screen, and they force the user to make a choice. The dialog won't go away until the user has selected one of the available options.

The simplest dialog control is called **MessageDialog** and is typically used to display a warning or a message that the user can acknowledge simply by pressing the **Close** button.

Code Listing 165

```
private async void OnShowDialog(object sender, RoutedEventArgs e)
{
    MessageDialog dialog = new MessageDialog("Internet connection not
available. Try again later.", "Error");
    await dialog.ShowAsync();
}
```

We create a new **`MessageDialog`** object, passing as parameters the content (required) and the title (optional). In the end, we call the **`ShowAsync()`** method to show the dialog.

The following image shows the result of the previous code:

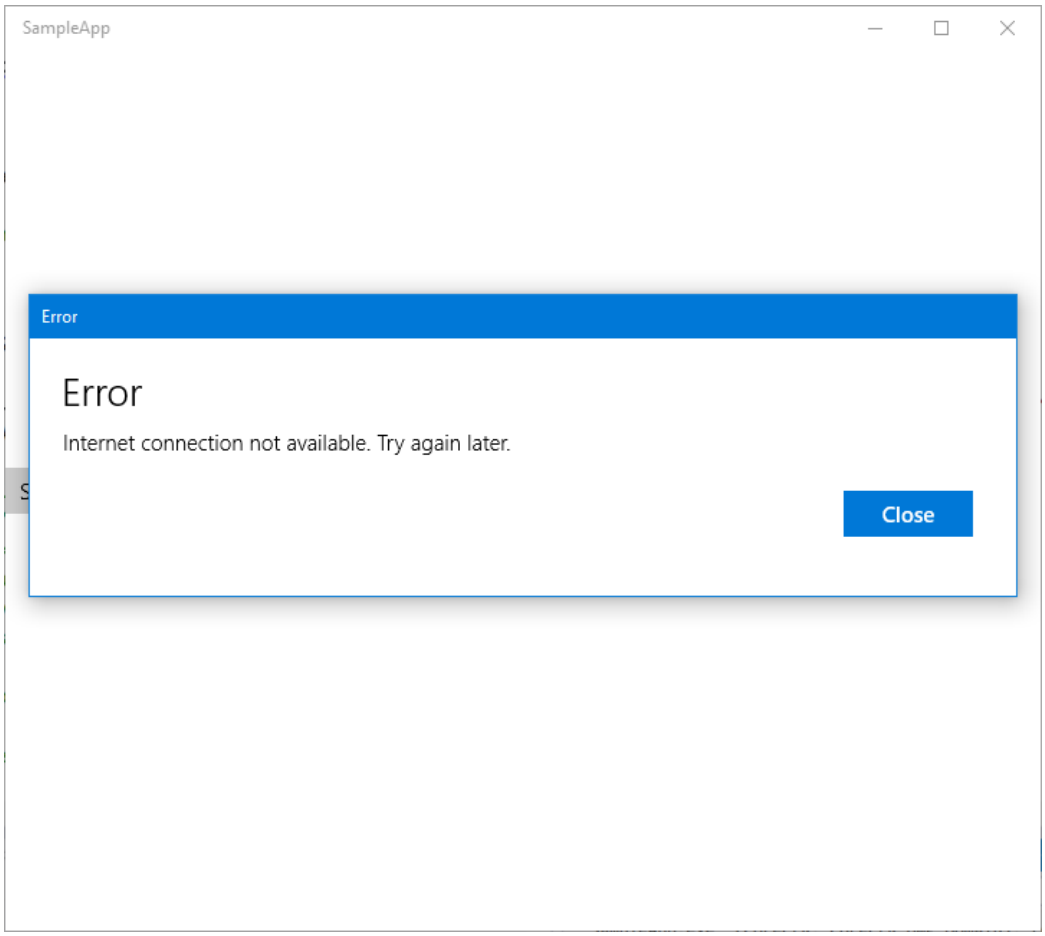


Figure 48: A simple MessageDialog.

A more customizable type of dialog is the **`ContentDialog`** class, which can offer customized content and multiple buttons. Typically, it's used when you want to present a choice to the user. Look at the following code:

Code Listing 166

```
private async void OnShowDialog(object sender, RoutedEventArgs e)
{
    ContentDialog dialog = new ContentDialog();
    dialog.Title = "Cancel item";
    dialog.Content = "Do you want to cancel the selected item?";
    dialog.PrimaryButtonText = "Ok";
    dialog.SecondaryButtonText = "Cancel";
    ContentDialogResult result = await dialog.ShowAsync();

    if (result == ContentDialogResult.Primary)
```

```
    {
        //delete the item
    }
    else
    {
        //cancel the operation
    }
}
```

We create a new **ContentDialog** object and, other than assigning a **Title** and **Content** like with the **MessageDialog** class, we also set the text of the two buttons that will be displayed in the dialog by using the **PrimaryButtonText** and the **SecondaryButtonText** properties.

In this case, when we call the **ShowAsync()** method, we also get back a result that is an enumerator of type **ContentDialogResult**. Thanks to this return value, we can understand which button has been pressed and act in the proper way. The following image shows the previous code in action:

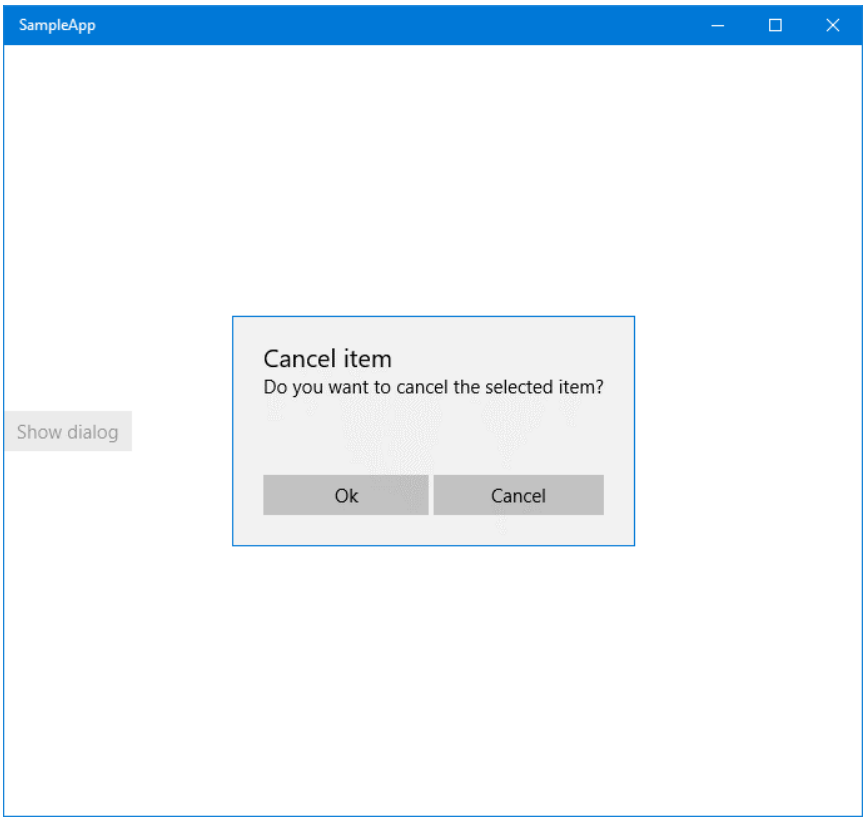


Figure 49: A simple ContentDialog.

If you want, you can also have deeper control over the look and feel of the **ContentDialog** layout by leveraging a specific item template available in the Universal Windows Platform. In this case, instead of leveraging the base **ContentDialog** class in C# code like we did before, we can use the full power of XAML to define the aspect of the dialog.

To achieve this result, you can right-click on your project and choose **Add -> New item -> ContentDialog**. You will get a new XAML file with the following definition:

Code Listing 167

```
<ContentDialog
    x:Class="SampleApp.CustomContentDialog"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:SampleApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="Cancel Item"
    PrimaryButtonText="Ok"
    SecondaryButtonText="Cancel"
    PrimaryButtonClick="ContentDialog_PrimaryButtonClick"
    SecondaryButtonClick="ContentDialog_SecondaryButtonClick">

    <Grid>
        <!-- content of the dialog -->
    </Grid>
</ContentDialog>
```

As you can see, it isn't a regular page, since the root node is called **`ContentDialog`**. However, it behaves like one. It has a XAML file (with the user interface definition) and a .cs file (which handles the logic and contains C# code). You can also see how the properties that we previously set in code (like **`Title`** or **`PrimaryButtonText`**) can now be set directly in XAML.

The main differences compared to the previous approach are:

- The **`Content`** can be expressed, instead of with a simple string, with the full XAML stack. We can place any XAML content we want inside the **`Grid`** and, therefore, completely customize the look and feel of the dialog.

- By default, the dialog offers two events raised when the user clicks on the two buttons (**`PrimaryButtonClick`** and **`SecondaryButtonClick`**). This way, other than just knowing which button has been pressed, we can also include some additional logic (for example, if the user presses the button to confirm that he wants to cancel the selected item, we could store some data in the local settings).

The following sample shows a completely customized **`ContentDialog`**, where the **`Content`** isn't just a plain string, but is made by various XAML controls, like a **`StackPanel`**, an **`Image`**, and a **`TextBlock`**:

Code Listing 168

```
<ContentDialog
    x:Class="SampleApp.CustomContentDialog"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:local="using:SampleApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="Question"
    PrimaryButtonText="Yes"
    SecondaryButtonText="No"
    PrimaryButtonClick="ContentDialog_PrimaryButtonClick"
    SecondaryButtonClick="ContentDialog_SecondaryButtonClick">

    <Grid>
        <StackPanel Orientation="Horizontal">
            <Image Source="/Assets/image.jpg" Width="100" />
            <TextBlock Text="Do you like this logo?" />
        </StackPanel>
    </Grid>
</ContentDialog>
```

To display this dialog, you just need to create a new instance of the custom dialog you've created and then call the **`ShowAsync()`** method. As you can see from the **`x:Class`** attribute, this dialog is connected to a code-behind class called **`CustomContentDialog`**, so here is how we can display it in C# code:

Code Listing 169

```
private async void OnShowDialog(object sender, RoutedEventArgs e)
{
    CustomContentDialog dialog = new CustomContentDialog();
    ContentDialogResult result = await dialog.ShowAsync();

    if (result == ContentDialogResult.Primary)
    {
        //handle the positive feedback
    }
    else
    {
        //handle the negative feedback
    }
}
```

Flyouts

Flyouts are like dialogs, but they should be used to display transient information. They aren't modal, but they have a light dismiss behavior (which means that if the user clicks or taps outside the dialog, it will be automatically be dismissed, without forcing them to make a choice). A flyout is typically defined thanks to the **`Flyout`** property exposed by most of the XAML controls that handle the interaction with the user. The **`Button`** control is one of these, like you can see in the following sample:

Code Listing 170

```
<Button Content="Delete item">
    <Button.Flyout>
        <Flyout>
            <StackPanel>
                <TextBlock Text="Do you really want to delete it?" />
                <StackPanel Orientation="Horizontal">
                    <Button Content="Yes" Click="OnDelete" />
                    <Button Content="No" Click="OnCancel" />
                </StackPanel>
            </StackPanel>
        </Flyout>
    </Button.Flyout>
</Button>
```

Inside the **Flyout** property exposed by the **Button** control, we define a **Flyout** control that simply contains other XAML controls. In this case, we're asking the user if he really wants to delete an item when he presses the button. The difference from the **ContentDialog** control sample we've seen before is that, this time, the user won't be forced to make a choice. It's enough for them to click outside the flyout to hide the dialog. We don't need to invoke any C# code to display the flyout. It will be enough to press the button to show it. Here is how the previous code looks in a sample app:

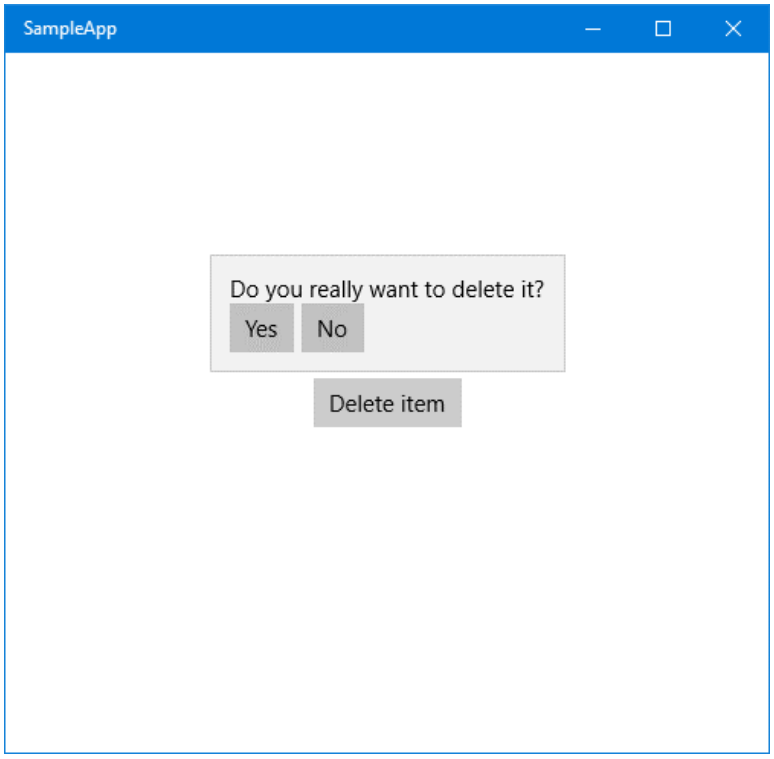


Figure 50: A Flyout control.

The main difference from the standard usage of the **Button** control is that, in this case, we won't handle the **Click** event of the **Button** control. Its purpose will be just to display the flyout. It will

be up to the two buttons contained in the flyout (Yes and No) to handle the **Click** event and to perform the expected logic (like deleting the item if the user has pressed Yes or canceling the operation if she has pressed No).

Context menus

The Universal Windows Platform offers a special version of the **Flyout** control, called **MenuFlyout**, that can be used to create context menus, which allow you to perform additional operations on a selected item. This concept is widely used on the desktop. When you right-click with the mouse on an item, you are prompted with a list of options you can choose from.

Every XAML control supports this kind of flyout thanks to the **ContextFlyout** property. The following sample shows a **MenuFlyout** added to a **TextBlock** control:

Code Listing 171

```
<TextBlock Text="This is an item">
    <TextBlock.ContextFlyout>
        <MenuFlyout>
            <MenuFlyoutItem Text="Open" Click="OnOpen" />
            <MenuFlyoutItem Text="Edit" Click="OnEdit" />
            <MenuFlyoutSeparator />
            <MenuFlyoutItem Text="Delete" Click="OnDelete" />
            <ToggleMenuFlyoutItem Text="Is active" />
        </MenuFlyout>
    </TextBlock.ContextFlyout>
</TextBlock>
```

As you can see, a **MenuFlyout** can contain different kinds of items:

- **MenuFlyoutItem** is simply an item of the list that has a **Text** property (the label) and a **Click** event (to handle the operation to perform when you select it).
- **MenuFlyoutSeparator** adds a separator that can be used to group different options in multiple categories.
- **ToggleMenuFlyoutItem** behaves like a **ToggleButton** control and can assume two statuses, active or inactive, based on the value of the **IsChecked** property.

The following image shows an application that implements the previous code:

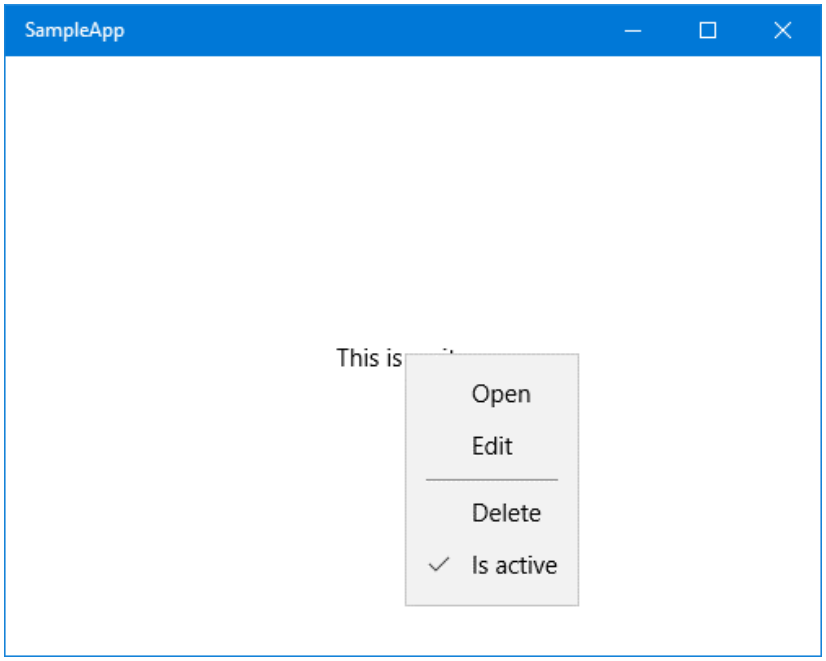


Figure 51: A MenuFlyout control.