THE MICROSOFT JOURNAL FOR DEVELOPERS

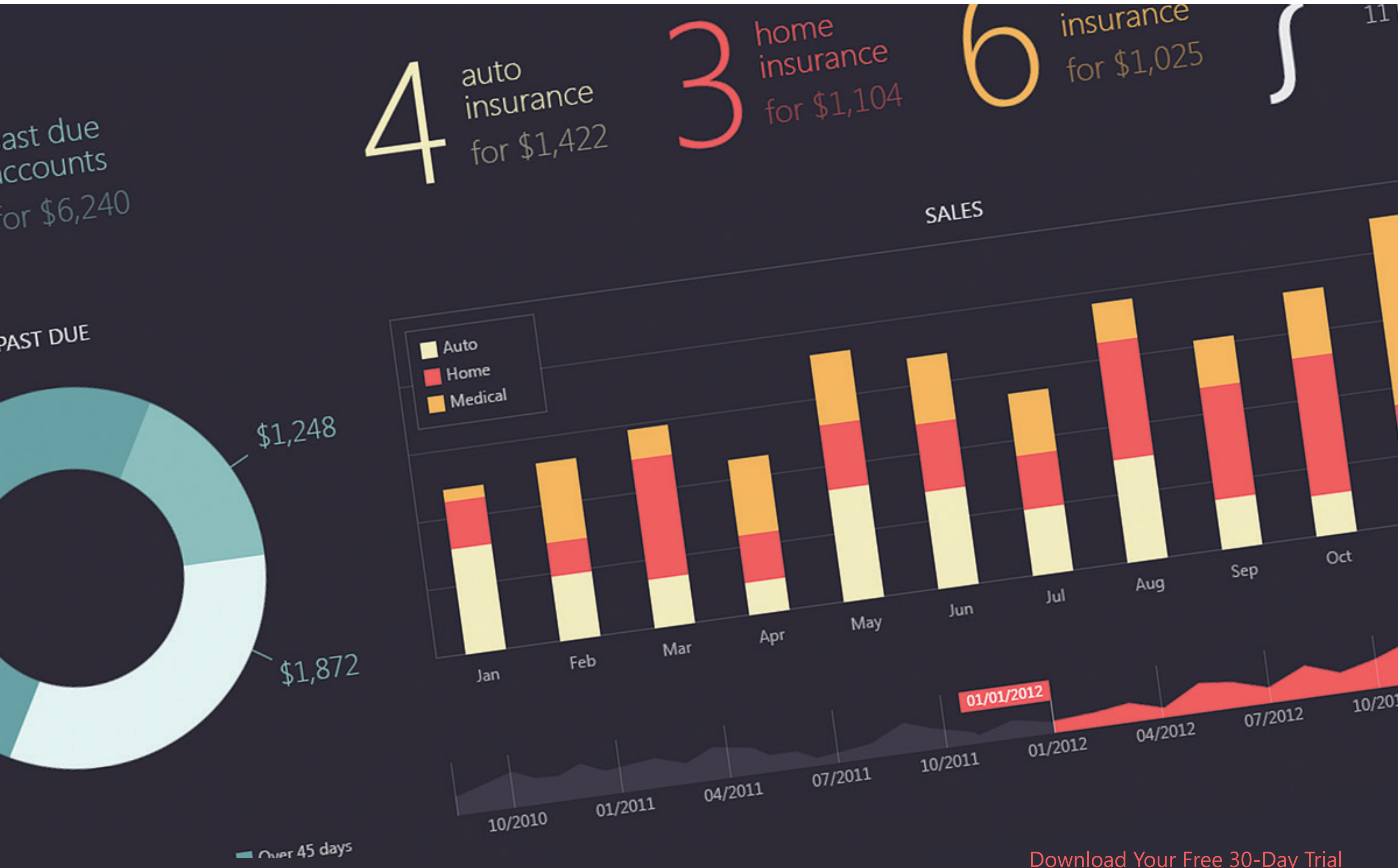
OCTOBER 15, 2015 VOL 30 NO 11

msdn magazine

Special Issue

Your Next Great Dashboard Starts Here

Create high impact and information rich decision support systems for desktops and the web with the DevExpress Universal Subscription.



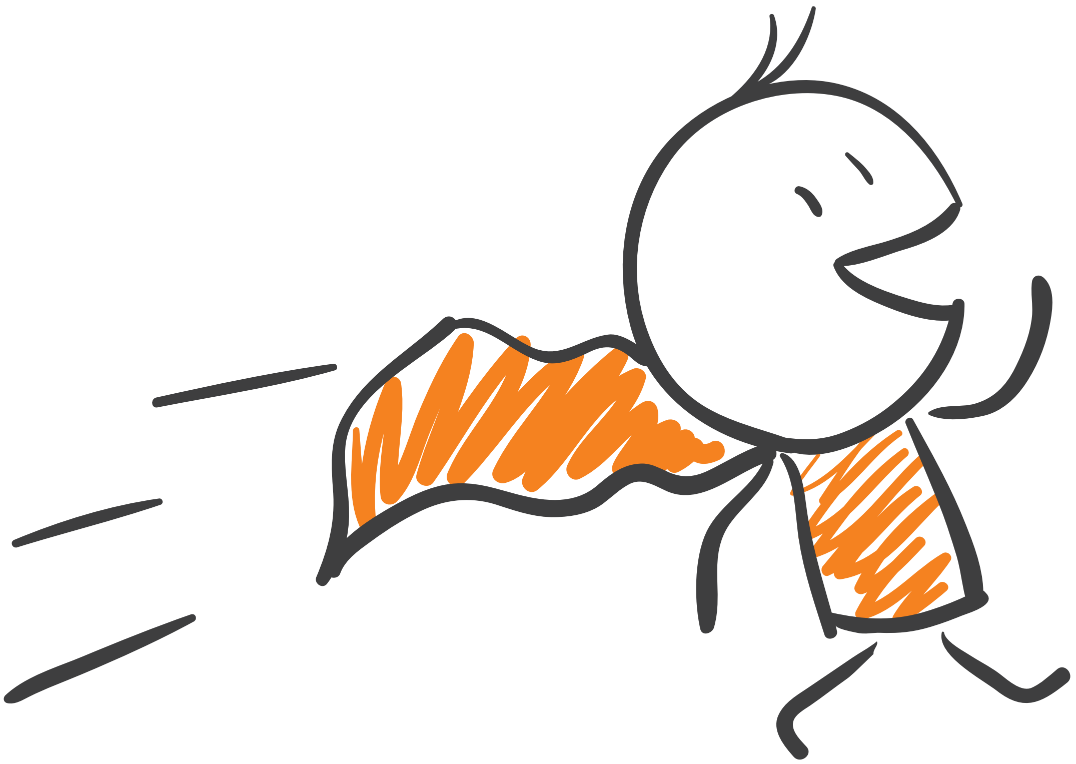
Download Your Free 30-Day Trial

devexpress.com/dashboard

DevExpress®

Unleash the UI Superhero in You

With DevExpress tools, you'll deliver amazing user-experiences for Windows®, the Web and Your Mobile World



Experience the DevExpress difference today.
Download your free 30-day trial.
devexpress.com/try

DevExpress® WIN ASP MVC WPF HTML

THE MICROSOFT JOURNAL FOR DEVELOPERS

OCTOBER 15, 2015 VOL 30 NO 11

msdn magazine

Special Issue

Windows 10

Microsoft

msdn magazine

OCTOBER 15, 2015 VOLUME 30 NUMBER 11

ID STATEMENT MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. $35.00, International $60.00. Annual digital subscription rates payable in U.S. funds are: U.S. $25.00, International $25.00. Single copies/back issues: U.S. $10, all others $12. Send orders with payment to: MSDN Magazine, P.O. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: P.O. Box 201, Richmond Hill, ON L4B 4R5, Canada.

CORPORATE ADDRESS 1105 Media, 9201 Oakdale Ave. Ste 101, Chatsworth, CA 91311 www.1105media.com

MEDIA KITS Direct your Media Kit requests to Chief Revenue Officer Dan LaBianca, 972-687-6702 (phone), 972-687-6799 (fax), dlabianca@1105media.com

REPRINTS For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International Phone: 212-221-9595. E-mail: 1105reprints@parsintl.com. www.magreprints.com/QuickQuote.asp

LIST RENTAL This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Jane Long, Merit Direct. Phone: 913-685-1301; E-mail: jlong@meritdirect.com; Web: www.meritdirect.com/1105

Reaching the Staff
Staff may be reached via e-mail, telephone, fax, or mail. A list of editors and contact information is also available online at Redmondmag.com.
E-mail: To e-mail any member of the staff, please use the following form: FirstinitialLastname@1105media.com
Irvine Office (weekdays, 9:00 a.m. – 5:00 p.m. PT)
Telephone 949-265-1520; Fax 949-265-1528
4 Venture, Suite 150, Irvine, CA 92618
Corporate Office (weekdays, 8:30 a.m. – 5:30 p.m. PT)
Telephone 818-814-5200; Fax 818-734-1522
9201 Oakdale Avenue, Suite 101, Chatsworth, CA 91311
The opinions expressed within the articles and other contentsherein do not necessarily express those of the publisher.

Microsoft

BPA WORLDWIDE

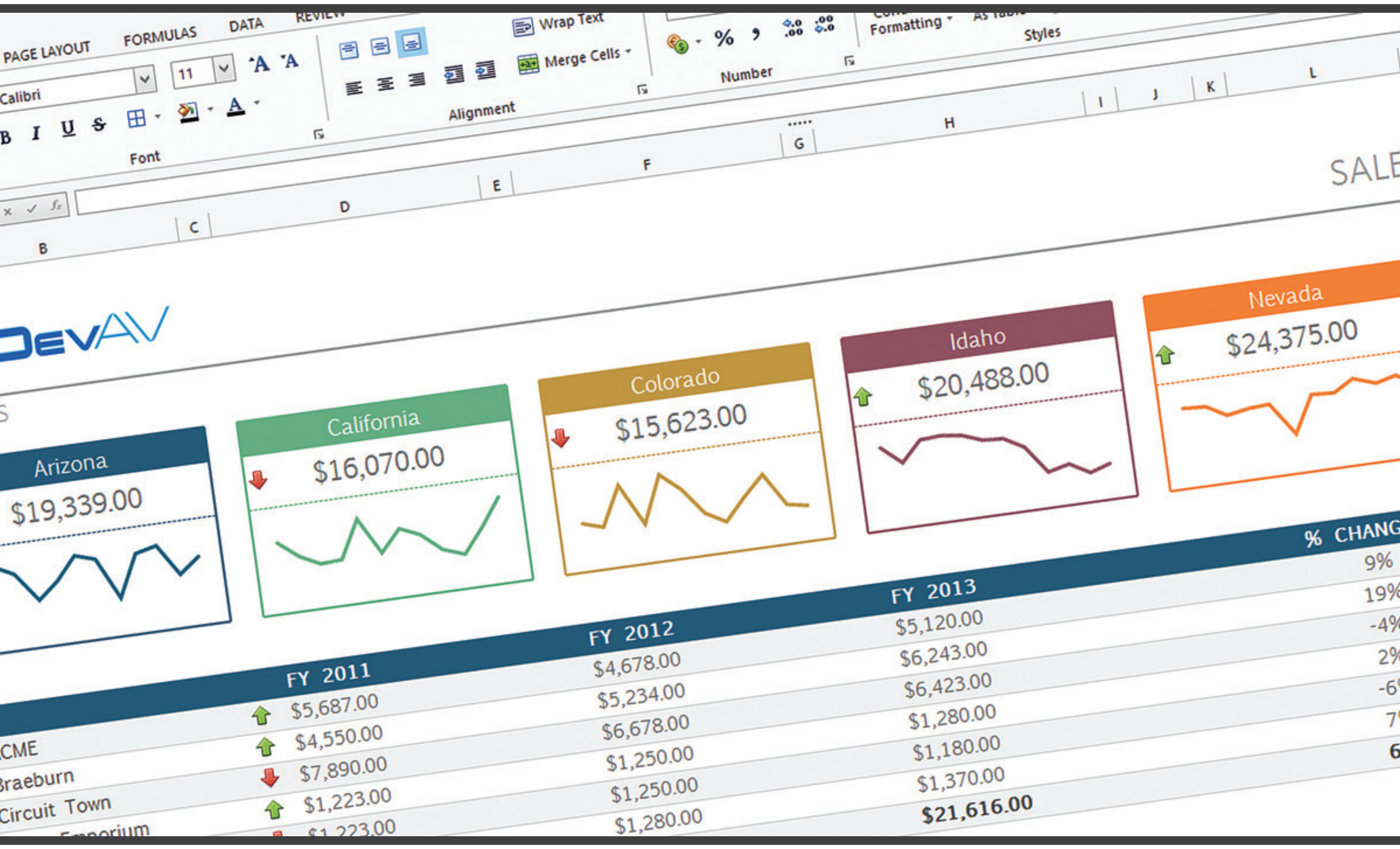
1906 100 2006 AMERICAN BUSINESS MEDIA A Century Moving Business Forward

WPA Western Publications Association

DevExpress

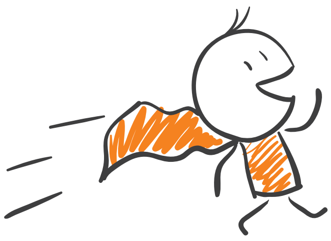
Office® Inspired Apps

Get started today and create high-performance, high-impact .NET solutions that fully replicate the look, feel and user-experience of Microsoft Office.®



Arizona	California	Colorado	Idaho	Nevada
$19,339.00	$16,070.00	$15,623.00	$20,488.00	$24,375.00

	FY 2011	FY 2012	FY 2013	% CHANG
CME	$5,687.00	$4,678.00	$5,120.00	9%
raeburn	$4,550.00	$5,234.00	$6,243.00	19%
Circuit Town	$7,890.00	$6,678.00	$6,423.00	-4%
Emporium	$1,223.00	$1,250.00	$1,280.00	2%
	$1,223.00	$1,250.00	$1,180.00	-6
		$1,280.00	$1,370.00	7
			$21,616.00	6



Your Next Great Business App Starts Here

Explore our complete range of Office-inspired controls for all major .NET platforms.
devexpress.com/office

A Tale of Two Windows

A quarter century ago Microsoft released the most important piece of software in its long history. Windows 3.0 was not the first version of Windows, nor was it the best, but it *was* the most critical in terms of its ultimate impact. At a time when leadership of the PC industry was up for grabs, Windows 3.0 did more than cement Microsoft's dominance in the space. It set the course for the evolution of the PC platform—and by extension, software development and IT—for two decades to come.

Before Windows 3.0 entered the market, the PC platform could have gone in a very different direction. Windows 2.0 was just another graphical operating environment, and products like GEM and DESQview boasted innovative UIs and multi-tasking capabilities. IBM, meanwhile, had convinced Microsoft to help it develop OS/2 as a successor OS to DOS.

A quarter century ago Microsoft was a potent young upstart that found a way to breathe new life into a faltering Windows franchise and steer the course of personal computing for two decades to come.

All that changed when a pair of Microsoft engineers—David Weisse and Murray Sargent—spent a month in 1988 quietly hacking the Windows 2.x kernel, DLL libraries and other assets to get Windows to run in protected mode on a 286 PC. (You can read Sargent's wonderful retelling at bit.ly/106glhy.) Freed from the 640KB memory barrier, and enabled by some capable driver and graphics work, Windows suddenly became something much bigger than it was. Steve Ballmer and Bill Gates, belatedly informed of the skunkworks effort, recognized it immediately. Windows 3.0 was approved as an official project, and within two years of its 1990 release would sell more than 10 million copies.

Windows 3.0 did a great many things. It established a look and feel for Microsoft OSes that would endure until the release of Windows 8 in 2012. And it restored Microsoft's approach to the burgeoning PC market, prompting it to develop its own graphical OS. But, ultimately, it proved out the adaptability of the platform itself.

A quarter century ago Microsoft was a potent young upstart that found a way to breathe new life into a faltering Windows franchise and steer the course of personal computing for two decades to come. The echoes of that achievement can be heard today. Windows 10 is the third iteration of Microsoft's new-look OS and represents a major step forward. From the refined and articulated dual-mode UI to the impressive scope of Microsoft's Universal Windows Platform (UWP) vision, Windows 10 in many ways does what Windows 3.0 achieved 25 years before. It extends the horizons of the platform, and offers an inclusive path forward for developers and end users alike.

Today, a Windows developer can write an app and have it run on *any* modern, Windows-enabled client, from the smallest Internet of Things device to the largest Surface Hub room-based display, and to every phone, tablet and PC in between. Windows 10 also presents bridges to Android and iOS mobile app developers and provides full support for traditional .NET and COM-based applications via virtualization.

A quarter century ago, we stood at the precipice of the graphical age. Now, as we enter the era of ubiquitous, device-based computing and its reliance on touch- and speech-based UIs, the question begs: Might there be a Windows 3.0 for the modern era?

[illegible]

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

Switch to Amyuni PDF

AMYUNI

NEW v5.5

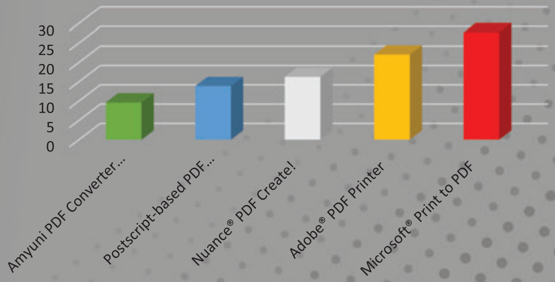
Create and Edit PDFs in .NET, COM/ActiveX, WinRT & UWP

- Edit, process and print PDF 1.7 documents
- Create, fill-out and annotate PDF forms
- Fast and lightweight 32- and 64-bit components for .NET and ActiveX/COM
- New Universal Apps DLLs enable publishing C#, C++, CX or Javascript apps to windows Store
- Updated Postscript/EPS to PDF conversion module

High Performance PDF Printer for Desktops and Servers

- Print to PDF in a fraction of the time needed with other tools. WHQL tested for all Windows platforms. Version 5.5 updated for Windows 10 support

Benchmark Testing - Amyuni vs Others
Seconds required to convert a document to PDF



Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
- Includes our Microsoft WHQL certified PDF Converter printer driver
- Export PDF documents into other formats such as Jpeg, PNG, XAML or HTML5
- Import and Export XPS files using any programming environment

Other Developer Components from Amyuni®

- WebkitPDF: Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver
- PDF2HTML5: Conversion of PDF to HTML5 including dynamic forms
- Postscript to PDF Library: For document workflow applications that require processing of Postscript documents
- OCR Module: Free add-on to PDF Creator uses the Tesseract engine for character recognition
- Javascript engine: Integrate a full Javascript interpreter into your applications to process PDF files or for any other need

CERTIFIED FOR Windows Server® 2008

Windows Server 2012 Certified

Windows® 7

Windows

All development tools available at
www.amyuni.com

AMYUNI Technologies

USA and Canada
Toll Free: 1866 926 9864
Support: 514 868 9227
sales@amyuni.com

Europe
UK: 0800-015-4682
Germany: 0800-183-0923
France: 0800-911-248

.NET and Universal Windows Platform Development

Daniel Jacobson

With Visual Studio 2015, you can now use the latest .NET technology to build Universal Windows Platform (UWP) applications that run on all Windows 10 devices—including the phone in your pocket, the laptop or tablet in your bag, the Surface Hub in your office, the Xbox console in your home, HoloLens, and any other devices you can imagine on the Internet of Things (IoT). It's truly an exciting time to be a Windows developer.

What's New with the UWP?

As a .NET developer, you'll appreciate all that the UWP has to offer. UWP apps will run in "Windowed" mode on the huge number of desktops that have been, and will continue to be, upgraded to Windows 10. UWP apps will be able to reach all Windows 10 devices with one application package and one code base. In addition, UWP apps take advantage of the new Microsoft .NET Core Framework (explained in detail later in this article). Your .NET business logic can run on other platforms that support .NET Core, such as ASP.NET 5. UWP apps deploy a small copy of the .NET Core with the app, so the app will always run against the .NET version you tested it against. All .NET UWP apps take full advantage of .NET Native, which generates highly optimized native machine code, resulting in performance gains (also explained in this article).

This article discusses:

- Universal Windows Platform development using the latest .NET technology
- Changes to NuGet and how it relates to the .NET Framework
- .NET Native, and what it means to developers

Technologies discussed:

Microsoft .NET Framework, Visual Studio 2015, Universal Windows Platform, Windows 10

.NET Core Framework

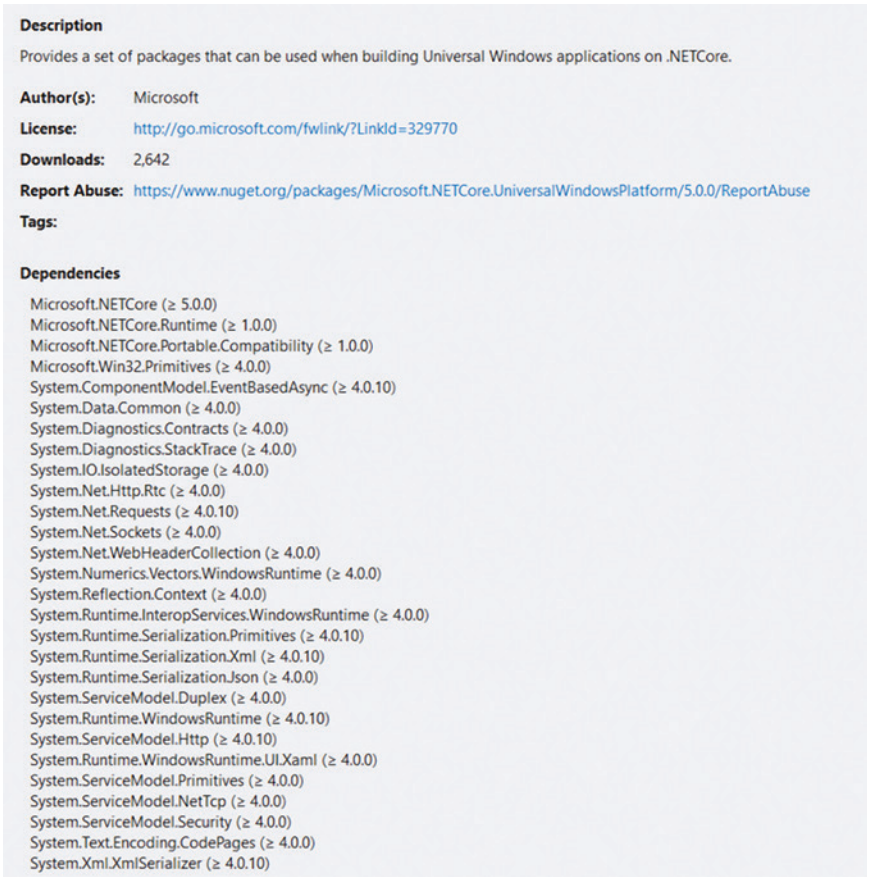
The .NET Core Framework is a new version of .NET for modern device and cloud workloads. It's a general-purpose and modular implementation of the Microsoft .NET Framework that can be ported and used in many different environments for a variety of workloads. In addition, the .NET Core Framework is open source and available on GitHub (github.com/dotnet/corefx) and is supported by Microsoft on Windows, Linux, and Mac OS X. If you're a UWP developer utilizing the latest .NET technology, this brings you huge advantages. In Visual Studio 2015, you can utilize .NET Core portable class libraries (PCLs) to target any UWP app, .NET 4.6 app or ASP.NET 5 app—even those that are cross-platform.

In addition, the .NET Core Framework is a superset of the .NET APIs that were previously available in Windows Store app development. This means UWP developers now have several additional namespaces available in their API arsenal. One such namespace is System.Net.Sockets, which is used for UDP communication. This was previously unavailable in Windows Runtime (WinRT) apps, and the workaround was to use the WinRT-specific UDP APIs. Now that Sockets is available in the .NET Core, you can utilize the same Socket code in your UWP apps and other .NET apps.

Another advantage is that the System.Net.Http.HttpClient API is built on the WinRT HTTP stacks. This provides the ability to use HTTP/2 by default if the server supports it, resulting in lower latency and fewer round-trip communications.

The Windows Communication Foundation (WCF) client (and the associated Add Service Reference dialog) was previously unavailable in Windows Phone .appx projects, but because it's a part of .NET Core, it can be used by all .NET UWP apps.

Finally, .NET Core is the underlying framework on which .NET Native depends. When .NET Native was designed, it was clear that the .NET Framework wouldn't be suitable as the foundation for the framework class libraries. That's because .NET Native statically links



Description

Provides a set of packages that can be used when building Universal Windows applications on .NETCore.

Author(s):	Microsoft
License:	http://go.microsoft.com/fwlink/?LinkId=329770
Downloads:	2,642
Report Abuse:	https://www.nuget.org/packages/Microsoft.NETCore.UniversalWindowsPlatform/5.0.0/ReportAbuse
Tags:	

Dependencies

Microsoft.NETCore (≥ 5.0.0)
Microsoft.NETCore.Runtime (≥ 1.0.0)
Microsoft.NETCore.Portable.Compatibility (≥ 1.0.0)
Microsoft.Win32.Primitives (≥ 4.0.0)
System.ComponentModel.EventBasedAsync (≥ 4.0.10)
System.Data.Common (≥ 4.0.0)
System.Diagnostics.Contracts (≥ 4.0.0)
System.Diagnostics.StackTrace (≥ 4.0.0)
System.IO.IsolatedStorage (≥ 4.0.0)
System.Net.Http.Rtc (≥ 4.0.0)
System.Net.Requests (≥ 4.0.10)
System.Net.Sockets (≥ 4.0.0)
System.Net.WebHeaderCollection (≥ 4.0.0)
System.Numerics.Vectors.WindowsRuntime (≥ 4.0.0)
System.Reflection.Context (≥ 4.0.0)
System.Runtime.InteropServices.WindowsRuntime (≥ 4.0.0)
System.Runtime.Serialization.Primitives (≥ 4.0.10)
System.Runtime.Serialization.Xml (≥ 4.0.10)
System.Runtime.Serialization.Json (≥ 4.0.0)
System.ServiceModel.Duplex (≥ 4.0.0)
System.Runtime.WindowsRuntime (≥ 4.0.10)
System.ServiceModel.Http (≥ 4.0.10)
System.Runtime.WindowsRuntime.UI.Xaml (≥ 4.0.0)
System.ServiceModel.Primitives (≥ 4.0.0)
System.ServiceModel.NetTcp (≥ 4.0.0)
System.ServiceModel.Security (≥ 4.0.0)
System.Text.Encoding.CodePages (≥ 4.0.0)
System.Xml.XmlSerializer (≥ 4.0.10)

Figure 1 **Viewing .NET Framework Libraries in NuGet**

the framework with the application, and then removes the extra stuff that isn't needed by the application. (This is a gross simplification, but you get the idea. For more details, take a look at "Inside .NET Native" at `bit.ly/1UR7ChW`.)

The traditional .NET Framework implementation isn't factored, which makes it a challenge for a linker to reduce the amount of framework that gets compiled into the application. The .NET Core is, after all, essentially a fork of the .NET Framework whose implementation is optimized around factoring concerns. Another benefit of this implementation is the ability to ship the .NET Core Framework as a set of NuGet packages, such that you can update individual classes out-of-band from the .NET Framework. Before going any further, though, let's talk about the changes in NuGet.

What's New in NuGet?

With the UWP, NuGet 3.1 support is built-in. Included in this release are features that improve package dependency management and a local caching of your packages for reuse in multiple projects.

With NuGet 3.1, the package-dependency declaration model has been updated. Beginning with ASP.NET 5, NuGet introduced support for the project.json file, and this is the same model the UWP supports. Project.json enables you to describe the dependencies of a project with a clear definition of the packages on which you immediately depend. Because the format is the same for ASP.NET 5 and the UWP, you can use the same file to define package references for both platforms, as well as PCLs.

Changing from packages.config to project.json enables you to "reboot" the references in your projects, and there's now a new transitive dependency capability of NuGet. If you reference a package that references another NuGet package, it used to be difficult to manage package versioning. For example, NHibernate is a package that depends on Iesi.Collections. In the packages.config, you'd have two references, one for each package. When you want to update NHibernate, do you also update Iesi.collections? Or, if there's an update for Iesi.collections, do you also have to update NHibernate to support the new features? It turned into a messy cycle, and package version management was difficult. The transitive dependencies feature of NuGet abstracts this decision to update package references with improved semantic versioning in the package definition files (nuspecs).

In addition, NuGet now downloads and stores a copy of the packages you use in a global packages folder located in your %userprofile%\.nuget\packages folder. Not only will this improve performance in package referencing because you'll only need to download each package once, but it should also reduce the disk space used on your workstation as you can share the same package binaries from project to project.

NuGet and .NET Core

What happens when you take the factoring-focused .NET Core and combine it with the package-dependency management of NuGet 3.1? You get the ability to update individual .NET Framework packages out-of-band from the rest of the .NET Framework. With the UWP, the .NET Core is included as a set of NuGet packages in your app. When you create a new project, you'll see only the general Microsoft.NETCore.UniversalWindowsPlatform package dependency, but if you look at this package on NuGet, you'll see all of the .NET Framework libraries that are included, as shown in **Figure 1**.

For example, say there's an update to the System.Net.Sockets class that introduces a new API you'd like to use in your application. With traditional .NET, your application would need to take a dependency on a new build of the entire .NET Framework. With the UWP and .NET Core with NuGet, you can update your NuGet dependencies to include the latest version of just that package. Then, when your application is compiled and packaged, that version of the framework library will get included in your application. This gives you the flexibility to use the latest and greatest .NET technology, without forcing your users to always have the latest framework installed on their devices.

In addition to being able to update your .NET classes at your own cadence, the componentized nature of .NET Core also enables .NET Native, providing performance benefits for all Windows 10 C# and Visual Basic applications when delivered to consumer devices.

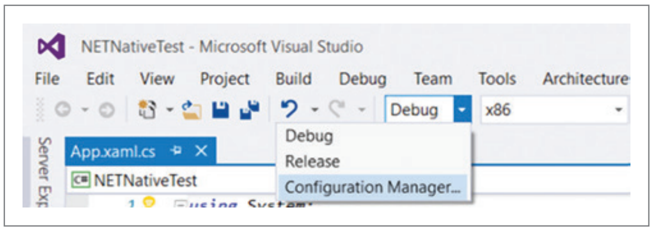

NETNativeTest - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture
Debug
x86
App.xaml.cs
NETNativeTest
Debug
Release
Configuration Manager...

Figure 2 **Opening the Configuration Manager**

What Is .NET Native?

Now that you know the .NET Core Framework enables .NET Native, I'll explain in more detail what it is and what it does for you as a UWP developer.

.NET Native is an ahead-of-time (AOT) compilation process—it turns your managed .NET code into native machine code at compile time. In contrast, traditional .NET uses just-in-time (JIT) compilation, which defers the native compilation of a method until its first execution at run time. .NET Native is more similar to a C++ compiler. In fact, it uses the Visual Studio C++ compiler as part of its tool chain. *Every* managed (C# or Visual Basic) Universal Windows app will utilize this new technology. The applications are automatically compiled to native code before they reach consumer devices. If you'd like to dive deeper into how it works, I highly recommend reading the MSDN Library article, "Compiling Apps with .NET Native," at bit.ly/1QcTGxm.

How Does .NET Native Impact You and Your App?

Your mileage likely will vary, but for most cases your app will start up faster, perform better, and consume fewer system resources. You can expect to see up to a 60 percent performance improvement on your applications the first time they start, and up to a 40 percent improvement on subsequent startup times ("warm" startup). Your applications will consume less memory when compiled natively. All dependencies on the .NET runtime are removed, so your end users will never need to break out of their setup experience to acquire the specific version of the .NET Framework your app references. In fact, all the .NET dependencies are packaged within your application, so the behavior of your app shouldn't change just because there's a change in the .NET Framework installed on the machine.

Even though your application is being compiled to native binaries, you still get to take advantage of the .NET languages you're familiar with (C# or Visual Basic), and the excellent tools associated with them. Finally, you can continue to use the comprehensive and consistent programming model available with the .NET Framework, with extensive APIs for business logic, built-in memory management and exception handling.

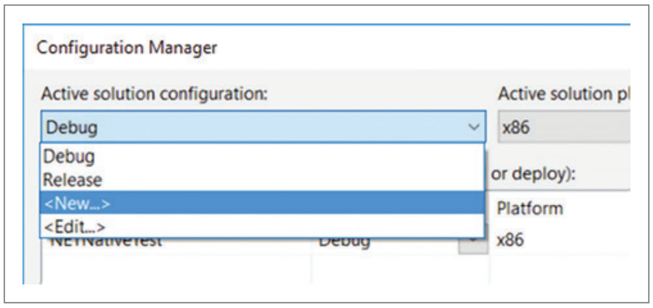


Figure 3 **Creating a New Configuration**

With .NET Native, you get the best of both worlds: managed development with C++ performance. How cool is that?

Debug Versus Release-Compile Configuration

.NET Native compilation is a complex process, and that makes it a little slower than classic .NET compilation. The benefits mentioned earlier come at the cost of compilation time. You could choose to compile natively every time you want to run your app, but you'd be spending extra time waiting for the build to finish. The Visual Studio tooling is designed to address this and create the smoothest possible developer experience.

When you build and run in "Debug" configuration, you're running Intermediate Language code against the CoreCLR packaged within your application. The .NET system assemblies are packaged alongside your application code, and your application takes a dependency on the Microsoft.NET.CoreRuntime (CoreCLR) package. If the CoreCLR framework is missing from the device you're testing on, Visual Studio will automatically detect that and install it before deploying your application.

This means you get the best development experience possible—fast compilation and deployment, rich debugging and diagnostics, and all of the tools you're accustomed to with .NET development.

When you switch to "Release" mode, by default your app utilizes the .NET Native toolchain. Because the package is compiled to native binaries, the package doesn't need to contain the .NET Framework libraries. Moreover, the package is dependent on the latest installed .NET Native runtime as opposed to the CoreCLR package. The .NET Native runtime on the device will always be compatible with your application package.

Local native compilation via the "Release" configuration will enable testing your application in an environment that's similar to what your customers will experience. It's important to test this on a regular basis as you proceed with development! By testing your application using the code generation and runtime technology your customers will experience, you'll make sure you've addressed all possible bugs (such as potential race conditions that will result from different performance characteristics).

A good rule of thumb is to test your app this way periodically throughout development to make sure you identify and correct any issues that might come from the .NET Native compiler. There should be no issues in the majority of cases; however, there are still a few things that don't play so nicely with .NET Native. Four-plus dimensional arrays are one example. Ultimately, your customers will be getting the .NET Native compiled version of your application, so it's always a good idea to test that version throughout development and before shipping.

In addition to testing with .NET Native compilation, you might also notice that the AnyCPU build configuration has disappeared. With .NET Native, AnyCPU is no longer a valid build configuration because native compilation is architecture-dependent. An additional consequence of this is that when you package your application, you should select all three architecture configurations (x86, x64 and ARM) to make sure your application is applicable to as many devices as possible. This is the *Universal* Windows Platform, after all.

NEVRON www.nevron.com

NEVRON
OPEN VISION for .NET

The Complete UI Suite for .NET

Develop cutting edge User Interfaces from a single code base and embed them in WinForms, WPF, Silverlight, Xamarin.Mac and MonoMac projects.

free

NOV User Interface for .NET
60+ widgets for rich client side development that are completely free.

NOV Text Editor for .NET
Advanced Microsoft Word-like text editor that can display HTML as well. Suitable for displaying, editing and converting large documents.

new

NOV Diagram for .NET
Advanced Microsoft Visio-like diagram with full support for smart shapes and many interactivity features.

new

NOV Map for .NET
Display geographical data from ESRI shape files with build-in data analysis.

NOV Barcode for .NET
Display a large variety of Linear and Matrix barcodes.

NOV Gauge for .NET
Advanced Linear and Radial gauges for display of KPIs and Dashboards.

coming soon

NOV Chart for .NET
Advanced Charting component for business, presentation, financial, statistical and general use.

coming soon

NOV Grid for .NET
Advanced hierarchical grid component for displaying and editing relational data.

coming soon

NOV Scheduler for .NET
Advanced scheduling component for displaying and managing appointments and tasks.

* All NOV components and frameworks are distributes as portable class libraries and work from a single code base inside WinForms, WPF, Silverlight, Xamarin.Mac, MonoMac and soot other environments.

Learn more at **www.nevron.com** today

www.nevron.com | email@nevron.com | +1 888-201-6088 (Toll free, USA and Canada)

Visual Studio Partner

With that said, you can still build AnyCPU libraries and DLLs to be referenced in your UWP app. These components will be compiled to architecture-specific binaries based on the configuration of the project (.appx) consuming it.

.NET Native in the Cloud

One great feature of .NET Native is that the compiler can be hosted in the cloud. That means when improvements are made to the compiler that might have a beneficial impact on your application, the Store's cloud-hosted .NET Native compiler can recompile your application package to reap the benefits. Any time this compilation is done, it will be transparent to you as the developer, but it will ultimately mean happier consumers of your application.

However, this can have some impact on your workflow. For example, it's a good idea to make sure you always have the latest tools installed so you can be testing your .NET Native compilation against the most recent local version of the compiler. In addition, when you build your Store package in Visual Studio, two packages are created—one .appxupload and one "test" .appx for sideloading. The .appxupload contains the MSIL binaries, as well as an explicit reference to the version of the .NET Native toolchain your app consumes (referenced in the AppxManifest.xml as "ilc.exe"). This package then goes to the Store and is compiled using the exact same version of the .NET Native toolchain. Because the compiler is cloud-hosted, it can iterate to fix bugs without you having to recompile your app locally.

With .NET Native, you have to be careful about which package you upload to the Store. Because the Store does the native compilation for you, you can't upload the native binaries generated by the local .NET Native compiler. The Visual Studio workflow will guide you through this process so you select the right package. For full guidance on creating a Store package, take a look at the MSDN Library article, "Packaging Universal Windows Apps for Windows 10," at bit.ly/1OQTTG0. This will guide you through the package creation process to make sure you generate and choose the right package to upload to the store.

Debugging with .NET Native

If you find issues in your application you suspect are caused by .NET Native, there's a technique you can use to help debug the issue. Release configurations fully optimize code (for example, code inlining is applied in many places) by default, which loses some debugging artifacts. As a result, trying to debug a Release configuration app can be difficult; you might experience unpredictable stepping and breakpoint behavior, as well as the inability to inspect variables due to memory optimization. Because the default behavior of Release configurations is to use the .NET Native compiler *with* code optimization, it's difficult to debug any issues that might be a result of the .NET Native compilation process.

A good way to get around this is to create a custom-build configuration for your project that utilizes the .NET Native compiler but doesn't fully optimize code. To create a custom-build configuration, open the Configuration Manager from the build configuration dropdown, as shown in **Figure 2**.

In the Active solution configuration dropdown, choose <New…> to create a new configuration, as shown in **Figure 3**.

Give the new configuration a name that will be useful to you later. I like to use "Debug .NET Native." Copy settings from the "Release" build configuration and then click OK.

Close the Configuration Manager and open the project's property page by right-clicking on the project in Solution Explorer, and clicking Properties. Navigate to the Build tab and make sure that Compile with .NET Native tool chain is checked, and Optimize code is unchecked, as shown in **Figure 4**.

You now have a build configuration you can use for debugging .NET Native-specific issues.

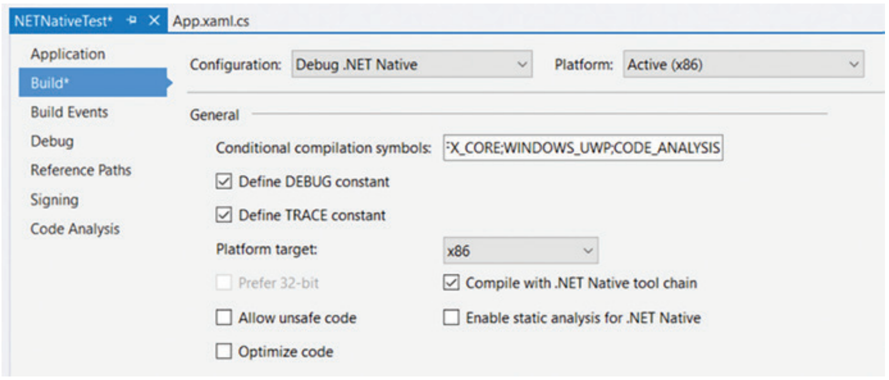
For more information about debugging with .NET Native, refer to the MSDN Library article, "Debugging .NET Native Windows Universal Apps," at bit.ly/1Ixd07v.

.NET Native Analyzer

Of course it's good to know how to debug issues, but wouldn't it be better if you could avoid them from the get-go? The Microsoft.NET-Native.Analyzer (bit.ly/1LugGnO) can be installed in your application via NuGet. From the Package Manager Console, you can install the package via the following command: Install-Package Microsoft.NET-Native.Analyzer. At development time, this analyzer will give you warnings if your code isn't compatible with the .NET Native compiler. There's a small section of the .NET surface that's not compatible, but for the majority of apps this will never be a problem.

Closing Thoughts

As you can see, it's an exciting time to be a .NET Windows developer. With the UWP, .NET Native and changes to NuGet, it has never been easier to create apps across so many different devices that your customers will love. For the first time, you can take advantage of the latest advances in any .NET class and still expect your application to run on all Windows 10 devices. ■



Configuration: Debug .NET Native
Platform: Active (x86)

Conditional compilation symbols: ;X_CORE;WINDOWS_UWP;CODE_ANALYSIS
☑ Define DEBUG constant
☑ Define TRACE constant
Platform target: x86
☐ Prefer 32-bit
☑ Compile with .NET Native tool chain
☐ Allow unsafe code
☐ Enable static analysis for .NET Native
☐ Optimize code

Figure 4 **Creating a Build Configuration for Debugging .NET Native**

Daniel Jacobson *is a program manager for Visual Studio, working on tools for Windows platform developers. Reach him at dajaco@microsoft.com.*

Thanks to the following technical experts for reviewing this article: *Kino Aguilar, Adam Denning, Yishai Galatzer, Jenny Hayes, Jeremy Meng, Harikrishna Menon, Jessica Prince, Unni Ravindranathan, Navit Saxena, Michael Strehovsky, Debbie Thorn, Matthew Whilde, Lucian Wischik*

TEXT CONTROL

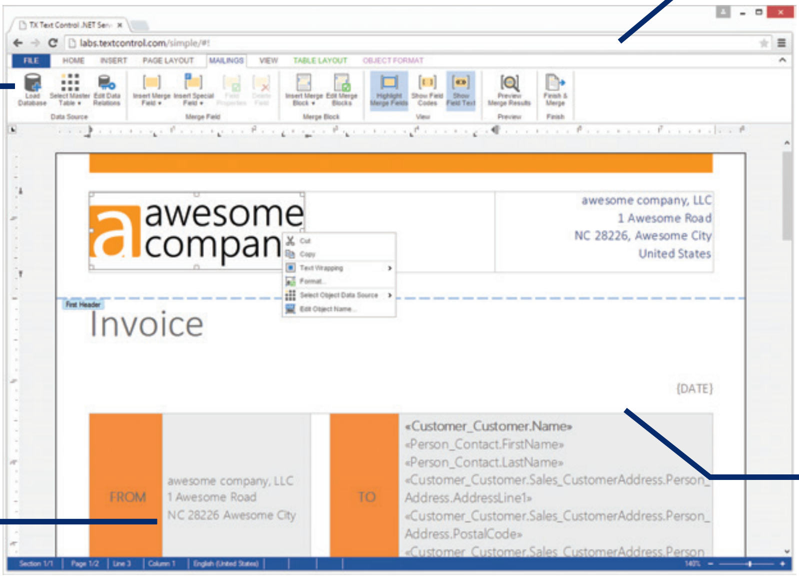
Reporting!

Combine powerful reporting with easy-to-use word processing

The **Text Control Reporting Framework** combines powerful reporting features with an easy-to-use, MS Word compatible word processor. Users create documents and templates using ordinary MS Word skills. TX Text Control is completely independent from MS Word or any other third-party application and can be completely integrated into your business application.

ASP.NET ▪ Windows Forms ▪ WPF

Database support for ADO.NET, ODBC, DataSet, DataTable and all IEnumerable business objects



awesome company

awesome company, LLC
1 Awesome Road
NC 28226, Awesome City
United States

Invoice

{DATE}

FROM
awesome company, LLC
1 Awesome Road
NC 28226 Awesome City

TO
«Customer_Customer.Name»
«Person_Contact.FirstName»
«Person_Contact.LastName»
«Customer_Customer.Sales_CustomerAddress.Person_Address.AddressLine1»
«Customer_Customer.Sales_CustomerAddress.Person_Address.PostalCode»
«Customer_Customer.Sales_CustomerAddress.Person_

Cross-browser, cross-platform document and template editing

Create Adobe PDF and PDF/A documents

MS Word compatible templates and MS Word inspired UI

Live demos and 30-day trial version download at:

http://reporting.textcontrol.com

Visual Studio Partner

HTML

TEXT CONTROL

Windows Composition Turns 10

Kenny Kerr

The Windows composition engine, otherwise known as the Desktop Window Manager (DWM), gets a new API for Windows 10. DirectComposition was the primary interface for composition, but, as a classic COM API, it was largely inaccessible to the average app developer. The new Windows composition API is built on the Windows Runtime (WinRT) and provides the foundation for high-performance rendering by blending the world of immediate-mode graphics offered by Direct2D and Direct3D with a retained visual tree that now sports much-improved animation and effects capabilities.

I first wrote about the DWM back in 2006 when Windows Vista was in beta (goo.gl/19jCyR). It allowed you to control the extent of the blur effect for a given window and to create custom chrome that blended nicely with the desktop. **Figure 1** illustrates the height of this achievement in Windows 7. It was possible to produce hardware-accelerated rendering with Direct3D and Direct2D to create stunning visuals for your app (goo.gl/IufcN1). You could even blend the old world of GDI and USER controls with the DWM (goo.gl/9ITISE). Still, any keen observer could tell that the DWM had more to offer—a lot more. The Windows Flip 3D feature in Windows 7 was convincing proof.

Windows 8 introduced a new API for the DWM called DirectComposition, its name giving tribute to the DirectX family of classic COM APIs that inspired its design. DirectComposition began to give developers a clearer picture of what the DWM was capable of doing. It also offered improved terminology. The DWM is really the Windows composition engine, and it was able to produce the dazzling effects in Windows Vista and Windows 7 because it fundamentally changed the way desktop windows were rendered. By default, the composition engine created a redirection surface for each top-level window. I described this in detail in my June 2014 column (goo.gl/oMlVa4). These redirection surfaces formed part of a visual tree, and DirectComposition allowed apps to make use of this same technology to provide a lightweight retained-mode API for high-performance graphics. DirectComposition offered a visual tree and surface management that allowed the app to offload the production of effects and animations to the composition engine. I described these capabilities in my August (goo.gl/CNwnWR) and September 2014 columns (goo.gl/y7ZMLL). I even produced a course on high-performance rendering with DirectComposition for Pluralsight (goo.gl/fgg0XN).

This article is based on prerelease software that is subject to change.

This article discusses:

- The unveiling of the Desktop Window Manager
- The evolution of visuals and surfaces
- Moving from DirectComposition to Windows Composition
- Introducing an exemplary Windows Runtime API
- Using Direct2D to render visuals

Technologies discussed:

Windows Composition, Windows Runtime, Desktop Window Manager, DirectComposition, Direct2D

Windows 8 debuted with DirectComposition, along with impressive improvements to the rest of the DirectX family of APIs, but it also ushered in a new era for the Windows API that would forever change the way that developers look at the OS. The introduction of the Windows Runtime overshadowed everything else. Microsoft promised a fresh new way to build apps and access OS services that spelled the eventual retirement of the so-called Win32 API that had long been the dominant way to build apps and interact with the OS. Windows 8 got off to a rocky start, but Windows 8.1 fixed a lot of problems and Windows 10 now provides a far more comprehensive API that will satisfy many more developers interested in building first-class apps, nay, serious applications, for Windows.

Windows 10 shipped in July 2015 with a preview of the new composition API that wasn't yet ready for production. It was still subject to change and therefore couldn't be used in Universal Windows apps submitted to the Windows Store. That's just as well because the composition API that's now available for production has changed substantially, and for the better. This Windows 10 update is also the first time that the same composition API is available on all form factors, giving further credence to the universal part of the Universal Windows Platform. Composition works the same regardless of whether you're targeting your multi-display desktop powerhouse or the small smartphone in your pocket.

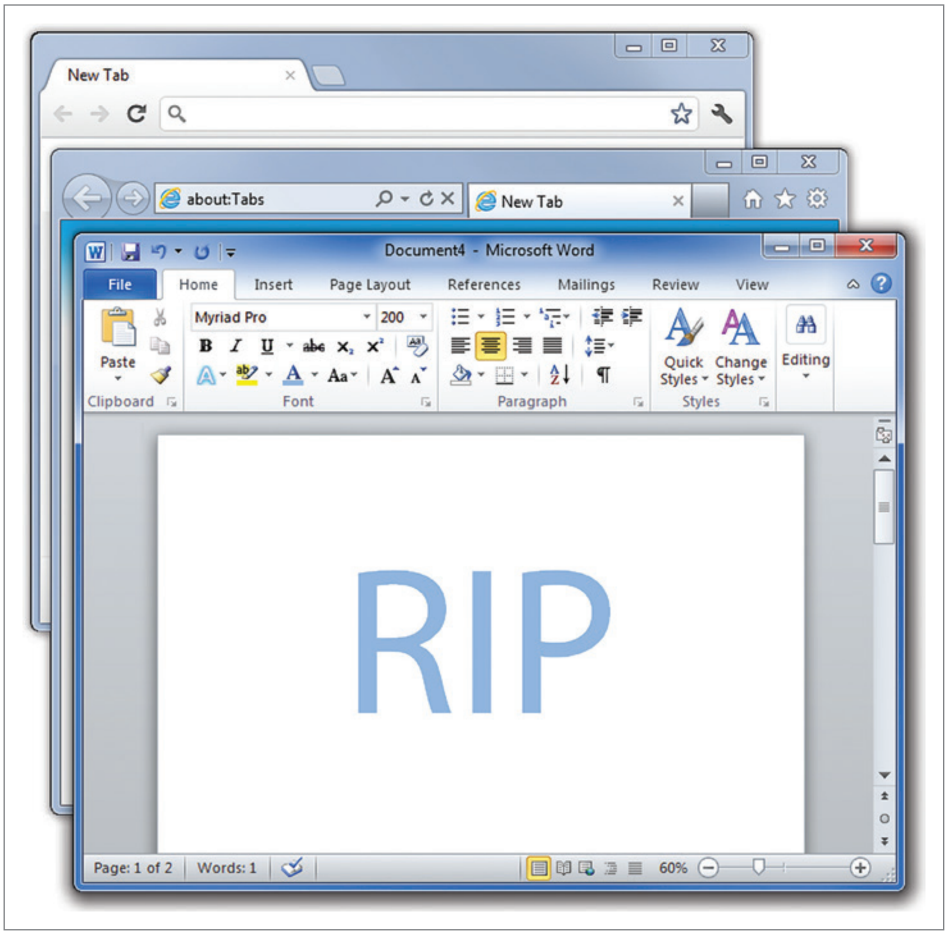


Figure 1 **Windows Aero**

Naturally, the thing everyone likes about the Windows Runtime is that it finally delivers on the promise of a common language runtime for Windows. If you prefer to code in C#, you can use the Windows Runtime directly via the support built into the Microsoft .NET Framework. If, like me, you prefer to use C++, you can use the Windows Runtime with no intermediate or costly abstractions. The Windows Runtime is built on COM rather than .NET and as such is ideally suited for C++ consumption. I'll use Modern C++ for the Windows Runtime (moderncpp.com), the standard C++ language projection, but you can follow along in your favorite language as the API is the same, regardless. I'll even offer up some examples in C# to illustrate how seamlessly the Windows Runtime can support different languages.

The Windows composition API distances itself from its DirectX roots. While DirectComposition provided a device object, modeled after the Direct3D and Direct2D devices, the new Windows composition API starts with a compositor. It does, however, serve the same purpose, acting as the factory for composition resources. Beyond that, Windows composition is very similar to DirectComposition. There's a composition target that represents the relationship between a window and its visual tree. The differences become more apparent as you look more closely at visuals. A DirectComposition visual had a content property that provided a bitmap of some kind. The bitmap was one of three things: a composition surface, a DXGI swap chain or the redirection surface of another window. A typical DirectComposition application consisted of visuals and surfaces, with surfaces acting as the content or bitmaps for the different visuals. As depicted in **Figure 2**, a composition visual tree is a slightly different beast. The new visual object has no content property and is instead rendered with a composition brush. This turns out to be a more flexible abstraction. While the brush can just render a bitmap as before, simple solid color brushes can be created very efficiently and more elaborate brushes can be defined, at least conceptually, in a manner not unlike how Direct2D provides effects that can be treated as images. The right abstraction makes all the difference.

Let's walk through some practical examples to illustrate how this all works and give you a glimpse of what's possible. Again, you can pick your favorite WinRT language projection. You can create a compositor with modern C++, as follows:

```
using namespace Windows::UI::Composition;

Compositor compositor;
```

Likewise, you can do the same with C#:

```
using Windows.UI.Composition;

Compositor compositor = new Compositor();
```

You can even use the more flamboyant syntax offered by C++/CX:

```
using namespace Windows::UI::Composition;

Compositor ^ compositor = ref new Compositor();
```

These are all equivalent from an API perspective and merely reflect the differences in language projection. There are basically two ways in which you can write a Universal Windows app today. Perhaps the most common approach is to use the OS's Windows.UI.Xaml namespace. If XAML isn't that important to your app, you can also use the underlying application model directly with no dependency on XAML. I described the WinRT application model in my August 2013 column (goo.gl/GI3OKP). Using this approach, you simply need a minimal implementation of the IFrameworkView and IFrameworkViewSource interfaces and you're good to go. **Figure 3** provides a basic outline in C# that you can use to get started. Windows composition also offers deep integration with XAML, but let's start with a simple XAML-free application as it provides a simpler playground in which to learn about composition. I'll get back to XAML a little later on in this article.

It is within the app's SetWindow method (see **Figure 3**) that the compositor should be constructed. In fact, this is the earliest point in the app's lifecycle that this can occur because the compositor depends on the window's dispatcher and this is the point at which both window and dispatcher are finally in existence. The relationship between the compositor and the app view can then be established by creating a composition target:

```
CompositionTarget m_target = nullptr;
// ...
m_target = compositor.CreateTargetForCurrentView();
```

It's vital that the app keep the composition target alive, so be sure to make it a member variable of your IFrameworkView implementation. As I mentioned before, the composition target represents the relationship between the window or view and its visual tree. All you can do with a composition target is set the root visual. Typically, this will be a container visual:

```
ContainerVisual root = compositor.CreateContainerVisual();
m_target.Root(root);
```

Here I'm using C++, which lacks language support for properties, so the Root property is projected as accessor methods. C# is very similar with the addition of property syntax:

```
ContainerVisual root = compositor.CreateContainerVisual();
m_target.Root = root;
```

DirectComposition provided only one kind of visual, which supported different kinds of surfaces for representing bitmap content. Windows composition offers small class hierarchies that represent different kinds of visuals, brushes, and animations, yet there's only one kind of surface and it can only be created using C++ because it's part of the Windows composition interop API intended for use by frameworks like XAML and more sophisticated app developers.

The visual class hierarchy is shown in **Figure 4**. A CompositionObject is a resource backed by the compositor. All composition objects may potentially have their properties animated. A Visual provides a slew of properties for controlling many aspects of the visual's relative position, appearance, clipping and rendering options. It includes a transform matrix property, as well as shortcuts for scale and rotation. This is a powerful base class. By contrast, ContainerVisual is a relatively simple class that simply adds a Children property. While you can create container visuals directly, a SpriteVisual adds the ability to associate a brush so that the visual can actually render pixels of its own.

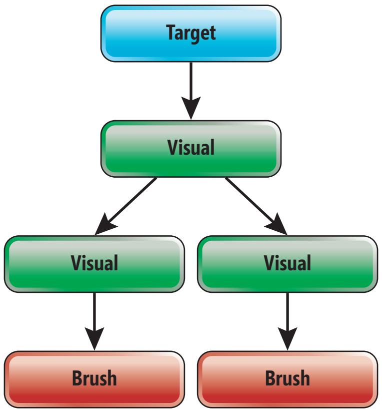


Figure 2 **The Windows Composition Visual Tree**

Given a root container visual, I can create any number of child visuals:

```
VisualCollection children = root.Children();
```

These can also be container visuals, but more likely they'll be sprite visuals. I could add three visuals as children of the root visual using a for loop in C++:

```
using namespace Windows::Foundation::Numerics;

for (unsigned i = 0; i != 3; ++i)
{
  SpriteVisual visual = compositor.CreateSpriteVisual();
  visual.Size(Vector2{ 300.0f, 200.0f });
  visual.Offset(Vector3{ 50 + 20.0f * i, 50 + 20.0f * i });

  children.InsertAtTop(visual);
}
```

You can easily imagine the app window in **Figure 5**, and yet this code won't result in anything being rendered because there's no brush associated with these visuals. The brush class hierarchy is shown in **Figure 6**. A CompositionBrush is simply a base class for brushes and provides no functionality of its own. A CompositionColorBrush is the simplest kind, offering only a color property

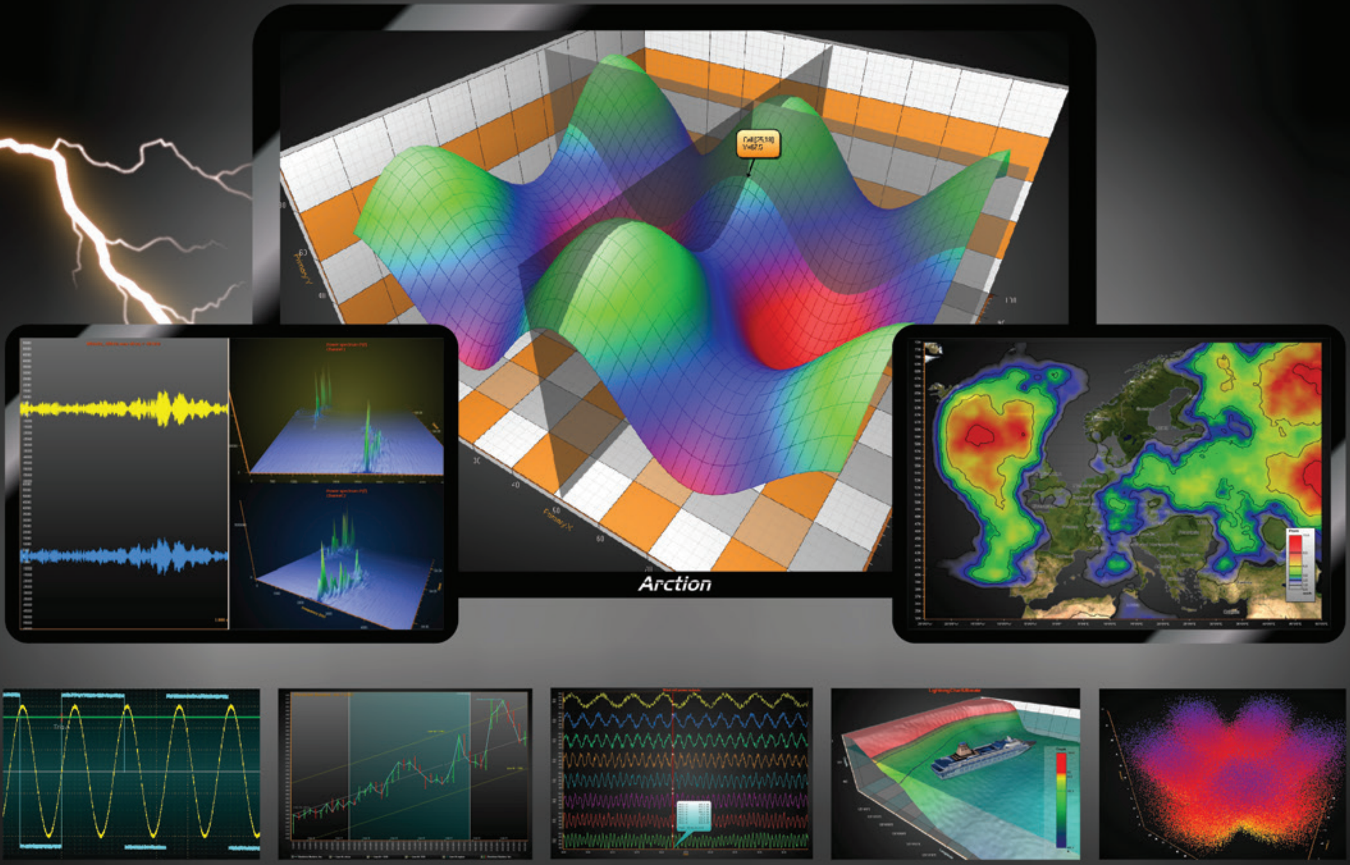
Figure 3 **Windows Runtime Application Model in C#**

```
using Windows.ApplicationModel.Core;
using Windows.UI.Core;

class View : IFrameworkView, IFrameworkViewSource
{
  static void Main()
  {
    CoreApplication.Run(new View());
  }

  public IFrameworkView CreateView()
  {
     return this;
  }

  public void SetWindow(CoreWindow window)
  {
    // Prepare composition resources here...
  }

  public void Run()
  {
    CoreWindow window = CoreWindow.GetForCurrentThread();
    window.Activate();
    window.Dispatcher.ProcessEvents(CoreProcessEventsOption.ProcessUntilQuit);
  }

  public void Initialize(CoreApplicationView applicationView) { }
  public void Load(string entryPoint) { }
  public void Uninitialize() { }
}
```

WINDOWS 10
COMPATIBLE

The FASTEST rendering Data Visualization components for WPF and WinForms...

LightningChart



HEAVY-DUTY DATA VISUALIZATION TOOLS FOR SCIENCE, ENGINEERING AND TRADING

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2015
- Hundreds of examples

WPF charts performance comparison

Opening large dataset	LightningChart is up to 977,000 % faster
Real-time monitoring	LightningChart is up to 2,700,000 % faster

WinForms charts performance comparison

Opening large dataset	LightningChart is up to 37,000 % faster
Real-time monitoring	LightningChart is up to 2,300,000 % faster

Results compared to average of other chart controls. See details at www.LightningChart.com/benchmark. LightningChart results apply for Ultimate edition.

Download a free 30-day evaluation from
www.LightningChart.com

Arction

for rendering solid color visuals. This might not sound very exciting, but don't forget that you can connect animations to that color property. The CompositionEffectBrush and CompositionSurfaceBrush classes are related, but are more complex brushes because they're backed by other resources. A CompositionSurfaceBrush will render a composition surface for any attached visuals. It has a variety of properties that control bitmap drawing, such as interpolation, alignment, and stretch, not to mention the surface itself. A CompositionEffectBrush takes a number of surface brushes to produce various effects.

Creating and applying a color brush is straightforward. Here's an example in C#:

```
using Windows.UI;

CompositionColorBrush brush = compositor.CreateColorBrush();
brush.Color = Color.FromArgb(0xDC, 0x5B, 0x9B, 0xD5);
visual.Brush = brush;
```

Creating an animation object and then applying that animation to a particular composition object is surprisingly easy.

The Color structure comes courtesy of the Windows.UI namespace and sports alpha, red, green, and blue as 8-bit color values, a departure from the DirectComposition and Direct2D preference for floating-point color values. A nice feature of this approach to visuals and brushes is that the color property can be changed at any time, and any visuals referring to the same brush will be updated automatically. Indeed, as I've hinted at before, the color property

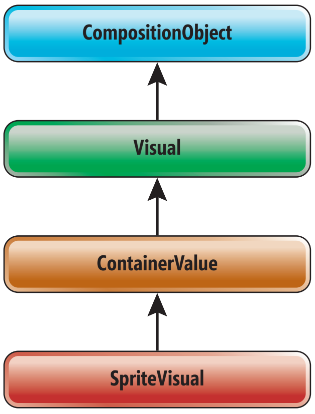


Figure 4 **Composition Visuals**

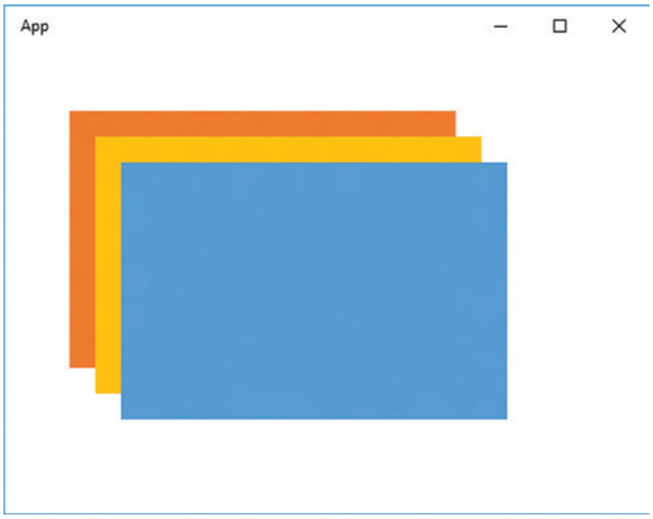


Figure 5 **Child Visuals in a Window**

can even be animated. So how does that work? This brings us to the animation classes.

The animation class hierarchy is shown in **Figure 7**. The CompositionAnimation base class provides the ability to store named values for use with expressions. I'll talk more about expressions in a moment. A KeyFrameAnimation provides typical keyframe-based animation properties like duration, iteration and stop behavior. The various keyframe animation classes offer type-specific methods for inserting keyframes, as well as type-specific animation properties. For example, ColorKeyFrameAnimation lets you insert keyframes with color values and a property to control the color space for interpolating between keyframes.

Creating an animation object and then applying that animation to a particular composition object is surprisingly easy. Suppose I want to animate the opacity of a visual. I could set the visual's opacity to 50 percent directly with a scalar value in C++, as follows:

```
visual.Opacity(0.5f);
```

Alternatively, I can create a scalar animation object with keyframes to produce an animation variable from 0.0 through 1.0, representing 0 percent through 100 percent opacity:

```
ScalarKeyFrameAnimation animation =
  compositor.CreateScalarKeyFrameAnimation();
animation.InsertKeyFrame(0.0f, 0.0f); // Optional
animation.InsertKeyFrame(1.0f, 1.0f);
```

The first parameter of InsertKeyFrame is the relative offset from the start of the animation (0.0) to the end of the animation (1.0). The second parameter is the value of the animation variable at that point in the animation timeline. So this animation will smoothly transition the value from 0.0 to 1.0 over the duration of the animation. I can then set the overall duration of this animation, as follows:

```
using namespace Windows::Foundation;

animation.Duration(TimeSpan::FromSeconds(1));
```

With the animation ready to go, I simply need to connect it to the composition object and property of my choice:

```
visual.StartAnimation(L"Opacity", animation);
```

The StartAnimation method is actually inherited from the CompositionObject base class, meaning that you can animate the properties of a variety of different classes. This is another departure from DirectComposition, where each animatable property provided overloads for scalar values, as well as animation objects. Windows composition offers a much richer property system that opens the door to some very interesting capabilities. In particular, it supports the ability to write textual expressions that cut down on the amount of code that needs to be written for more interesting animations and effects. These expressions are parsed at run time, compiled and then efficiently executed by the Windows composition engine.

Imagine you need to rotate a visual along the Y axis and give it the appearance of depth. The visual's RotationAngle property, measured in radians, isn't enough because that won't produce a transformation that includes perspective. As the visual rotates, the edge closest to the human eye should appear larger, while the opposite edge should appear smaller. **Figure 8** shows a number of rotating visuals that illustrate this behavior.

How would you accomplish such an animated effect? Well, let's start with a scalar keyframe animation for the rotation angle:

```
ScalarKeyFrameAnimation animation = compositor.CreateScalarKeyFrameAnimation();
animation.InsertKeyFrame(1.0f, 2.0f * Math::Pi,
  compositor.CreateLinearEasingFunction());
animation.Duration(TimeSpan::FromSeconds(2));
animation.IterationBehavior(AnimationIterationBehavior::Forever);
```

The linear easing function overrides the default acceleration/deceleration function to produce a continuous rotating motion. I then need to define a custom object with a property I can refer to from within an expression. The compositor provides a property set for just this purpose:

```
CompositionPropertySet rotation = compositor.CreatePropertySet();
rotation.InsertScalar(L"Angle", 0.0f);
```

A property set is also a composition object, so I can use the StartAnimation method to animate my custom property just as easily as any built-in property:

```
rotation.StartAnimation(L"Angle", animation);
```

WinRT classes might implement additional COM interfaces not directly visible if all you have is a component's Windows metadata.

I now have an object whose Angle property is in motion. Now I need to define a transform matrix to produce the desired effect, while delegating to this animated property for the rotation angle itself. Enter expressions:

```
ExpressionAnimation expression =
  compositor.CreateExpressionAnimation(
    L"pre * Matrix4x4.CreateFromAxisAngle(axis, rotation.Angle) * post");
```

An expression animation is not a keyframe animation object, so there's no relative keyframe offsets at which animation variables might change (based on some interpolation function). Instead, expressions simply refer to parameters that may themselves be animated in the more traditional sense. Still, it's up to me to define what "pre," "axis," "rotation," and "post" are. Let's start with the axis parameter:

```
expression.SetVector3Parameter(L"axis", Vector3{ 0.0f, 1.0f, 0.0f });
```

The CreateFromAxisAngle method inside the expression expects an axis to rotate around and this defines the axis around the Y axis.

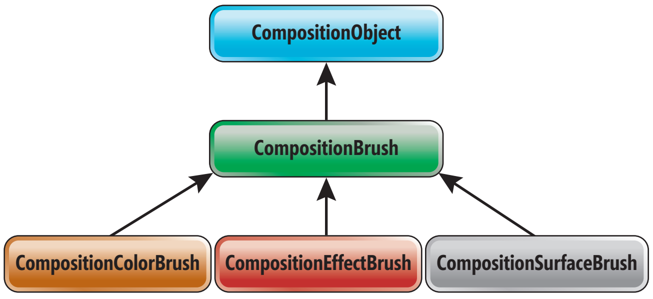


Figure 6 **Composition Brushes**

It also expects an angle of rotation and for that we can defer to the rotation property set with its animated "Angle" property:

```
expression.SetReferenceParameter(L"rotation", rotation);
```

To ensure that the rotation occurs through the center of the visual rather than the left edge, I need to pre-multiply the rotation matrix created by CreateFromAxisAngle with a translation that logically shifts the axis to the point of rotation:

```
expression.SetMatrix4x4Parameter(
  L"pre", Matrix4x4::Translation(-width / 2.0f, -height / 2.0f, 0.0f));
```

Remember that matrix multiplication is not commutative, so the pre and post matrixes really are just that. Finally, after the rotation matrix, I can add some perspective and then restore the visual to its original location:

```
expression.SetMatrix4x4Parameter(
  L"post", Matrix4x4::PerspectiveProjection(width * 2.0f) *
    Matrix4x4::Translation(width / 2.0f, height / 2.0f, 0.0f));
```

This satisfies all the parameters referred to by the expression and I can now simply use the expression animation to animate the visual using its TransformMatrix property:

```
visual.StartAnimation(L"TransformMatrix", expression);
```

So I've explored various ways to create, fill and animate visuals, but what if I need to render visuals directly? DirectComposition offered both preallocated surfaces and sparsely allocated bitmaps called virtual surfaces that were allocated on demand and also resizable. Windows composition seemingly provides no ability to create surfaces. There's a CompositionDrawingSurface class, but no way to create it without outside help. The answer comes from the Windows composition interop API. WinRT classes might implement additional COM interfaces not directly visible if all you have is a component's Windows metadata. Given the knowledge of these cloaked interfaces, you can easily query for them in C++. Naturally, this is going to be a bit more work as you're stepping outside of the neat abstractions provided to mainstream developers by the Windows composition API. The first thing I need to do is create a rendering device and I'll use Direct3D 11 because Windows composition doesn't yet support Direct3D 12:

```
ComPtr<ID3D11Device> direct3dDevice;
```

I'll then prepare the device creation flags:

```
unsigned flags = D3D11_CREATE_DEVICE_BGRA_SUPPORT |
                 D3D11_CREATE_DEVICE_SINGLETHREADED;

#ifdef _DEBUG
flags |= D3D11_CREATE_DEVICE_DEBUG;
#endif
```

The BGRA support allows me to use the more approachable Direct2D API for rendering with this device, and then the D3D11CreateDevice function creates the hardware device itself:

```
check(D3D11CreateDevice(nullptr, // Adapter
                        D3D_DRIVER_TYPE_HARDWARE,
                        nullptr, // Module
                        flags,
                        nullptr, 0, // Highest available feature level
                        D3D11_SDK_VERSION,
                        set(direct3dDevice),
                        nullptr, // Actual feature level
                        nullptr)); // Device context
```

I then need to query for the device's DXGI interface, because that's what I'll need to create a Direct2D device:

```
ComPtr<IDXGIDevice3> dxgiDevice = direct3dDevice.As<IDXGIDevice3>();
```

Now it's time to create the Direct2D device itself:

```
ComPtr<ID2D1Device> direct2dDevice;
```

Here, again, I'll enable the debug layer for added diagnostics:

```
D2D1_CREATION_PROPERTIES properties = {};

#ifdef _DEBUG
properties.debugLevel = D2D1_DEBUG_LEVEL_INFORMATION;
#endif
```

I could first create a Direct2D factory to create the device. That would be useful if I needed to create any device-independent resources. Here I'll just use the shortcut provided by the D2D1-CreateDevice function:

```
check(D2D1CreateDevice(get(dxgiDevice), properties, set(direct2dDevice)));
```

Because CompositionGraphicsDevice is a WinRT type, I can again use modern C++.

The rendering device is ready. I have a Direct2D device I can use to render whatever I can imagine. Now I need to tell the Windows composition engine about this rendering device. This is where those cloaked interfaces come in. Given the compositor that I've used throughout, I can query for the ICompositorInterop interface:

```
namespace abi = ABI::Windows::UI::Composition;

ComPtr<abi::ICompositorInterop> compositorInterop;
check(compositor->QueryInterface(set(compositorInterop)));
```

ICompositorInterop provides methods for creating a composition surface from a DXGI surface, which would certainly be handy if you want to include an existing swap chain in a composition visual tree, but it provides something else that's far more interesting. Its CreateGraphicsDevice method will create a CompositionGraphicsDevice object given a rendering device. The CompositionGraphicsDevice class is a normal class in the Windows composition API, rather than a cloaked interface, but it doesn't provide a constructor so you need to use C++ and the ICompositorInterop interface to create it:

```
CompositionGraphicsDevice device = nullptr;
check(compositorInterop->CreateGraphicsDevice(get(direct2dDevice), set(device)));
```

Because CompositionGraphicsDevice is a WinRT type, I can again use modern C++, rather than pointers and manual error handling. And it's the CompositionGraphicsDevice that finally allows me to create a composition surface:

```
using namespace Windows::Graphics::DirectX;

CompositionDrawingSurface surface =
  compositionDevice.CreateDrawingSurface(Size{ 100, 100 },
    DirectXPixelFormat::B8G8R8A8UIntNormalized,
    CompositionAlphaMode::Premultiplied);
```

Here I'm creating a composition surface 100 x 100 pixels in size. Note that this represents physical pixels rather than the logical and DPI-aware coordinates assumed and provided by the rest of Windows composition. The surface also provides 32-bit alpha-blended rendering supported by Direct2D. Of course, Direct3D and Direct2D are not yet offered through the Windows Runtime, so it's back to cloaked interfaces to actually draw to this surface:

```
ComPtr<abi::ICompositionDrawingSurfaceInterop> surfaceInterop;
check(surface->QueryInterface(set(surfaceInterop)));
```

Much like DirectComposition before it, Windows composition provides BeginDraw and EndDraw methods on the ICompositionDrawingSurfaceInterop interface that subsume and take the place of the typical calls to the Direct2D method calls that go by the same names:

```
ComPtr<ID2D1DeviceContext> dc;
POINT offset = {};

check(surfaceInterop->BeginDraw(nullptr, // Update rect
                                __uuidof(dc),
                                reinterpret_cast<void **>(set(dc)),
                                &offset));
```

Windows composition takes the original rendering device provided at the time the composition device was created and uses it to create a device context or render target. I can optionally provide a clipping rectangle in physical pixels, but here I'm just opting for unrestricted access to the rendering surface. BeginDraw also returns an offset, again in physical pixels, indicating the origin of the intended drawing surface. This will not necessarily be the top-left corner of the render target, and care must be taken to adjust or transform any drawing commands to properly accommodate this offset. Again, don't call BeginDraw on the render target as Windows composition has already done that for you. This render target is logically owned by the composition API and care must be taken not to hold on to it following the call to EndDraw. The render target is now ready to go, but isn't aware of the logical or effective DPI for the view. I can use the Windows::Graphics::Display namespace to get the logical DPI for the current view and set the DPI that will be used by Direct2D for rendering:

```
using namespace Windows::Graphics::Display;

DisplayInformation display = DisplayInformation::GetForCurrentView();
float const dpi = display.LogicalDpi();
dc->SetDpi(dpi, dpi);
```

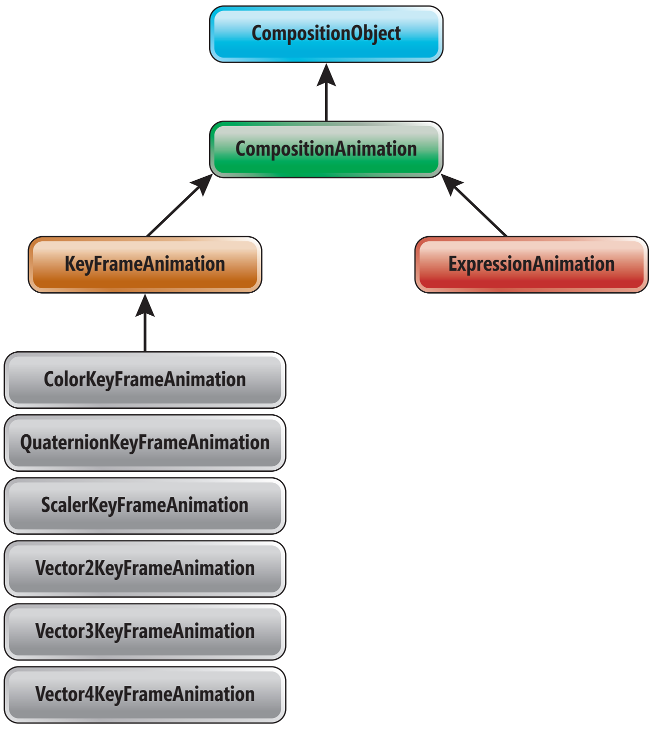


Figure 7 **Composition Animations**

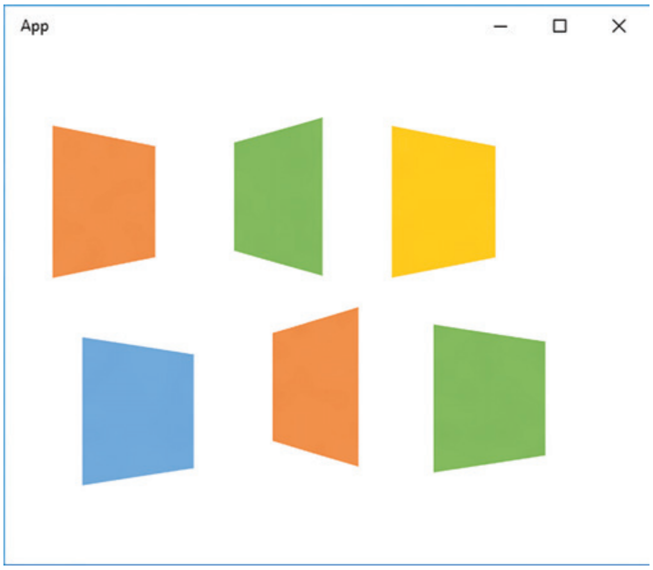


Figure 8 **Rotating Visuals**

The final step before rendering can commence is to handle the composition offset somehow. One simple solution is to use the offset to produce a transform matrix. Just remember that Direct2D trades in logical pixels, so I need to use not only the offset, but also the newly established DPI value:

```
dc->SetTransform(D2D1::Matrix3x2F::Translation(offset.x * 96.0f / dpi,
                                              offset.y * 96.0f / dpi));
```

At this point, you can draw to your heart's content before calling the EndDraw method on the surface's interop interface to ensure that any batched Direct2D drawing commands are processed and changes to the surface are reflected in the composition visual tree:

```
check(surfaceInterop->EndDraw());
```

Of course, I haven't yet associated the surface with a visual and, as I've mentioned, visuals no longer provide a content property and must be rendered using a brush. Fortunately, the compositor will create a brush to represent a preexisting surface:

```
CompositionSurfaceBrush brush = compositor.CreateSurfaceBrush(surface);
```

I can then create a normal sprite brush and use this brush to bring the visual to light:

```
SpriteVisual visual = compositor.CreateSpriteVisual();
visual.Brush(brush);
visual.Size(Vector2{ ... });
```

If that's not enough interoperability for you, you can even take a XAML element and retrieve the underlying composition visual. Here's an example in C#:

```
using Windows.UI.Xaml.Hosting;

Visual visual = ElementCompositionPreview.GetElementVisual(button);
```

Despite its seemingly temporary status, ElementCompositionPreview is in fact ready for production and may be used by apps submitted to the Windows Store. Given any UI element, the static GetElementVisual method will return the visual from the underlying composition visual tree. Notice that it returns a Visual rather than a ContainerVisual or SpriteVisual, so you can't directly work with visual children or apply a brush, but you can adjust the many visual properties offered by Windows composition. The ElementCompositionPreview helper class provides some additional static methods for adding child visuals in a controlled way. You can change the visual's offset and things like UI hit testing will continue to work at the XAML level. You can even apply an animation directly with Windows composition without breaking the XAML infrastructure built upon it. Let's create a simple scalar animation to rotate the button. I need to retrieve the compositor from the visual, then creating an animation object works as before:

```
Compositor compositor = visual.Compositor;
ScalarKeyFrameAnimation animation = compositor.CreateScalarKeyFrameAnimation();
```

Let's build a simple animation to slowly rotate the button forever with a linear easing function:

```
animation.InsertKeyFrame(1.0f, (float) (2 * Math.PI),
  compositor.CreateLinearEasingFunction());
```

I can then indicate that a single rotation should take 3 seconds and continue forever:

```
animation.Duration = TimeSpan.FromSeconds(3);
animation.IterationBehavior = AnimationIterationBehavior.Forever;
```

Finally, I can simply connect the animation to the visual provided by XAML, instructing the compositing engine to animate its RotationAngle property:

```
visual.StartAnimation("RotationAngle", animation);
```

Although you may be able to pull this off with XAML alone, the Windows composition engine provides far more power and flexibility given that it resides at a much lower level of abstraction and can undoubtedly provide better performance. As another example, Windows composition provides quaternion animations not currently supported by XAML.

There's so much more to talk about when it comes to the Windows composition engine. In my humble opinion, this is the most groundbreaking WinRT API to date.

There's so much more to talk about when it comes to the Windows composition engine. In my humble opinion, this is the most groundbreaking WinRT API to date. The amount of power at your disposal is staggering and, yet, unlike so many other large UI and graphics APIs, it doesn't introduce a performance tradeoff or even a prohibitive learning curve. In many ways, Windows composition is representative of all that's good and exciting about the Windows platform.

You can find the Windows Composition team on Twitter: @WinComposition. ■

Kenny Kerr *is a computer programmer based in Canada, as well as an author for Pluralsight and a Microsoft MVP. He blogs at kennykerr.ca and you can follow him on Twitter: @kennykerr.*

Thanks to the following Microsoft technical experts for reviewing this article: *Mark Aldham, James Clarke, John Serna, Jeffrey Stall and Nick Waggoner*

Keep Apps Alive with Background Tasks and Extended Execution

Shawn Henry

It used to be the life of an application was easy. When a user launched an app, it could run wild on the system: consuming resources, popping up windows and generally doing as it pleased without regard for others. Nowadays, things are tougher. In a mobile-first world, apps are restricted to a defined application lifecycle. This lifecycle specifies when an app can run, and most of the time it can't—if the app isn't the current thing the user is doing, it's not allowed to run.

In most cases, this is good—users know that an app isn't consuming power or sapping performance. For apps, the OS is enforcing what has always been good practice, that applications enter a quiesced steady state when not in active use. This is especially important because the application model in the Universal Windows Platform (UWP) needs to scale from the lowest-end devices, like phones and Internet of Things (IoT) devices, all the way to the most powerful desktops and Xboxes.

This article discusses:

- The application lifecycle
- Extended execution
- Background tasks
- Application triggers

Technologies discussed:

Windows 10, Universal Windows Platform

For complex apps, the modern application model can seem restrictive at first, but as this article will describe, there are a few ways applications can expand the box and run (completing tasks and delivering notifications), even when not in the foreground.

The Application Lifecycle

In traditional Win32 and .NET desktop development, apps are typically in one of two states: "running" or "not running," and most of the time they're running. This may seem obvious, but think of an instant messaging application like Skype or a music app like Spotify: the user launches it, does some action (sending a message or searching for music), then goes off and does something else. All the while, Skype sits in the background waiting for messages to come in, and Spotify keeps playing music. This contrasts with modern apps (such as Windows apps built on the UWP), which spend most of their time in a state other than running.

Windows apps are always in one of three states: running, suspended or not running, as shown in **Figure 1**. When a user launches a Windows app, for example by tapping on a Tile on the Start menu, the app is activated and enters the running state. As long as the user is interacting with the app, it stays in the running state. If the user navigates away from the app or minimizes it, a suspended event is fired. This is an opportunity for the app to serialize any state it might need for when it's resumed or re-activated, such as the current page of the application or partially filled-out form data. When an app is in the suspended state, its process and threads are suspended by Windows and can't execute. When the user navigates back to the app, the application threads are unfrozen and the application resumes the running state.

If an application is in the suspended state but the resources it's using (typically memory), are required for something else, such as running another app, the application is moved to the not running state.

From an app execution perspective, the difference between the suspended and not running states is whether the application is allowed to stay resident in memory. When an application is merely suspended, its execution is frozen, but all of its state information stays in memory. When an application isn't running, it's removed from memory. Because no event is fired when an app moves from suspended to not running, it's important for an application to serialize all of the state information it needs from memory, in case it's re-activated from not running.

Figure 2 shows what happens to resource usage as applications transition through the lifecycle. When an application is activated, it begins to consume memory, typically reaching a relatively stable plateau. When an application is suspended, its memory consumption typically goes down—buffers and used resources are released, and CPU consumption goes to zero (enforced by the OS). When an app moves from suspended to not running, both memory and CPU use go to zero, again enforced by the OS.

On many desktop systems with lots of RAM and large page files, it's not common—but not impossible—for an application to be removed from memory, but this transition is much more common on mobile and other resource-constrained devices. For this reason, it's important to test your UWP app on a wide variety of devices. The Windows Emulator that ships with Visual Studio can be immensely helpful for this; it allows developers to target devices with as little 512MB of memory.

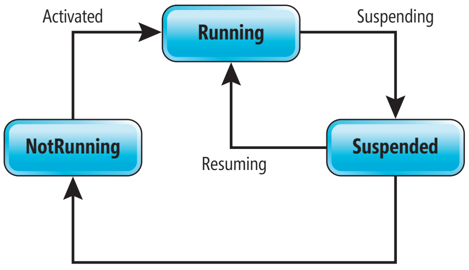


Figure 1 **The Windows Application Lifecycle**

Extended Execution

The application lifecycle I described is efficient—apps aren't consuming resources if they aren't being used—but it can result in a scenario where what an app needs can't be done. For example, a typical use case for a social or communications app is to sign in and sync a bunch of cloud data (contacts, feeds, conversation history and so forth). With the basic application lifecycle as described, in order to download contacts, the user would need to keep the app open and in the foreground the entire time! Another example might be a navigation or fitness application that needs to track a user's location. As soon as the user switched to a different app or put the device in their pocket, this would no longer work. There needs to be a mechanism to allow applications to run a bit longer.

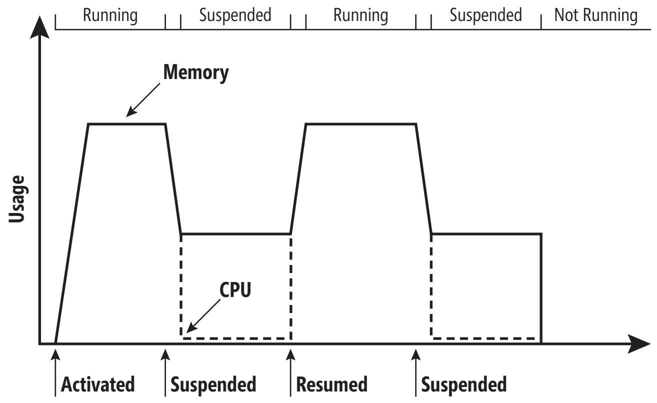


Figure 2 **The Application Lifecycle and Resource Usage**

The Universal Windows Platform introduces the concept of *extended execution* to help with these types of scenarios. There are two cases where extended execution can be used:

1. At any point during regular foreground execution, while the application is in the running state.
2. After the application has received a suspending event (the OS is about to move the app to the suspended state) in the application's suspending event handler.

The code for these two cases is the same, but the application behaves a little differently in each. In the first case, the application stays in the running state, even if an event that normally would trigger suspension occurs (for example, the user navigating away from the application). The application will never receive a suspending event while the execution extension is in effect. When the extension is disposed, the application becomes eligible for suspension again.

With the second case, if the application is transitioning to the suspended state, it will stay in a suspending state for the period of the extension. Once the extension expires, the application enters the suspended state without further notification.

Figure 3 shows how to use extended execution to extend the uspending state of an application. First, a new ExtendedExecutionSession is created and supplied with a Reason and Description. These two properties are used to allocate the correct resource set for that application (that is, the amount of memory, CPU and execution

Figure 3 **Extended Execution in the OnSuspending Handler**

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();

  using (var session = new ExtendedExecutionSession())
  {
    session.Reason = ExtendedExecutionReason.SavingData;
    session.Description = "Upload Data";

    session.Revoked += session_Revoked;

    var result = await session.RequestExtensionAsync();

    if (result == ExtendedExecutionResult.Denied)
    {
      UploadBasicData();
    }

    // Upload Data
    var completionTime = await UploadDataAsync(session);
  }
  deferral.Complete();
}
```

time it's allowed) and to expose information to the user about what applications are doing in the background. The application then hooks up a revocation event handler, which is called if Windows can no longer support the extension, for example, if another high-priority task, like a foreground application or an incoming VoIP call, needs the resources. Finally, the application requests the extension and, if successful, begins its save operation. If the extension is denied, the application performs a suspension operation as if it hadn't received the extension, like a regular suspension event.

Figure 4 shows the impact this has on the app lifecycle; compare it to **Figure 2**. When an application gets a suspension event, it begins to release or serialize resources. If the app determines it needs more time and takes an extension, it remains in the suspending state until the extension is revoked.

As mentioned earlier in case 2, execution extension doesn't have to be requested only in the suspension handler; it can be requested at any point while the application is in the running state. This is useful for when the application knows beforehand it will need to continue running in the background, as with the navigation app mentioned earlier. The code is very similar to the previous example, and is shown in **Figure 5**.

Figure 6 shows the application lifecycle for this scenario. Again, it's very similar; the difference is that when the execution extension is revoked, the application will likely quickly transition through the suspended state into the not running state. This happens because, in this case, the extension is typically revoked only due to resource pressure, a situation that can only be mitigated by releasing resources (that is, removing the app from memory).

Background Tasks

There's another way an app can run in the background, and that's as a *background task*. Background tasks are separate components in an application that implement the IBackgroundTask interface. These components can be executed without heavyweight UI frameworks, and typically execute in a separate process (although they can also be run in-proc with the primary application executable).

A background task is executed when its associated trigger is fired. A trigger is a system event that can fire and activate an app, even if the app isn't running. For example, the TimeTrigger can be generated with a specific time interval (say, every 30 minutes), in which case the application's background task would activate every 30 minutes when the trigger fires. There are many trigger types

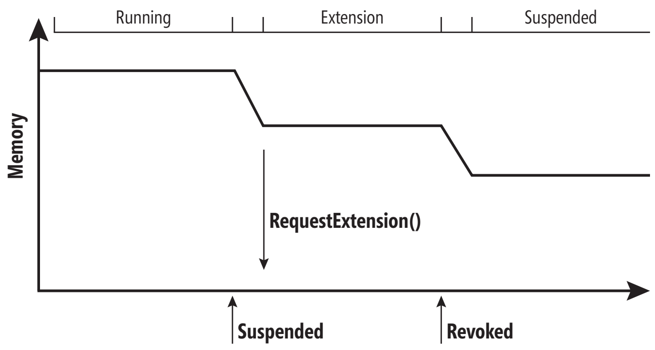


Figure 4 **Resource Usage During Extended Execution**

Figure 5 **Extended Execution During Regular Execution**

```
private async void StartTbTNavigationSession()
{
  using (var session = new ExtendedExecutionSession())
  {
    session.Reason = ExtendedExecutionReason.LocationTracking;
    session.Description = "Turn By Turn Navigation";

    session.Revoked += session_Revoked;

    var result = await session.RequestExtensionAsync();

    if (result == ExtendedExecutionResult.Denied
    {
      ShowUserWarning("Background location tracking not available");
    }

    // Do Navigation
    var completionTime = await DoNavigationSessionAsync(session);
  }
}
```

supported by Windows, including these background trigger types: TimeTrigger, PushNotificationTrigger, LocationTrigger, ContactStoreNotificationTrigger, BluetoothLEAdvertisementWatcherTrigger, UserPresent, InternetAvailable and PowerStateChange.

Using a background task is a three-step process: The component needs to be created, then declared in the application manifest and then registered at run time.

Background tasks are typically implemented in separate Windows Runtime (WinRT) component projects in the same solution as the UI project. This allows the background task to be activated in a separate process, reducing the memory overhead required by the component. A simple implementation of an IBackgroundTask is shown in **Figure 7**. IBackgroundTask is a simple interface that defines just one method, Run. This is the method that's called when the background task's trigger is fired. The method's only parameter is an IBackgroundTaskInstance object that contains context about the activation (for example, the payload of an associated push notification or the action used by a toast notification) and event handlers to handle lifecycle events such as cancellation. When the Run method completes, the background task is terminated. For this reason, just as in the suspension handler shown earlier, it's important to use the deferral object (also hanging off the IBackgroundTaskInstance) if your code is asynchronous.

In the application manifest, the background task must also be registered. This registration tells Windows the trigger type, the entry point and the executable host of the task, as follows:

```
<Extensions>
  <Extension Category="windows.backgroundTasks"
EntryPoint="BackgroundTasks.TimerTask">
    <BackgroundTasks>
      <Task Type="timer" />
    </BackgroundTasks>
  </Extension>
```

This can also be done without diving into XML by using the manifest designer included with Visual Studio.

Finally, the task must also be registered at run time, and this is shown in **Figure 8**. Here I use the BackgroundTaskBuilder object to register the task with a TimeTrigger that will fire every 30 minutes, if the Internet is available. This is the ideal type of trigger for common operations like updating a tile or periodically syncing small amounts of data.

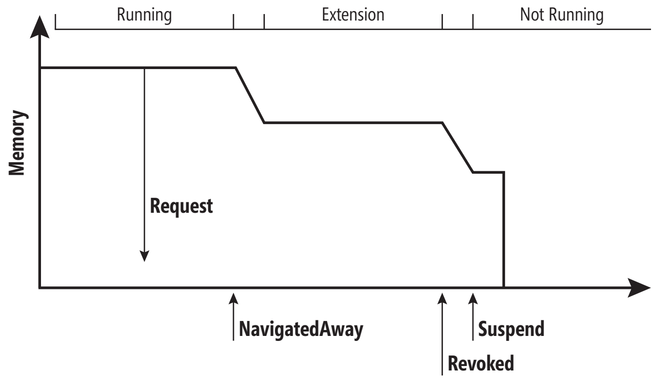


Figure 6 **Resource Usage During Extended Execution**

The biggest advantage of background tasks can also be their curse: Because background tasks run in the background (when the user might be doing something important in the foreground or the device is in standby), they are tightly restricted in the amount of memory and CPU time they can use. For example, the background task registered in **Figure 8** will run for only 30 seconds and can't consume more than 16MB of memory on devices with 512MB of memory; the memory caps scale with the amount of memory on the device. When developing and implementing background tasks, it's important to take this into consideration. Background tasks should be tested on a variety of devices, especially low-end devices, before you publish an application. There are a couple of other things to note, as well:

- If Battery Saver is available and active (typically when the battery is below a certain charge threshold), background tasks are prevented from running until the battery is recharged past the battery-saver threshold.
- On previous versions of Windows, applications needed to be "pinned" to the lock before they were allowed to execute in the background and, in some cases, there were a maximum number of background tasks that could be registered, device-wide. This is no longer the case in Windows 10, but an app must always call BackgroundExecutionManger.RequestAcessAsync to declare its intent to run in the background.

Figure 7 **A BackgroundTask Implementation**

```
public sealed class TimerTask : IBackgroundTask
{
  public void Run(IBackgroundTaskInstance taskInstance)
  {
    var deferral = taskInstance.GetDeferral();
    taskInstance.Canceled += TaskInstance_Canceled;

    await ShowToastAsync("Hello from Background Task");

    deferral.Complete();
  }

  private void TaskInstance_Canceled(IBackgroundTaskInstance sender,
    BackgroundTaskCancellationReason reason)
  {
    // Handle cancellation

    deferral.Complete();
  }
}
```

Figure 8 **Background Task Registration**

```
private void RegisterBackgroundTasks()
{
  BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
  builder.Name = "Background Test Class";
  builder.TaskEntryPoint = "BackgroundTaskLibrary.TestClass";

  IBackgroundTrigger trigger = new TimeTrigger(30, true);
  builder.SetTrigger(trigger);
  IBackgroundCondition condition =
    new SystemCondition(SystemConditionType.InternetAvailable);
  builder.AddCondition(condition);

  IBackgroundTaskRegistration task = builder.Register();

  task.Progress += new BackgroundTaskProgressEventHandler(task_Progress);
  task.Completed += new BackgroundTaskCompletedEventHandler(task_Completed);
}
```

A Better Way to Do Contact Syncing

There are many background operations that can be completed with either extended execution or with background tasks. Typically, it's better to use background tasks if possible—they're more reliable and efficient.

For example, the scenario mentioned earlier—syncing data when an app is first launched—can also be done, and done more efficiently, with a background task, in this case using a trigger that's new to Windows 10: the application trigger.

ApplicationTrigger (like DeviceUseTrigger) belongs to a special class of triggers that are triggered directly from the foreground portion of the application. This is done by explicitly calling RequestAsync on the trigger object. ApplicationTrigger is particularly useful for scenarios where the application wants to start an operation in the foreground that should opportunistically continue in the background if the application is no longer in the foreground, and that operation isn't dependent on being tightly coupled with the foreground. The following shows an example of an ApplicationTrigger task that can be used in place of extended execution for most scenarios:

```
var appTrigger = new ApplicationTrigger();

var backgroundTaskBuilder = new BackgroundTaskBuilder();
backgroundTaskBuilder.Name = "App Task";
backgroundTaskBuilder.TaskEntryPoint = typeof(AppTask).FullName;
backgroundTaskBuilder.SetTrigger(appTrigger);
backgroundTaskBuilder.Register();

var result = await appTrigger.RequestAsync();
```

Wrapping Up

This article gives an overview of the application lifecycle and background execution in Windows 10 and introduces a couple of new mechanisms that apps can use to run in the background: extended execution and background tasks. Background tasks are a better choice for low-end and memory-constrained devices (like phones), whereas extended execution is more appropriate for higher-end devices (like desktop PCs). ■

Shawn Henry *is a Senior Program Manager on the Universal Windows Platform team. Reach him on Twitter: @shawnhenry.*

Thanks to the following Microsoft technical experts for reviewing this article: *Anis Mohammed Khaja Mohideen and Abdul Hadi Sheikh*

Adaptive and Interactive Notifications in Windows 10

Thomas Fennel

Few tools in a developer's toolbox impact user engagement like notifications. They permeate the experience within an OS, whether Windows mobile, Windows desktop, Xbox or even HoloLens. There's an undeniable lure notifications provide to users. From helping complete a task such as replying to a message, to providing simple and timely information such as news headlines and even delighting them with an experience they didn't expect—such as a Cortana reminder—notifications provide value to users and engage them with your application.

What Are Notifications in Windows?

Notifications represent a broad category of user engagements throughout an OS. From a user's perspective, notifications are represented in many experiences.

Tiles are the most iconic form of notifications for Windows OSes. You see tiles on the Start screen both on Windows desktop and mobile and in various incarnations across other form factors and device types. Tiles provide several benefits to users.

This article discusses:

- The purpose of notifications
- Using tile and toast templates
- How to use adaptive live tiles and interactive toast notifications

Technologies discussed:

Windows 10, Tiles, Notifications, Toasts

First, they're a quick way to launch apps using primary tiles—these are tiles you pin from the applications list directly to Start by touching and holding or right-clicking on an app in the list. They also serve as a way to get directly to content deeper within an app using secondary tiles—these are tiles a developer pins programmatically with user consent from within the app.

Arguably more important is the ability of tiles to offer engagement with customers by providing content from within the app on the tile itself, via live tiles. These give you a way to reach a user without them actually launching your app—the canonical examples being weather forecasts and sports scores for your favorite teams. Getting that information quickly and in one place on the Start menu can delight users.

Toasts are the notifications that pop up at the top of a mobile device or the bottom-right of a desktop screen. They're interruptive notifications that seek to have users engage with an app by launching it. Before Windows 10, tapping a toast notification could only launch users into the app with some static arguments the developer could set when creating the notification. Also, before Windows 10, there were mobile notifications called alarms and reminders that created larger, modal dialogs to which the user had to respond before doing anything else. These were similar to toasts, and they've been folded into a category of notifications called Action Required Toasts.

Badges are the final type of notification. You see badges predominantly on the lock screen, where they're usually a count of missed items in an application. Or sometimes they're small glyphs that represent status—think of an exclamation point or something similar. Badges are also represented on primary and secondary tiles.

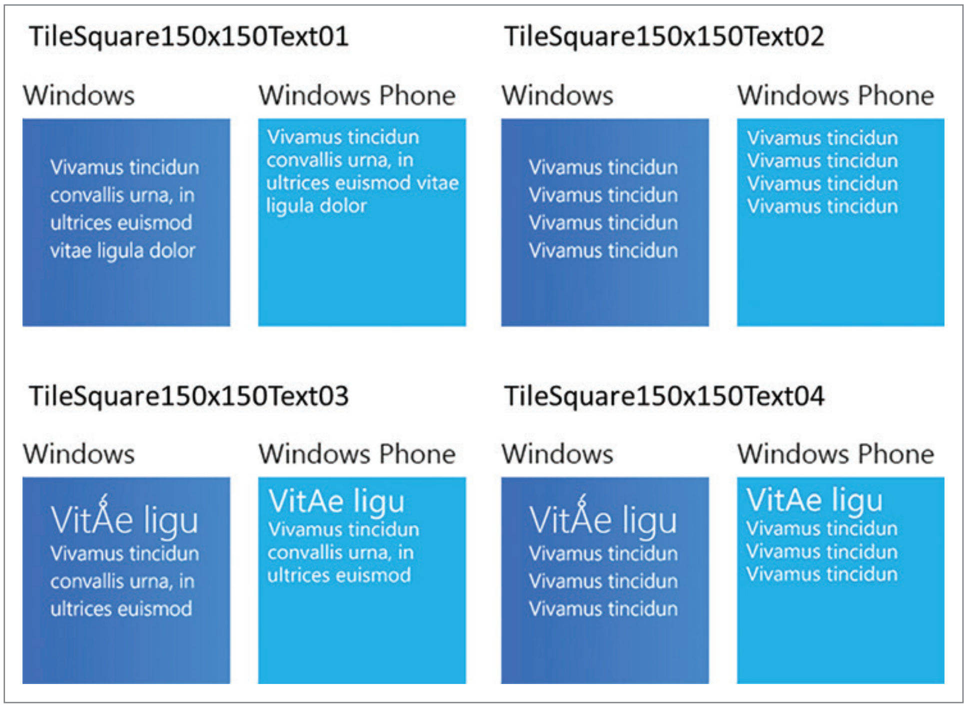

TileSquare150x150Text01
Windows
Windows Phone
Vivamus tincidun convallis urna, in ultrices euismod vitae ligula dolor
Vivamus tincidun convallis urna, in ultrices euismod vitae ligula dolor
TileSquare150x150Text02
Windows
Windows Phone
Vivamus tincidun Vivamus tincidun Vivamus tincidun Vivamus tincidun
Vivamus tincidun Vivamus tincidun Vivamus tincidun Vivamus tincidun
TileSquare150x150Text03
Windows
Windows Phone
VitÅe ligu Vivamus tincidun convallis urna, in ultrices euismod
VitAe ligu Vivamus tincidun convallis urna, in ultrices euismod
TileSquare150x150Text04
Windows
Windows Phone
VitÅe ligu Vivamus tincidun Vivamus tincidun Vivamus tincidun
VitAe ligu Vivamus tincidun Vivamus tincidun Vivamus tincidun

Figure 1 **Four Templates from the Tile Template Catalog That Provide Only Subtle Differences in Style and Layout**

If It Ain't Broke, Don't Fix It

When we started working on adaptive and interactive notifications, we were feverishly driven by the feedback we heard from the developer community about the state of notifications both on Windows Phone and Windows 8. We didn't want to create something new just because ... we've all lived through that in the past and it's how we ended up with such big differences between the notification technologies, infrastructures and developer models between Windows Phone and Windows desktop. Instead, for Windows 10, we focused on what developers liked about the existing notifications framework in both platforms and made a commitment to doing what was needed, rather than simply trying to completely break free from our legacy. It was an interesting time because not only were we starting Windows 10, we were also integrating the Windows Phone and Windows teams into one platform team. During this transition, it wasn't always easy to take such a principled approach, but we knew it was righteous to bring the best of both worlds together, and the result—adaptive and interactive notifications—represents the "One Microsoft" approach.

The Tile and Toast Template Catalogs Wouldn't Scale

So just how do developers design these tile and toast notifications? Previous versions of Windows had a relatively inflexible series of templates that we put into the tile and toast template catalogs. These templates allowed developers to easily and cheaply create notifications that are visually consistent with the Windows design language. However, in order to create all possible permutations of images and text configurations that developers might want, we ended up with hundreds of nearly identical templates.

Figure 1 is a sampling of the tile catalog we offered in Windows Phone 8.1 and Windows 8.1. Text01 is a single line of wrapping text. Text02 is four separate lines of non-wrapping text. Text03 is a header with a single line of wrapping text. Text04 is a header with three lines of non-wrapping text. These clearly won't scale to every possible combination that a developer might want.

For example, what if you need two lines of wrapping text? Unfortunately, that isn't part of the catalog, so you're told to use an image-only template and render a custom bitmap. But that results in blurry text when scaled to different resolutions, much higher data usage when downloaded from the cloud, and a performance impact and unreliability when generating the bitmap in a background task.

Toasts are the notifications that pop up at the top of a mobile device or the bottom right of a desktop screen.

Toasts were even more problematic, as there were really only two styles. There were technically eight templates, but they broke down into toast with image or toast without image, as shown in **Figure 2**. Feedback from developers indicated that not only were the layouts for toasts not flexible enough, they weren't interactive even in simple ways. Toasts with decent amounts of text in them—say three or four lines—couldn't be consumed easily on small screens because we only showed the first two lines of text at most. Even basic interactivity such as being able to expand the notification to show more content wasn't possible with the existing framework.

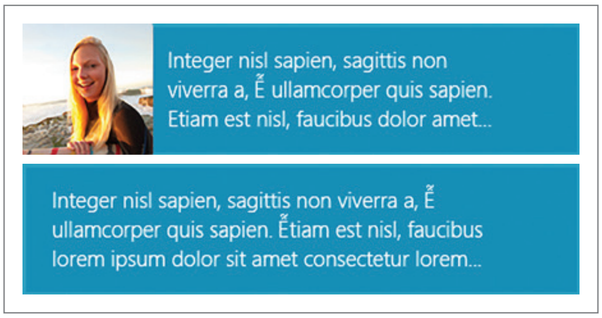

Integer nisl sapien, sagittis non viverra a, Ĕ ullamcorper quis sapien. Etiam est nisl, faucibus dolor amet...
Integer nisl sapien, sagittis non viverra a, Ĕ ullamcorper quis sapien. Ĕtiam est nisl, faucibus lorem ipsum dolor sit amet consectetur lorem...

Figure 2 **Two Templates from the Toast Template Catalog**

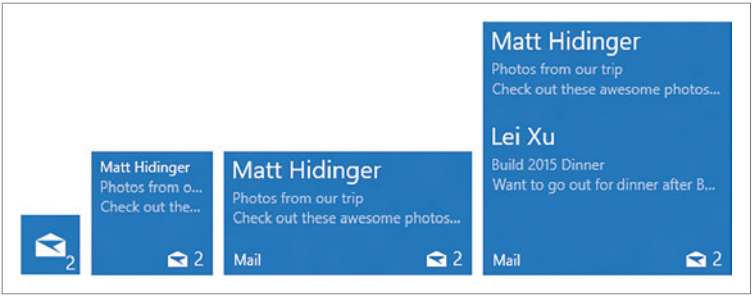


Figure 3 **Small, Medium, Wide and Large Tiles on Windows 10 Desktop**

Not Every Developer Wants (or Can Afford) to Create a Rich Notification

Templates solve a significant problem, because designing, developing and testing a custom-drawn notification such as a live tile is both challenging and costly. Our notification platform runs on a variety of form factors and screen densities, and we support five different tile sizes and two toast sizes, all which would need to be custom designed, developed and tested. Templates can alleviate this pain because developers who want to create a notification that fits within our template language can benefit from minimal development cost and almost zero test cost, and we take on the burden of making sure they look beautiful across all screens.

Templates solve a significant problem, because designing, developing and testing a custom-drawn notification such as a live tile is both challenging and costly.

Developers don't have to worry about margins, padding, font sizes or weights; they can create something beautiful and consistent with our language by pasting in some XML and changing some <text> attributes.

At the same time, there's a powerful need to provide something flexible enough for developers to integrate with their own branding. Developers spend a lot of money and time creating a unique look and building a brand to engage users. Highlighting that brand was a central tenet of our work.

What Does This Mean for Templates?

Templates in Windows 10 effectively serve only one main purpose: to target a specific tile size. In Windows 10 desktop and mobile, we have small, medium and wide tiles. In desktop we also have an additional size for large tiles. We think of these tiles in terms of a logical 4-units-wide-by-4-units-high grid. Small tiles occupy 1x1 units on that grid—little squares, really—hence the small name. Medium tiles occupy 2x2 units on that grid, or medium squares. Sensing a trend? Wide tiles are 4x2 units in that grid and large tiles are the full 4x4 units.

When we were designing adaptive and interactive notifications, we concluded that we could do a lot to collapse the catalog to just the key pivot for size. We ended up with only four templates for tiles—TileSmall, TileMedium, TileWide and TileLarge—and one template for toasts: ToastGeneric. In the future we hope to collapse further and have a TileGeneric template for scenarios where the content is the same between all tile sizes. Then developers can just provide one template for their tiles.

In previous versions of Windows, templates also served to define the animation characteristic for the entire surface. A tile that had animations to show a collection of images cycling through it would have its own template. Similarly, a tile that needed to show a collection of thumbnails all at once that animate and fade would need another template. Now animations are also handled through the new hint-* attributes we've introduced, which I'll discuss later.

Even with all that, we will *not* be deprecating the old template catalog. If developers upgrade to the new Universal Windows Platform (UWP) in Windows 10 without changing the server or client code for constructing their notifications, the app will continue to work great on Windows 10 mobile and desktop. In fact, some things that previously didn't work consistently or at all between mobile and desktop now work great. For example, using any of the ToastImageAndText0* templates on Windows Mobile 10 correctly displays the image in the notification where it previously was omitted.

Adaptive and Interactive Notifications

In Windows 10 we evolved our notification story to provide a richer developer experience and UX by introducing three main features.

The first is adaptive tiles/toasts, which provide a flexible schema to generate more visually rich notifications and have them adaptively show up across different form factors.

The second is interactive toasts, which provide a way to create toast notifications with simple interactions so users can perform quick actions or inline replies inside the notification without activating the app and requiring users to switch context from what they're currently doing.

Figure 4 **The XML Payload to Create a Medium-Size Adaptive Tile**

```
<binding template="TileMedium" branding="logo">
  <group>
    <subgroup>
      <text hint-style="caption">Matt Hidinger</text>
      <text hint-style="captionsubtle">Photos from our trip</text>
      <text hint-style="captionsubtle">Check out these awesome photos
        I took while in New Zealand!</text>
    </subgroup>
  </group>

  <text />

  <group>
    <subgroup>
      <text hint-style="caption">Lei Xu</text>
      <text hint-style="captionsubtle">Build 2015 Dinner</text>
      <text hint-style="captionsubtle">Want to go out for dinner after
        Build tonight?</text>
    </subgroup>
  </group>
</binding>
```

Extreme Performance Linear Scalability

Cache data, reduce expensive database trips, and scale your apps to extreme transaction processing (XTP) with NCache.

In-Memory Distributed Cache

- Extremely fast & linearly scalable with 100% uptime
- Mirrored, Replicated, Partitioned, and Client Cache
- Entity Framework & NHibernate Second Level Cache

ASP.NET Optimization in Web Farms

- ASP.NET Session State storage
- ASP.NET View State cache
- ASP.NET Output Cache provider

Full Integration with Microsoft Visual Studio

- NuGet Package for NCache SDK
- Microsoft Certified for Windows Server 2012 R2

Open Source

Celebrating 10 Years of Market Leadership!

NCache™

FREE Download

sales@alachisoft.com **US: +1 (925) 236 3830** **www.alachisoft.com**

The third is to allow apps to subscribe to events about toast notification history changes, so an app can get notified when its own toast notifications are handled by the system or the user. Essentially this is a way to know what has changed for their app inside Action Center.

Adaptive Live Tile and Adaptive and Interactive Toast Notifications in Action

I'll use the new adaptive and interactive notifications to demonstrate how to build a typical project that showcases the key aspects of tiles and toasts using a real-world scenario. Specifically, I'll address the types of new notifications you can use in an e-mail application.

I'll start with the tile of your e-mail application. In terms of how you create the tile itself, either through the app manifest for your primary tile or through the secondary tile APIs, we haven't made any significant changes in the Windows 10 release, so I'm not going to dive deep into how you do those operations because they're well-documented already. Defining your tile and toast layout is done using a semantic XML schema, commonly referred to as a notification payload. Really what I'll focus on for tiles is how to create a more visually rich tile experience using the new tile templates and corresponding XML payload.

I'll create an adaptive tile notification that shows the user new e-mail notifications. A nice way to engage users is to give them a count of new messages since the last time they opened the application combined with whom the message was from and a few lines of text from the body of the e-mail. **Figure 3** shows how this could look in all four sizes on Windows 10 desktop.

Figure 4 describes the notification payload that represents the tiles in **Figure 3**.

There are a few interesting things to note in **Figure 4**.

First, I'm only showing the payload for the medium tile, seen in the TileMedium binding, because the XML for all four sizes is rather long. In order to reproduce all the tiles you see in **Figure 3**, you'd need to also use the TileSmall, TileWide and TileLarge templates, albeit with very similar content for each. You can put all these bindings into one notification payload.

Second, there's more content in the actual notification than is showing on the tile. You can see this represented by the two different sets of <group> and <subgroup> tags. Why? Because this tile could be displayed on multiple screen densities, so I've included more information

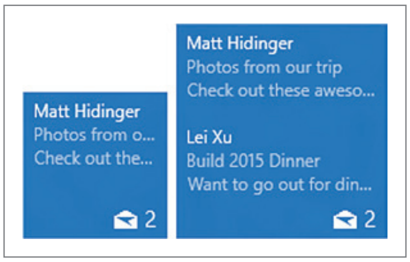


Figure 5 **On the Left, a Medium Tile on a Low-Density Screen Versus a High-Density Screen on the Right**

in case it gets displayed on a high-density screen that can show more content. The <group> and <subgroup> tags help the system know what content to keep together semantically so things aren't cut off in strange places—it's your way to describe a "unit" of information you'd like to keep together if at all possible.

The third thing to notice is the text element hanging out by itself in the middle of the payload. That's actually how you specify a simple blank line, as you can see in the large-tile example.

Finally, the hint-* elements I mentioned earlier can be seen in this payload. These are a large part of the new flexibility we've introduced to help you do things such as the simple text styling shown in **Figure 5**. A wide variety of hints let you specify things such as the opacity of an image, the way a profile photo should be cropped and even how to animate items on a tile for certain targeted scenarios. More information on available hints and other tile details is available on the MSDN Blogs at bit.ly/1NYvsbw.

A nice way to engage users is to give them a count of new messages since the last time they opened the application.

To further illustrate the point about information density, **Figure 5** is an example of the payload from **Figure 4** displayed on a low-density screen versus a high-density screen. I know it seems like a medium tile next to a large tile, but in reality it's a representation of a medium tile on a low-density screen typically found on a lower-priced phone versus a high-density screen found on a flagship phone.

Now I'll move on to the second interesting and engaging thing you could do, this time using adaptive and interactive toast notifications. Again, as with the tile APIs, nothing in the way you create the toast notifier and toast object has changed in the toast APIs. Just like tiles, what's really new here is the payload you can use with those APIs.

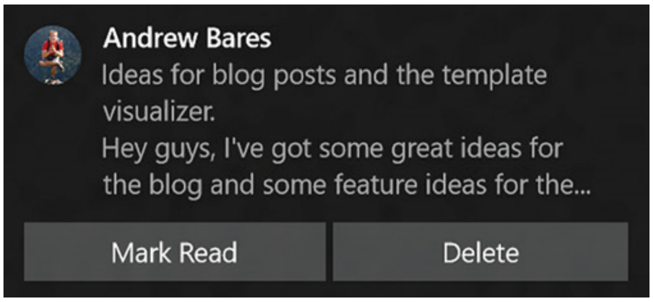
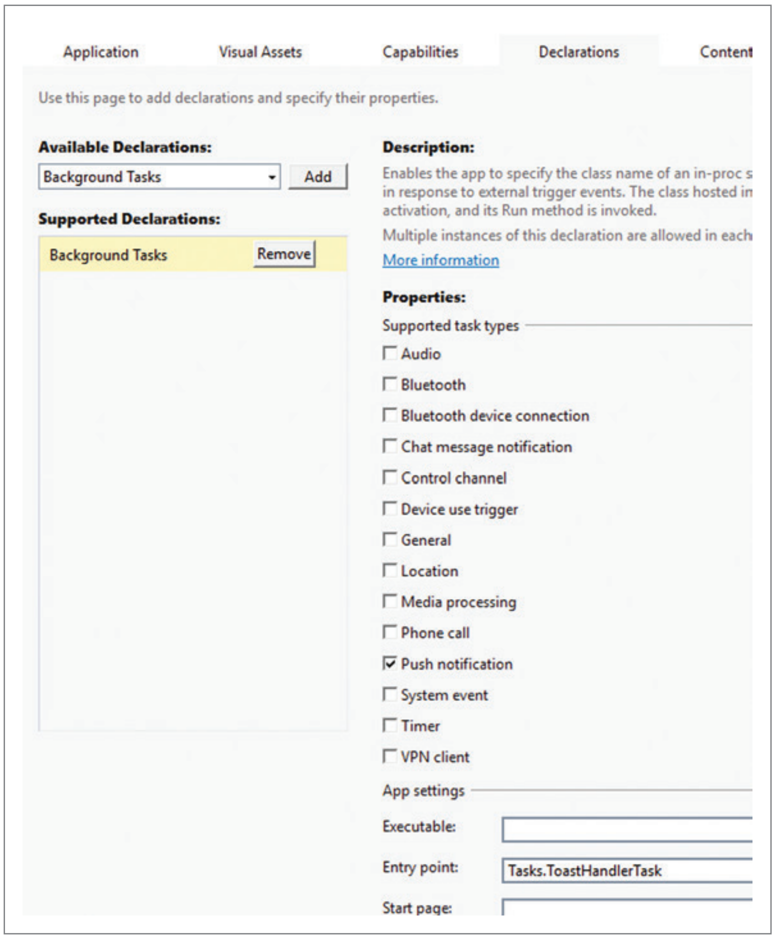

Andrew Bares
Ideas for blog posts and the template visualizer.
Hey guys, I've got some great ideas for the blog and some feature ideas for the...

Figure 6 **An Adaptive and Interactive Toast You Might See Used by an E-mail Application**

Figure 7 **An Adaptive and Interactive Toast You Might See Used by an E-mail Application**

```
<toast>
  <visual>
    <binding template="ToastGeneric">
      <text>Andrew Bares</text>
      <text>Ideas for blog posts and the template visualizer.</text>
      <text>Hey guys, I've got some great ideas for the blog and
        some feature ideas for the...</text>
      <image placement="AppLogoOverride" hint-crop="circle" src="AndrewBares.png" />
    </binding>
  </visual>
  <actions>
    <action activationType="background" content="Mark Read" arguments="read" />
    <action activationType="background" content="Delete" arguments="delete" />
  </actions>
  <audio src="ms-winsoundevent:Notification.Mail"/>
</toast>
```



Use this page to add declarations and specify their properties.

Available Declarations:
Background Tasks — Add

Supported Declarations:
Background Tasks — Remove

Description:
Enables the app to specify the class name of an in-proc s
in response to external trigger events. The class hosted in
activation, and its Run method is invoked.
Multiple instances of this declaration are allowed in each
More information

Properties:
Supported task types
- [ ] Audio
- [ ] Bluetooth
- [ ] Bluetooth device connection
- [ ] Chat message notification
- [ ] Control channel
- [ ] Device use trigger
- [ ] General
- [ ] Location
- [ ] Media processing
- [ ] Phone call
- [x] Push notification
- [ ] System event
- [ ] Timer
- [ ] VPN client

App settings
Executable:
Entry point: Tasks.ToastHandlerTask
Start page:

Figure 8 **Declare a Background Task to Handle Interactive Toast Actions in Package.appxmanifest**

You can create a toast notification to be shown when a new e-mail is received and give users some instant gratification by both giving them several lines of text from the e-mail and a couple of simple quick actions to take on the e-mail itself. Two of the most common things people do with e-mail are to mark an item as read, so they don't have to pay attention to it again when they see their message list later, and delete the mail immediately. On Windows 10 desktop, that kind of toast notification would look like **Figure 6**.

The XML in **Figure 7** would generate the notification shown in **Figure 6**.

As with the tile notification payload I showed earlier, there are some interesting new things to notice in the toast payload from **Figure 7**, as well.

Figure 9 **Handle Arguments or User Input in a Background Task**

```
namespace Tasks
{
  public sealed class ToastHandlerTask : IBackgroundTask
  {
    public void Run(IBackgroundTaskInstance taskInstance)
    {
      // Retrieve and consume the pre-defined arguments and user inputs here.
      var details = taskInstance.TriggerDetails as NotificationActionTriggerDetails;
      var arguments = details.Arguments;
      // Handle either marking the mail as read or deleting it from the database.
    }
  }
}
```

First, unlike the previous templates, you can specify text elements freely and they don't need their own IDs. I've created three lines of text in the toast notification by using three separate text elements.

Second, notice that the first line of text is bold while all subsequent lines are not. For now, you can't use all the hint-* styles available for text on tile notifications in toasts, so this text styling will always apply for toasts until a future release where we enable more hints and styles on toasts, as well. However, you can use some of the hints. For example, notice that the hint-crop attribute *does* work here and I've cropped the image of Andrew to be a nice circle by using it in combination with the AppLogoOverride placement attribute that lets me display an image in place of where the app logo would normally appear. More information on available hints and other toast notification details is available on the MSDN Blogs at bit.ly/1N3o7GY.

Finally, in the actions section of the payload, notice how there are two buttons created by using individual action elements for the common tasks people perform with e-mails. The arguments are what the app will receive when the buttons are clicked and the application is invoked. In this case, I use an activationType of background because I want to handle these actions in the application's background task. Alternatively, you can have an activationType of foreground if you want to launch the app to complete the action or an activationType of protocol if you want to invoke a Web site or an app-to-app communication through a standard protocol launch.

Now that I've created the XML payload needed to display the notification, I need to handle the actions taken by the user on this interactive toast notification.

To start, because I've chosen an activationType of background, I'll need a background task in which I can execute code. That task needs to be declared in the apps Package.appxmanifest, as shown in **Figure 8**.

Then, once the task is registered in the Package.appxmanifest, you can add the code to your background task to actually deal with user actions. In **Figure 9** you can see the simple stub where you would add your own code to handle the actions.

Now you have an app that can dynamically display visually appealing content on your tile and engage users quickly for their most commonly desired actions with a toast. Your app is more useful and delightful to users, and they can quickly get new information and interact with your experience without ever needing to context-switch into your app from what they're currently doing. We look forward to the amazing things you'll do and sincerely hope you'll get better user engagement through adaptive and interactive notifications in Windows 10. ■

Thomas Fennel *is a principal program manager lead at Microsoft in the Windows Developer Ecosystem and Platform division. Reach him at tfennel@microsoft.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Andrew Bares, Matt Hidinger and Lei Xu*

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ORLANDO
ROYAL PACIFIC RESORT AT
UNIVERSAL ORLANDO

NOV 16-20

Fill-up on real-world, practical information and training on the Microsoft Platform as Visual Studio Live! returns to warm, sunny Orlando for the conference more developers rely on to code with industry experts and successfully navigate the .NET highway.

Connect with Visual Studio Live!

twitter.com/vslive
@VSLive

facebook.com
Search "VSLive"

linkedin.com – Join the "Visual Studio Live" group!

EVENT PARTNERS
Microsoft Magenic amazon web services

PLATINUM SPONSORS
GRIDSTORE PLURALSIGHT

GOLD SPONSORS
GitHub e-SignLive by Silanis

5 Great Conferences 1 Great Price

Whether you are an

- Engineer
- Developer
- Programmer
- Software Architect
- Software Designer

You will walk away from this event having expanded your .NET skills and the ability to build better applications.

SESSIONS ARE FILLING UP QUICKLY - REGISTER TODAY!

Use promo code VSLOCT4

Scan the QR code to register or for more event details.

SUPPORTED BY Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA INC. YOUR GROWTH. OUR BUSINESS.

Take the Tour

LIVE! 360 TECH EVENTS WITH PERSPECTIVE

Visual Studio Live! Orlando is part of Live! 360, the Ultimate Education Destination. This means you'll have access to four (4) other co-located events at no additional cost:

SharePoint LIVE! TRAINING FOR COLLABORATION

SQL Server LIVE! TRAINING FOR DBAs AND IT PROS

Modern Apps LIVE! MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

TECHMENTOR IN-DEPTH TRAINING FOR IT PROS

Five (5) events and hundreds of sessions to choose from – mix and match sessions to create your own, custom event line-up – it's like no other conference available today!

TURN THE PAGE FOR MORE EVENT DETAILS.

VSLIVE.COM/ORLANDO

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ORLANDO
ROYAL PACIFIC RESORT AT UNIVERSAL ORLANDO

NOV 16-20

Check Out the Additional Sessions for Devs, IT Pros, & DBAs at Live! 360

LIVE! 360

SharePoint LIVE!
TRAINING FOR COLLABORATION

SharePoint Live! features 12+ developer sessions, including:

- Workshop: Developer OnRamp to the Office 365 & Azure AD Highway! - Andrew Connell
- An Introduction to SharePoint 2013 Add-ins for Developers - Rob Windsor
- SharePoint Developers, Come to the Client Side (We Have Cookies) - Bill Ayers
- SharePoint Development Lifecycle for Solutions and Add-ins - Robert Bogue
- Building Office Add-Ins with Visual Studio - Bill Ayers
- Utilizing jQuery in SharePoint - Get More Done Faster - Mark Rackley
- Using Azure to Replace Server-Side Code in Office 365 - Paul Schaeflein

SQL Server LIVE!
TRAINING FOR DBAs AND IT PROS

SQL Server Live! features 15+ developer sessions, including:

- Encrypting Data in SQL Server - David Dye
- Kick Start! SQL Server 2014 Performance Tips and Tricks - Pinal Dave
- Database Development with SQL Server Data Tools - Leonard Lobel
- Exploring T-SQL Enhancements: Windowing and More - Leonard Lobel
- Python and R for SQL and Business Intelligence Professionals - Jen Stirrup

TECHMENTOR
IN-DEPTH TRAINING FOR IT PROS

TechMentor features IT Pro and DBA sessions, including:

- Workshop: Windows PowerShell Scripting and Toolmaking (BYOL-RA) - Jeffery Hicks
- Your Approach to BYOD Sucks! - Simon May
- Windows Server 2016: Here's What We Know - Don Jones
- Ethical Hacking: Methodologies and Fundamentals - Mike Danseglio & Avril Salter
- DISCUSSION; Career Strategies for the IT Pro - Greg Shields
- PowerShell Unplugged: Stuff I Stole from Snover - Don Jones
- Workshop: SQL Server 2014 for Developers - Leonard Lobel

CHECK OUT THE FULL LIVE! 360 AGENDA AT LIVE360EVENTS.COM

Featured Live! 360 Speakers

ANDREW BRUST

MIGUEL CASTRO

ANDREW CONNELL

DON JONES

DEBORAH KURATA

ROCKFORD LHOTKA

MATTHEW MCDERMOTT

TED NEWARD

JOHN PAPA

BRIAN RANDELL

RACHEL REESE

GREG SHIELDS

AGENDAS AT-A-GLANCE: VISUAL STUDIO LIVE! & MODERN APPS LIVE!

Cloud Computing	Database and Analytics	Lessons Learned and Advanced Practices	Mobile Client	Visual Studio / .NET Framework	Web Development	Windows Client	Modern Apps Live! Sponsored by: Magenic

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Pre-Conference: Sunday, November 15, 2015
6:00 PM	9:00 PM	Dine-A-Round Dinner @ Universal CityWalk - 6:00pm - Meet at Conference Registration Desk to walk over with the group

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Pre-Conference Workshops: Monday, November 16, 2015			
8:00 AM	5:00 PM	**VSM01** Workshop: Service Oriented Technologies: Designing, Developing, & Implementing WCF and the Web API *- Miguel Castro*	**VSM02** Workshop: Triple D: Design, Development, and DevOps *- Billy Hollis and Brian Randell*	**VSM03** Workshop: Busy Developer's Guide to MEANJS *- Ted Neward*	**MAM01** Workshop: Modern App Technology Overview - Android, iOS, Cloud, and Mobile Web *- Nick Landry, Kevin Ford, & Steve Hughes*
5:00 PM	6:00 PM	EXPO Preview			
6:00 PM	7:00 PM	**Live! 360 Keynote:** Microsoft 3.0: New Strategy, New Relevance - *Pacifica 6* *Mary Jo Foley, Journalist and Author; with Andrew Brust, Senior Director, Datameer*			

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Day 1: Tuesday, November 17, 2015				
8:00 AM	9:00 AM	**Visual Studio Live! & Modern Apps Live! Keynote:** The Future of Application Development - Visual Studio 2015 and .NET 2015 *Jay Schmelzer, Director of Program Management, Visual Studio Team, Microsoft*				
9:00 AM	9:30 AM	Networking Break • Visit the EXPO - *Pacifica 7*				
9:30 AM	10:45 AM	**VST01** AngularJS 101 *- Deborah Kurata*	**VST02** A Tour of Azure for Developers *- Adam Tuliper*	**VST03** Busy Developer's Guide to NoSQL *- Ted Neward*	**VST04** Visual Studio, TFS, and VSO in 2015 - What's New? *-Brian Randell*	**MAT01** Defining Modern App Development *- Rockford Lhotka*
11:00 AM	12:15 PM	**VST05** From ASP.NET Site to Mobile App in About an Hour *- Ryan J. Salva*	**VST06** Introduction to Next Generation of Azure PaaS – Service Fabric and Containers *- Vishwas Lele*	**VST07** Real World SQL Server Data Tools (SSDT) *- Benjamin Day*	**VST08** Automate Your Builds with Visual Studio Online or Team Foundation Server *- Tiago Pascoal*	**MAT02** Modern App Architecture *- Brent Edwards*
12:15 PM	2:00 PM	Lunch • Visit the EXPO - *Oceana Ballroom / Pacifica 7*				
2:00 PM	3:15 PM	**VST09** I Just Met You, and "This" is Crazy, But Here's My NaN, So Call(Me), Maybe? *- Rachel Appel*	**VST10** Cloud or Not, 10 Reasons Why You Must Know "Websites" *- Vishwas Lele*	**VST11** Windows 10 for Developers: What's New in Universal Apps *- Nick Landry*	**VST12** Defensive Coding Techniques in C# *- Deborah Kurata*	**MAT03** ALM with Visual Studio Online (TFS) and Git *- Brian Randell*
3:15 PM	4:15 PM	Networking Break • Visit the EXPO - *Pacifica 7*				
4:15 PM	5:30 PM	**VST13** Better Unit Tests through Design Patterns for ASP MVC, WebAPI, and AngularJS *- Benjamin Day*	**VST14** Running ASP.NET Cross Platform with Docker *- Adam Tuliper*	**VST15** Build Your First Mobile App in 1 Hour with Microsoft App Studio *- Nick Landry*	**VST16** Putting CodedUI Tests on Steroids *- Donovan Brown*	**MAT04** Reusing Logic Across Platforms *- Kevin Ford*
5:30 PM	7:30 PM	Exhibitor Reception				

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Day 2: Wednesday, November 18, 2015				
8:00 AM	9:00 AM	**Live! 360 Keynote:** DevOps: What it Means to You - *Pacifica 6 - Sponsored By* PLURALSIGHT *Don Jones, Curriculum Director for IT Pro Content, Pluralsight & Brian Randell, Partner, MCW Technologies*				
9:15 AM	10:30 AM	**VSW01** Mobile App Development with Xamarin and F# *- Rachel Reese*	**VSW02** Notify Your Millions of Users with Notification Hubs *- Matt Milner*	**VSW03** Let's Write a Windows 10 App: A Basic Introduction to Universal Apps *- Billy Hollis*	**VSW04** To Git or Not to Git for Enterprise Development *- Benjamin Day*	**MAW01** Coding for Quality and Maintainability *- Jason Bock*
10:30 AM	11:00 AM	Networking Break • Visit the EXPO - Pacifica 7				
11:00 AM	12:15 PM	**VSW05** Automated UI Testing for Android and iOS Mobile Apps *- James Montemagno*	**VSW06** Busy Developer's Guide to the Clouds *- Ted Neward*	**VSW07** Designing and Building UX for Finding and Visualizing Data in XAML Applications *- Billy Hollis*	**VSW08** Anything C# Can Do, F# Can Do Better *- Rachel Appel & Rachel Reese*	**MAW02** Start Thinking Like a Designer *- Anthony Handley*
12:15 PM	1:45 PM	Birds-of-a-Feather Lunch • Visit the EXPO - *Oceana Ballroom / Pacifica 7*				
1:45 PM	3:00 PM	**VSW09** Stop Creating Forms In Triplicate - Use Xamarin Forms *- Matt Milner*	**VSW10** To Be Announced	**VSW11** Developing Awesome 3D Games with Unity and C# *- Adam Tuliper*	**VSW12** Unit Testing Makes Me Faster: Convincing Your Boss, Your Co-Workers, and Yourself *- Jeremy Clark*	**MAW03** Applied UX: iOS, Android, Windows *- Anthony Handley*
3:00 PM	4:00 PM	Networking Break • Visit the EXPO • Expo Raffle @ 3:30 p.m. - *Pacifica 7*				
4:00 PM	5:15 PM	**VSW13** Go Mobile with C#, Visual Studio, and Xamarin - *James Montemagno*	**VSW14** To Be Announced	**VSW15** Recruiters: The Good, The Bad, & The Ugly *- Miguel Castro*	**VSW16** DI Why? Getting a Grip on Dependency Injection *- Jeremy Clark*	**MAW04** Leveraging Azure Services *- Kevin Ford*
8:00 PM	10:00 PM	Live! 360 Dessert Luau - *Wantilan Pavilion - Sponsored by* amazon web services				

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Day 3: Thursday, November 19, 2015				
8:00 AM	9:15 AM	**VSH01** Getting Started with ASP.NET 5 *- Scott Allen*	**VSH02** Lessons Learned: Being Agile in a Waterfall World *- Philip Japikse*	**VSH03** Windows, NUI and You *- Brian Randell*	**VSH04** Improving Performance in .NET Applications *- Jason Bock*	**MAH01** Building for the Modern Web with JavaScript Applications *- Allen Conway*
9:30 AM	10:45 AM	**VSH05** Build Data Driven Web Applications with ASP. NET MVC *- Rachel Appel*	**VSH06** User Story Mapping *- Philip Japikse*	**VSH07** Building Adaptive Uis for All Types of Windows *- Ben Dewey*	**VSH08** Asynchronous Tips and Tricks *- Jason Bock*	**MAH02** Building a Modern App with Xamarin *- Nick Landry*
11:00 AM	12:15 PM	**VSH09** Automated Cross Browser Testing of Your Web Applications with Visual Studio CodedUI *- Marcel de Vries*	**VSH10** Performance and Debugging with the Diagnostic Hub in Visual Studio *- Sasha Goldshtein*	**VSH11** XAML Antipatterns *- Ben Dewey*	**VSH12** Roslyn and .NET Code Gems *- Scott Allen*	**MAH03** Building a Modern Cross-Platform App *- Brent Edwards*
12:15 PM	1:30 PM	Lunch on the Lanai - *Lanai / Pacifica 7*				
1:30 PM	2:45 PM	**VSH13** Hate JavaScript? Try TypeScript. *- Ben Hoelting*	**VSH14** Advanced Modern App Architecture Concepts *- Marcel de Vries*	**VSH15** WPF MVVM In Depth *- Brian Noyes*	**VSH16** Getting More Out of Visual Studio Online: Integration and Extensibility *- Tiago Pascoal*	**MAH04** DevOps And Modern Applications *- Dan Nordquist*
3:00 PM	4:15 PM	**VSH17** Grunt, Gulp, Yeoman and Other Tools for Modern Web Development *- Ben Hoelting*	**VSH18** The Vector in Your CPU: Exploiting SIMD for Superscalar Performance *- Sasha Goldshtein*	**VSH19** Building Maintainable and Extensible MVVM WPF Apps with Prism *- Brian Noyes*	**VSH20** Readable Code *- John Papa*	**MAH05** Analyzing Results with Power BI *- Steve Hughes*
4:30 PM	5:45 PM	Live! 360 Conference Wrap-Up - *Andrew Brust (Moderator), Andrew Connell, Don Jones, Rockford Lhotka, Matthew McDermott, Brian Randell, & Greg Shields*				

START TIME	END TIME	Visual Studio Live! & Modern Apps Live! Post-Conference Workshops: Friday, November 20, 2015		
8:00 AM	5:00 PM	**VSF01** Workshop: Angular in 0 to 60 *- John Papa*	**VSF02** Workshop: Native Mobile App Development for iOS, Android and Windows Using C# *- Marcel de Vries & Roy Cornelissen*	**MAF01** Workshop: Modern App Development In-Depth - iOS, Android, Windows, and Web *- Brent Edwards, Anthony Handley, & Allen Conway*

Sessions and speakers subject to change.

Linking and Integrating Apps in Windows 10

Arun Singh

Most app developers build or regularly maintain multiple apps. As the apps mature, users frequently demand workflows that involve the multiple apps working together. For example, maybe you have an app that manages product inventory and another app that does checkout. It would be ideal for the two apps to work together to complete a purchase workflow.

One way to solve this challenge is simply to incorporate all functionality into one app. In fact, this is an approach often seen in desktop class applications. However, this is a road fraught with peril. Soon, you end up with a bloated app where most users only use a specific subset of the functionality. The app developer must now manage both UI complexity and updates for the entire app. Even worse, as the UI complexity increases, users—especially those on mobile—begin to gravitate toward more focused options. In fact, the trend has been toward decomposing apps into individual experiences so users can install and use what they need on the go without having to worry about the extraneous bits they don't need.

This article discusses:

- How to make apps communicate with each other
- How to link and integrate apps in Windows 10
- Sharing data between apps more effectively

Technologies discussed:

Windows 10, Windows Runtime, Protocol Declaration, Universal Windows Platform

A second way to solve the problem is to leverage the cloud as a means of communication between apps. That works great until the amount of data gets larger than a certain size or you run into users with limited connectivity. This starts to show up with complaints such as, "I updated my status over here, but it doesn't show up in this other app over there!" In addition, I've always found it a little strange that app developers must resort to the cloud to communicate between two apps sitting on the same device. There has to be a better way.

In this article, I'll look at some of the tools that Windows 10 provides to make communication between apps easier. Communication between apps can take the form of an app launching another app with some data or it could mean apps just exchanging data with each other without having to launch anything. Windows 10 provides tools that can be leveraged for both of these scenarios.

Preparing an App for Deep Linking

Let's start with the example of a product inventory app that can display details about products. Let's also bring a sales app into the mix that can display broad trends about what's selling in which locations and what sort of sales needs to go where. The sales app

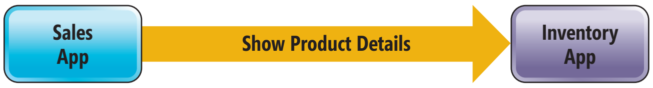
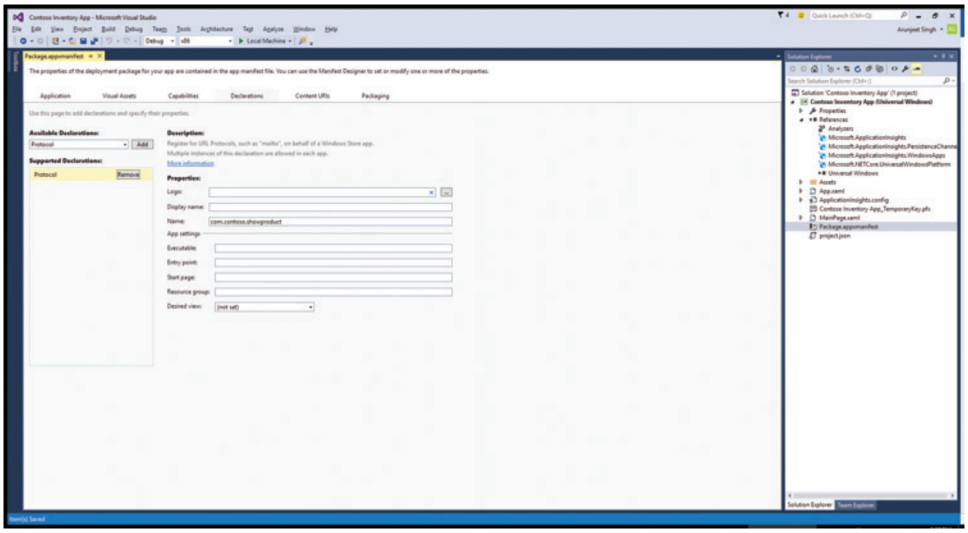


Figure 1 **Sales App Deep Links into the Inventory App**


Package.appxmanifest

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

Use this page to add declarations and specify their properties.

Available Declarations: Protocol

Supported Declarations: Protocol

Description:
Register for URL Protocols, such as "mailto", on behalf of a Windows Store app.
Multiple instances of this declaration are allowed in each app.

Properties:
Name: com.contoso.showproduct
Desired view: (not set)

Figure 2 **The Protocol Declaration**

has a drill-down UX that lets a user see details about individual products. Of course, the most detailed view of a product is in the inventory app. **Figure 1** shows an illustration of the scenario about which I'm talking.

In this scenario, the first thing you need to do is make the inventory app available for launch. To do this you add a protocol declaration to the inventory app's package manifest (package.appxmanifest). The protocol declaration is the inventory app's way of telling the world that it's available to be launched by other apps. **Figure 2** shows what this declaration looks like. Note that I use the protocol name com.contoso.showproduct. This is a good naming convention for custom protocols because Contoso owns the domain contoso.com. The odds of some other app developer mistakenly using the same custom scheme are remote.

Here's the XML the protocol declaration generated:

```
<uap:Extension Category="windows.protocol">
  <uap:Protocol Name="com.contoso.showproduct" />
</uap:Extension>
```

Next, you'll need to add some activation code so the inventory app can respond appropriately when it's launched using the new protocol. The code should go into the inventory app's Application class (App.xaml.cs) because that's where all activations are routed. You override the OnActivated method of the Application class to respond to protocol activations. **Figure 3** shows what that code looks like.

Figure 3 **Handling a Deep Link**

```
protected override void OnActivated(IActivatedEventArgs args)
{
  Frame rootFrame = CreateRootFrame();

  if (args.Kind == ActivationKind.Protocol)
  {
    var protocolArgs = args as ProtocolActivatedEventArgs;
    rootFrame.Navigate(typeof(ProtocolActivationPage), protocolArgs.Uri);
  }
  else
  {
    rootFrame.Navigate(typeof(MainPage));
  }

  // Ensure the current window is active
  Window.Current.Activate();
}
```

You check the kind of incoming IActivatedEventArgs to see if this is a protocol activation. If it is, you typecast the incoming arguments as ProtocolActivatedEventArgs and send the incoming URI onto the ProductDetails page. The ProductDetails page is set up to parse a URI such as com.contoso.showproduct:Details?ProductId=3748937 and show the corresponding product's details. At this point, the inventory app is ready to handle incoming deep links.

The last step to completing this scenario is to enable the sales app to deep link into the inventory app. This is the simplest part of the process. The sales app simply uses the Launcher.LaunchUriAsync API to deep link into the inventory app. Here's what that code might look like:

```
Uri uri = new Uri("com.contoso.showproduct:?ProductId=3748937");
await Launcher.LaunchUriAsync(uri);
```

Sharing Data Between Apps

There are scenarios where apps need to share data but don't necessarily involve sending the user into another app. For example, my sample sales app can display sales by region and even drill down to specific stores. When showing this data categorized by product it would be useful to have the number of units of that product available in a store or region. The best source for this data is the inventory app, but in this case launching the inventory app would be disruptive to the UX. This is exactly the sort of scenario the AppService extension (bit.ly/1JfcVkx) was designed to handle.

The idea is simple: The inventory app provides a "service" that the sales app can invoke. The sales app uses this service to query the inventory app for data it has. The connection between the sales app and the inventory app, once established, can be held open as long as the sales app hasn't been suspended.

Creating the Inventory App Service

Let's look at how the inventory app creates and publishes the app service it intends to provide. App services basically are specialized background tasks. So in order to add an app service you add a Windows Runtime Component (Universal Windows) project to the Visual Studio solution that contains the inventory app. You can find Windows Runtime Component projects in the Add New Project window of Visual Studio under Visual C# | Windows | Universal. The project template is in a similar location for other languages.

Inside the new Windows Runtime Component project, you add a new class called InventoryServiceTask. App services are specialized background tasks because, as shown earlier, you want this code running in the background without showing a UI. To tell the OS that InventoryServiceTask is a background task, you simply need to implement the IBackgroundTask interface. The Run method of the IBackgroundTask interface will be the entry point for the

Figure 4 **Initializing the Inventory App Service in Run Method**

```
namespace Contoso.Inventory.Service
{
  public sealed class InventoryServiceTask : IBackgroundTask
  {
    BackgroundTaskDeferral serviceDeferral;
    AppServiceConnection connection;

    public void Run(IBackgroundTaskInstance taskInstance)
    {
      // Take a service deferral so the service isn't terminated
      serviceDeferral = taskInstance.GetDeferral();

      taskInstance.Canceled += OnTaskCanceled;

      var details = taskInstance.TriggerDetails as AppServiceTriggerDetails;
      connection = details.AppServiceConnection;

      // Listen for incoming app service requests
      connection.RequestReceived += OnRequestReceived;
    }
  }
}
```

inventory app service. In there, you take a deferral to let the OS know that the task should be kept around for as long as the client (the sales app) needs it. You also attach an event handler to the app service-specific RequestReceived event. This event handler will be invoked any time the client sends a request for this service to handle. **Figure 4** shows what the code to initialize the inventory app service looks like.

Now let's look at the RequestReceived handler's implementation. Again, you take a deferral as soon as a request comes in. You'll release this deferral as soon as you're done handling the incoming request. The currency of communication between the app service client and app service is a data structure called ValueSet. ValueSets are key/value dictionaries that can carry simple types such as integers, floating point numbers, strings and byte arrays.

Figure 5 shows how the inventory app service handles incoming requests. It inspects the incoming message for a command and then responds with the right result. In this case, you show the GetProductUnitCountForRegion command to which the service responds with the number of units of the product and the last time the data it has was updated. The service could well be getting this data from a Web service or just retrieving it from an offline cache. The nice thing here is that the client (the sales app) doesn't need to know or care about from where the data comes.

Also shown in **Figure 5** is the implementation of the cancellation handler. It is important that the app service give up the deferral it took gracefully when cancellation is requested. Cancellation of an app service background task could happen either because the client closed the app service connection or the system ran out of resources. Either way, a graceful cancellation ensures that the cancellation isn't seen as a crash by the platform.

Before anyone can call the inventory app service, you must publish it and give it an endpoint. First, you add a reference to the new Windows Runtime Component in the inventory app project. Next, you add an app service declaration to the inventory app project, as shown in **Figure 6**. The Entry point is set to the fully qualified name of the InventoryServiceTask class and Name is the name you'll use to identify this app service endpoint. This is the same name app service clients will use to reach it.

Here's the XML the app service declaration generated:

```
<uap:Extension Category="windows.appService"
  EntryPoint="Contoso.Inventory.Service.InventoryServiceTask">
  <uap:AppService Name="com.contoso.inventoryservice"/>
</uap:Extension>
```

Another piece of information that clients will need to communicate with the inventory app service is the package family name of the inventory app. The simplest way to get this value is to use the Windows.ApplicationModel.Package.Current.Id.FamilyName API within the inventory app. I usually just output this value to the debug window and pick it up from there.

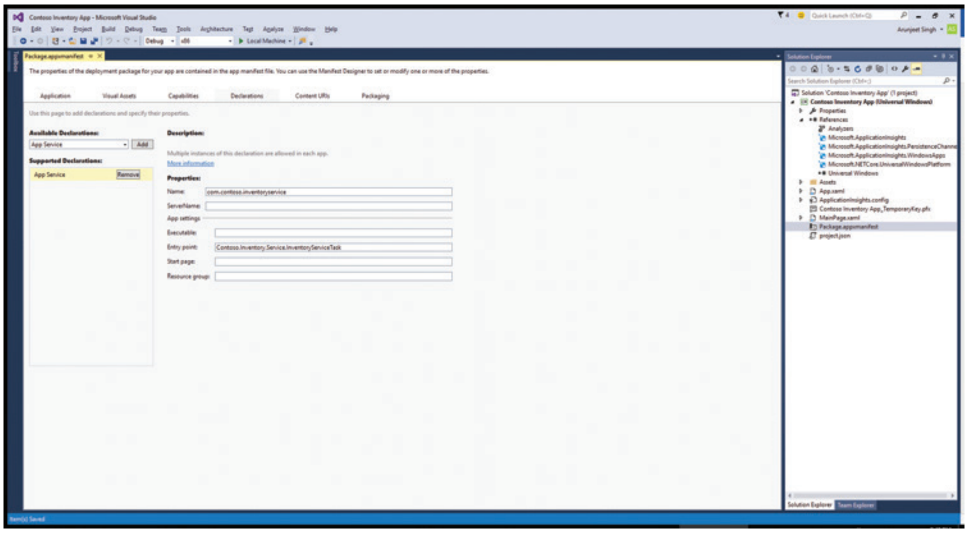
Calling the App Service

Now that the inventory app service is in place, you can call it from the sales app. To call an app service a client can use the AppServiceConnection API. An instance of the AppServiceConnection class requires the name of the app service endpoint and the package family name of the package where the service resides. Think of these two values as the address of an app service.

Figure 7 shows the code the sales app uses to connect to the app service. Notice the AppServiceConnection.AppServiceName property

Figure 5 **Receiving Requests for the Inventory App**

```
async void OnRequestReceived(AppServiceConnection sender,
  AppServiceRequestReceivedEventArgs args)
{
  // Get a deferral so we can use an awaitable API to respond to the message
  var messageDeferral = args.GetDeferral();

  try
  {
    var input = args.Request.Message;
    string command = input["Command"] as string;

    switch(command)
    {
      case "GetProductUnitCountForRegion":
        {
          var productId = (int)input["ProductId"];
          var regionId = (int)input["RegionId"];
          var inventoryData = GetInventoryData(productId, regionId);
          var result = new ValueSet();
          result.Add("UnitCount", inventoryData.UnitCount);
          result.Add("LastUpdated", inventoryData.LastUpdated.ToString());

          await args.Request.SendResponseAsync(result);
        }
        break;

      // Other commands

      default:
        return;
    }
  }
  finally
  {
    // Complete the message deferral so the platform knows we're done responding
    messageDeferral.Complete();
  }
}

// Handle cancellation of this app service background task gracefully
private void OnTaskCanceled(IBackgroundTaskInstance sender,
  BackgroundTaskCancellationReason reason)
{
  if (serviceDeferral != null)
  {
    // Complete the service deferral
    serviceDeferral.Complete();
    serviceDeferral = null;
  }
}
```


Contoso Inventory App - Microsoft Visual Studio

Package.appxmanifest

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

Use this page to add declarations and specify their properties.

Available Declarations: App Service

Supported Declarations: App Service

Description:

Multiple instances of this declaration are allowed in each app.

Properties:

Name: com.contoso.inventoryservice

ServerName:

App settings

Executable:

Entry point: Contoso.Inventory.Service.InventoryServiceTask

Start page:

Resource group:

Figure 6 **The App Service Declaration**

was set to the endpoint name declared in the inventory app's package manifest. Also, the Package Family Name of the inventory app was plugged into the AppServiceConnection.PackageFamilyName property. Once ready, call the AppServiceConnection.OpenAsync API to open a connection. The OpenAsync API returns a status upon completion and this status is used to determine if the connection was established successfully.

Once connected, the client sends the app service a set of values in a ValueSet using the AppServiceConnection.SendMessageAsync API. Notice that the Command property in the ValueSet is set to GetProductUnitCountForRegion. This is a command the app service understands. SendMessageAsync returns a response containing the ValueSet sent back by the app service. Parse out the UnitCount and LastUpdated values and display them. That's it. That's all it takes to communicate with an app service. You put the AppServiceConnection in a using block. This calls the Dispose method on the AppServiceConnection as soon as the using block ends. Calling Dispose is a way for the client to say that it's done talking to the app service and it can now be terminated.

Figure 7 **Calling the Inventory App Service**

```
using (var connection = new AppServiceConnection())
{
  // Set up a new app service connection
  connection.AppServiceName = "com.contoso.inventoryservice";
  connection.PackageFamilyName = "Contoso.Inventory_876gvmnfevegr";

  AppServiceConnectionStatus status = await connection.OpenAsync();

  // The new connection opened successfully
  if (status != AppServiceConnectionStatus.Success)
  {
    return;
  }

  // Set up the inputs and send a message to the service
  var inputs = new ValueSet();
  inputs.Add("Command", "GetProductUnitCountForRegion");
  inputs.Add("ProductId",productId);
  inputs.Add("RegionId", regionId);

  AppServiceResponse response = await connection.SendMessageAsync(inputs);

  // If the service responded with success display the result and walk away
  if (response.Status == AppServiceResponseStatus.Success)
  {
    var unitCount = response.Message["UnitCount"] as string;
    var lastUpdated = response.Message["LastUpdated"] as string;

    // Display values from service
  }
}
```

Wait, Does Microsoft Use These APIs?

Of course, Microsoft uses these APIs. Most Microsoft apps that ship as part of Windows 10 are in fact Universal Windows Platform apps. This includes apps such as Photos, Camera, Mail, Calendar, Groove Music and the Store. The developers who wrote these apps used many of the APIs described here to implement integration scenarios. For example, ever notice the "Get music in Store" link inside the Groove Music app? When you tap or click that link the Groove Music app uses the Launcher.LaunchUriAsync API to get you to the Store app.

Another great example is the Settings app. When you go into Accounts | Your account and try to use the camera to take a new profile picture, it uses an API called Launcher.LaunchUriForResultsAsync to launch the camera app to take that picture. LaunchUriForResultsAsync is a specialized form of LaunchUriAsync described in more detail at aka.ms/launchforresults.

A large number of apps also use app services to communicate information to Cortana in real time. For example, when an app tries to install voice commands that Cortana should respond to, it's actually calling an app service provided by Cortana.

Wrapping Up

Windows 10 comes with powerful tools to aid communication between apps running on the same device. These tools place no restriction or limits on what apps can talk to each other, or what kind of data they can exchange. That is very much by design. The intention is to let apps define their own contracts with each other and extend each other's functionality. This also lets app developers decompose their apps into smaller, bite-sized experiences that are easier to maintain, update and consume. This is very important as users increasingly live their lives across multiple devices and use the device they think is best-suited to a task. All of these APIs are also universal, which means they work on desktops, laptops, tablets, phones and soon on Xbox, Surface Hub and HoloLens. ■

Arun Singh *is a senior program manager on the Universal Windows Platform team. Follow him on Twitter: @aruntalkstech or read his blog at aruntalkstech.com.*

Thanks to the following technical experts for reviewing this article:
Hector Barbera, Jill Bender, Howard Kapustein, Abdul Hadi Sheikh, Stefan Wick and Jon Wiswall

NuGet Features Enhance Windows 10 Development

Jeffrey T. Fritz

Several new tools are now available from the NuGet team. It worked with several teams at Microsoft to deliver a new version of the NuGet client to support the Universal Windows Platform (UWP) and the new Portable Class Libraries (PCLs). The new NuGet tools are available through Tools | Extensions and Updates | Update in Visual Studio 2015, as well as the NuGet distribution site at bit.ly/1MgNt2J. NuGet has also released a new version of the NuGet command-line tool you can download from the same location on dist.nuget.org. This article will review the new capabilities and the process Windows developers need to follow to add NuGet support to their Windows 10 projects.

Project.Json

Beginning with ASP.NET 5, NuGet introduced support for the project.json file to describe project dependencies with a clear definition of the packages upon which you would immediately depend. In ASP.NET 5, this is the only file that defines project configuration. With NuGet 3.1, though, you use this file in your Universal Windows projects and modern PCLs (that target DNX, UWP and the Microsoft .NET Framework 4.6) to define your package references. The good news about this is the "Manage Packages" dialog in Visual Studio will appropriately maintain your packages.config or project.json file for you based on the type of project you're developing.

This article discusses:

- Adding NuGet support to Windows 10 projects
- Defining packages and dependencies
- Ensuring compatibility across packages

Technologies discussed:

Windows 10, Visual Studio 2015, ASP.NET 5, NuGet 3.1

This shift from the packages.config model also lets you "reboot" references in your projects and use the new transitive dependency capabilities of NuGet. Developers and package authors reported to the NuGet team that when they add packages to projects, their packages.config file became polluted with dependencies from their dependent packages.

For example, NHibernate is a package that depends on the Iesi.Collections package. In packages.config, there are two references, NHibernate and Iesi.Collections. When it's time to update NHibernate, there's the question, "Do I also update Iesi.Collections?" The opposite problem also exists. If there's an update for Iesi.Collections, do I need to update NHibernate to support the new features in Iesi.Collections? Developers could end up in this ugly cycle of managing their project's package dependencies brought to them through package references.

The transitive dependencies feature of NuGet abstracts this decision to update package references with improved support for semantic versioning in package definition files (nuspec documents). Developers have specified a range of dependency versions their packages support. When NuGet installs clients, those dependencies add a hard reference to a specific version in the packages.config file and those referenced packages look like any other package reference you've added to your project. You can see a great example of this problem in **Figure 1**.

When I add these things to my project, I really just need Microsoft.AspNet.Mvc, Microsoft.AspNet.Identity.EntityFramework, Newtonsoft.Json and Microsoft.Owin.Security.MicrosoftAccount. The other items referenced by these four packages are just noise, and now I have hard references to specific versions. With the transitive dependencies feature, the versions of these other packages go away. I'm left managing just the four libraries I'll actually use in my project.

The NuGet client will resolve and manage these other packages behind the scenes for you, and keep those references within the constraints of the dependent versions declared by the packages you're using in your project. This should dramatically simplify the project references experience.

Common Local Package Cache

Developers often have a "tribe" of packages and tools they prefer. Why download and install them multiple times on a single workstation when you clearly already have them in one project and want to use them in another? With projects managed by project.json, NuGet downloads and stores a copy of the packages in a global packages folder located in your %userprofile%\.nuget\packages folder. This should reduce disk space used on your workstation. It also prevents extra calls to fetch packages from NuGet.org to get items you already have.

Project.json and common local package cache support is available for ASP.NET 5 with NuGet 3.0, and for other project types starting with NuGet 3.1.

Deprecated Features

Starting with NuGet 3.1 when using project.json, you deprecate support for executing the install.ps1/uninstall.ps1 scripts and delivering elements in the /content package folder. Installing packages with these elements will neither execute the install.ps1 file nor copy content to your project. However, in projects that still use packages.config files, the current behavior is still supported. There are several reasons for this:

- With transitive package restore, picking what to uninstall and install is impossible to do reliably.
- When copying content into the user project, and packages are updated, there's an implicit uninstall process that you can't reliably run.
- NuGet needs to fully support development outside Visual Studio. With the movement to support a full cross-platform .NET development experience, Windows Powershell isn't available in other environments. More developers are also working outside of Visual Studio on .NET code, and they need support.
- Other package managers deliver a great experience for managing and delivering content. NuGet works well as a package manager for the .NET Framework, so continuing to use it is encouraged.
- There's no longer support for the "any" framework. You can no longer place files directly on the root of the build and lib folders and have them delivered to a project. It's important you declare which frameworks your files support so NuGet knows the priority order to resolve those references.
- Solution packages are no longer supported. These packages aren't modifying any specific project's capabilities and were typically used to deliver shared resources that were reused across projects. With the new shared packages folder, these resources may already be on disk from another project.

Figure 1 **The Contents of an ASP.NET MVC packages.config File**

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Antlr" version="3.4.1.9004" targetFramework="net46" />
  <package id="bootstrap" version="3.0.0" targetFramework="net46" />
  <package id="EntityFramework" version="6.1.3" targetFramework="net46" />
  <package id="jQuery" version="1.10.2" targetFramework="net46" />
  <package id="jQuery.Validation" version="1.11.1" targetFramework="net46" />
  <package id="KendoUICore" version="2015.2.624" targetFramework="net46" />
  <package id="Microsoft.AspNet.Identity.Core" version="2.2.1" targetFramework="net46" />
  <package id="Microsoft.AspNet.Identity.EntityFramework"
    version="2.2.1" targetFramework="net46" />
  <package id="Microsoft.AspNet.Identity.Owin" version="2.2.1" targetFramework="net46" />
  <package id="Microsoft.AspNet.Mvc" version="5.2.3" targetFramework="net46" />
  <package id="Microsoft.AspNet.Razor" version="3.2.3" targetFramework="net46" />
  <package id="Microsoft.AspNet.Web.Optimization" version="1.1.3" targetFramework="net46" />
  <package id="Microsoft.AspNet.WebPages" version="3.2.3" targetFramework="net46" />
  <package id="Microsoft.CodeDom.Providers.DotNetCompilerPlatform"
    version="1.0.0" targetFramework="net46" />
  <package id="Microsoft.jQuery.Unobtrusive.Validation"
    version="3.2.3" targetFramework="net46" />
  <package id="Microsoft.Net.Compilers"
    version="1.0.0" targetFramework="net46" developmentDependency="true" />
  <package id="Microsoft.Owin" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Host.SystemWeb" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.Cookies" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.Facebook" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.Google" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.MicrosoftAccount"
    version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.OAuth" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Owin.Security.Twitter" version="3.0.1" targetFramework="net46" />
  <package id="Microsoft.Web.Infrastructure" version="1.0.0.0" targetFramework="net46" />
  <package id="Modernizr" version="2.6.2" targetFramework="net46" />
  <package id="Newtonsoft.Json" version="6.0.4" targetFramework="net46" />
  <package id="Owin" version="1.0" targetFramework="net46" />
  <package id="Respond" version="1.2.0" targetFramework="net46" />
  <package id="WebGrease" version="1.5.2" targetFramework="net46" />
</packages>
```

New Target Frameworks

Another aspect of this new version of NuGet is support for new development frameworks and improved native package support across OSes and architectures. NuGet is reaching further outside the managed

Figure 2 **Target Frameworks Supported with NuGet 3.x**

Description	Base Code	Available Versions
Managed Framework Applications (Windows Forms, Console Applications, Windows Presentation Foundation, ASP.NET)	net	net11, net20, net35, net35-client, net35-full, net4, net40, net40-client, net40-full, net403, net45, net451, net452, net46
ASP.NET 5	dnxcore	dnxcore50
Windows Store	netcore	win8 = netcore45, win81 = netcore451, uap10.0
Windows Phone (appx model)	wpa	wpa81
Windows Phone (Silverlight)	wp	wp7 = sl3-wp, wp71 = sl4-wp71, sl4-wp, wp8 = wp8-, wp81
Silverlight	sl	sl2, sl3 = sl30, sl4 = sl40, sl5 = sl50
Xamarin		mono, MonoMac, Xamarin.Mac, MonoAndroid10, MonoTouch10, Xamarin.iOS10
Compact Framework	net-cf	net20-cf, net35-cf = cf35, net40-cf
Micro Framework	netmf	netmf41, netmf42, netmf43

.NET Framework model to support more ecosystems and empower you to take libraries to previously unreachable environments.

Target framework monikers (TFM) is shorthand used to create a package to declare which frameworks binaries support and which dependencies each framework needs. You'll find folder names in the package's lib and ref folders that use this notation. There are also elements in the package's nuspec dependencies element that declare a target Framework attribute with one of the TFM values to direct the NuGet client to deliver an appropriate library to a consuming project.

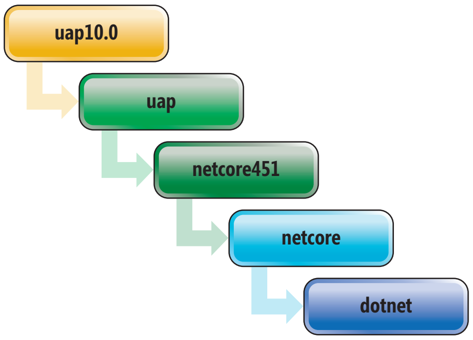


Figure 3 **Hierarchy of Frameworks Inspected for References for a Universal Windows Platform Project**

The following TFMs are still available and the new TFMs being introduced are listed in **Figure 2**.

Those items listed with an equals (=) symbol are synonyms supported by NuGet. That's a lot of support for a lot of different frameworks, but it can be confusing. Do you need to provide support for micro-framework in your managed framework package? How much Silverlight support do you need? You need to answer these questions to best fit the needs of your consumers.

You'll notice there's no explicit call to support PCLs in the table. While NuGet supports those combinations of frameworks, it wants you to have a more forward-compatible moniker for modern PCLs. This will give you greater flexibility in constructing your packages and defining the frameworks you support. NuGet 3.1 introduces the dotnet target moniker for modern PCLs.

Dotnet Target Moniker

In previous versions of NuGet, you could specify the frameworks with a PCL that worked as a collection of TFM abbreviations joined with plus symbols. You could end up with folder names like "portable-net45+win8+wpa81+wp8." That could be confusing and lead to incompatibility issues for your consumers. To make the PCL and cross-platform development experience easier, NuGet introduced the dotnet moniker.

This moniker isn't tied directly to any specific version or framework capabilities. It's an indirect reference that tells NuGet, "This is the reference you should use if it supports the framework and runtime capabilities that you have." The NuGet client then investigates that reference to determine the features and frameworks it supports. This process continues until the NuGet client resolves the exact features supported by the dotnet reference. It will then apply it if and only if it matches the features and requirements of your project. You can reference the dotnet moniker with the .NET Framework 4.5 and later derived framework versions, including Xamarin Android and Xamarin iOS.

This doesn't mean you can simply build a PCL, bundle it with declared dotnet dependencies and be done. If you want to be able to support projects using older versions of Visual Studio and NuGet clients building with traditional portable class libraries, you should still create and place a reference to the full PCL target framework moniker.

When installing a package into a project type that's fully compatible with the dotnet moniker (.NET Framework 4.6, UWP or ASP.NET 5), the dotnet moniker will be sought last. That will happen after attempting to find a reference that matches the framework or less-specific framework of your project. This hierarchy looks like **Figure 3**.

If your project is a modern PCL using project.json that targets any of these frameworks and no other, the dotnet moniker will be analyzed first. That will be followed by the standard PCL resolution strategy, as shown in **Figure 4**.

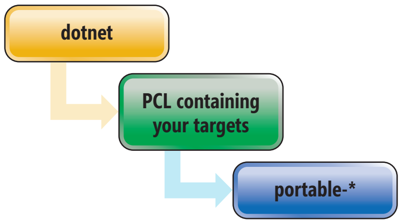


Figure 4 **Hierarchy of Frameworks Inspected for References in a Modern Portable Class Library Project**

NuGet Command Line

The command-like executable for NuGet, nuget.exe, is now available with support to install, update, and restore packages to a project with either a packages.config or project.json file. The pack command continues to work with nuspec files on disk and packages.config files. It hasn't been updated to generate a nuspec file based on a project.json file. To work around this, you'll need to craft your own nuspec file for any new package content you construct with a project.json packages reference. A future release will include an update to address this.

This version of the command-line executable also supports NuGet.org v3 endpoints. This new version of the nuget.org feed provides faster interactions and is a more reliable service. There's built-in redundancy and a content-delivery network enabled to assist in quickly delivering packages. Download a copy of the updated NuGet.exe at bit.ly/1UVOkcU.

If you've installed Windows 10 SDK/Windows 10 tools after upgrading the NuGet extension, the installer will downgrade the extension back to Version 3.1. You'll need to update it again to at least 3.1.1. The version showing up in the Extensions and Updates dialog is 3.1.60724.766. The Windows PowerShell console is 3.1.1.0.

Wrapping Up

These features that support the Windows 10 UWP application development and PCL projects are now available. These changes are the first step in broader use of the package manager and the .NET Framework. Microsoft continues to improve the .NET development experience, and will focus on delivering a package manager to support all .NET developers on any platform building any type of project. ■

Jeffrey T. Fritz *is a senior program manager at Microsoft working on the NuGet team. He enjoys long walks on the beach and killer Web applications that scale in the cloud. Reach him at jefritz@microsoft.com.*

Thanks to the following technical experts for reviewing this article: *Members of the NuGet team at Microsoft*

Syncfusion®
Deliver innovation with ease®

Syncfusion's Unique Big Data Solution for Windows

Q&A with Daniel Jebaraj, Vice President of Syncfusion

Syncfusion, Inc. is a leading provider of .NET and Javascript components, leveraging over 12 years of experience with Windows platforms and mobile devices.

Introduction

Syncfusion is a leading enterprise software vendor delivering a broad range of web, mobile, and desktop controls coupled with a service-oriented approach throughout the entire application lifecycle. With over 10,000 customers, including many Fortune 500 companies, Syncfusion creates powerful frameworks that can answer any challenge, from trading systems to managing vast oil fields.

How can Syncfusion help developers with Big Data?

- The Syncfusion Big Data Platform takes the guesswork out of Apache Hadoop, providing development-time support and tools for those working on Windows.
- We provide the missing pieces to integrate big data solutions with .NET, using our tools on top of open source tools to simplify development.
- We provide a complete production environment for deployment to local or cloud clusters.
- Easy-to-use installers are provided to enable one-click installation of a development-time Hadoop stack on any Windows machine. Check it out at syncfusion.com/bigdata15.

What kind of effort is involved in getting started with big data?

- 15 minutes is all it takes to get started. It is as simple as downloading and running a Windows installer.
- There are no additional dependencies to worry about, no virtual machines. Everything is on Windows.
- With our Big Data Platform, you can seriously cut down on startup time and focus on building your big data solution.

What is the cost of the Syncfusion Big Data platform?

- The platform is completely free for any use, including commercial.
- We offer optional paid commercial support for the platform.
- Samples, patches, and workarounds are provided where applicable, all delivered to our SLA.

What is coming next?

- We are particularly excited about our July 2015 release, which will be the first to include support for Apache Spark.
- Apache Spark allows for processing huge volumes of data in a scalable manner without being tied to the Map Reduce model.
- This has major implications when working with problems in several domains, most notably machine learning.
- With Apache Spark, you have a machine learning solution that can truly scale with your needs. This offers companies of all sizes truly astounding possibilities for benefitting from data.

Why choose Syncfusion?

- Syncfusion has over a decade of experience creating solutions for companies big and small.
- We have the expertise needed to make big data and predictive modeling work for customers on the Windows platform.
- No deployment fees whatsoever—save millions over comparable solutions.

If you are not currently a Syncfusion customer, contact Syncfusion to find out more. Call 1-888-9DOTNET or email sales@syncfusion.com today!

To learn more, please visit our website → **www.syncfusion.com/bigdata15**

Responsive Design for Universal Windows Apps

Mike Jacobs

A Universal Windows app can run on any Windows-based device, from your phone to your tablet or PC. You can even create Universal Windows apps that run on compact devices, such as wearables or household appliances. You can limit your app to a single device family (such as the mobile device family), or you can choose to make the app available on all devices running Windows.

Designing an app that looks good on such a wide variety of devices can be a big challenge. So how do you go about designing an app that provides a great UX on devices with dramatically different screen sizes and input methods? Fortunately, the Universal Windows Platform (UWP) provides a set of built-in features and universal building blocks that help you do just that. This article describes the design features of the UWP and provides recommendations for creating a responsive UI that adapts to difference devices and form factors.

This article discusses:

- Responsive design techniques for Universal Windows Apps running on various device families
- UI building blocks included in the Universal Windows Platform (UWP)
- Profile of device types supported by the UWP and their specific UI characteristics

Technologies discussed:

Universal Windows Platform

Let's start by taking a look at some of the features you get when you create a Universal Windows app. You don't have to do anything to benefit from these features—they're baked into the UWP.

One such feature is platform scaling, which optimizes the UI based on the type of Windows-powered device being used. The system employs an algorithm to normalize the way controls, fonts and other UI elements display on the screen. This scaling algorithm takes into account viewing distance and screen density (pixels per inch) to optimize for perceived size (rather than physical size). The scaling algorithm ensures that a 24-pixel font on a Surface Hub display set 10 feet away is just as legible to the user as a 24-pixel font on a 5-inch smartphone screen that's a few inches away. Because of how the scaling system works, when you design your Universal Windows app you're designing in effective pixels, rather than actual, physical pixels.

Another built-in capability of the UWP is universal input enabled via smart interactions. Although you can design your apps for specific input modes and devices, you aren't required to. That's because Universal Windows apps by default rely on smart interactions. That means you can design around a click interaction without having to know or define whether the click comes from an actual mouse click or the tap of a finger.

Universal Building Blocks

In addition, the UWP provides useful building blocks that make it easier to design apps for multiple device families. These building blocks include universal controls, universal styles and universal templates.

Modern Apps LIVE!
MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

Presented in Partnership with **Magenic**

ORLANDO
ROYAL PACIFIC RESORT AT UNIVERSAL ORLANDO

NOV 16-20

LIVE! 360
TECH EVENTS WITH PERSPECTIVE

5 Great Conferences 1 Great Price

Leading the Modern Apps Way

Presented in partnership with Magenic, Modern Apps Live! brings Development Managers, Software Architects and Development Leads together to break down the complex landscape of mobile, cross-platform, and cloud development and learn how to architect, design and build a complete Modern Application from start to finish.

What sets Modern Apps Live! apart is the singular topic focus; sessions build on each other as the conference progresses, leaving you with a holistic understanding of modern applications, which means a complete picture of what goes into building a modern app for Windows 10, iPad, and Windows Phone devices that all interact with state-of-the-art backend services running in public or private clouds.

SQL Server LIVE! SQL SERVER FOR MODERN DEVELOPERS

Visual Studio LIVE! EXPERT SOLUTIONS FOR .NET DEVELOPERS

SharePoint LIVE! TRAINING FOR COLLABORATION

Modern Apps LIVE! MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

TECHMENTOR IN-DEPTH TRAINING FOR IT PROS

SESSIONS ARE FILLING UP QUICKLY –
REGISTER TODAY!

Use promo code MALOCT1

Scan the QR code to register or for more event details.

SUPPORTED BY
Visual Studio   msdn magazine   Visual Studio MAGAZINE

PRODUCED BY
1105 MEDIA YOUR GROWTH. OUR BUSINESS.

MODERNAPPSLIVE.COM

Universal controls are guaranteed to work well on all Windows-powered devices, from smartphones to Surface Hub displays. They run the gamut from common form controls like radio button and text box to sophisticated controls like grid view and list view, which can generate lists of items from a stream of data and a template. These controls are input-aware and deploy with the proper set of input affordances, event states, and overall functionality for each device family.

The UWP also automatically imbues your apps with a default set of styles that optimizes presentation for every target form factor. You can customize the default styles or replace them completely to create unique, visual experiences. Universal styles deliver a number of capabilities, including:

- A set of styles that automatically gives your app the choice of a light or dark theme and can incorporate the user's accent color preference
- A Segoe-based type ramp that ensures that app text looks crisp on all devices
- Default animations for interactions
- Automatic support for high-contrast modes. These styles were designed with high-contrast in mind, so when an app runs on a device in high-contrast mode, it will display properly
- Automatic support for other languages. The default styles automatically select the correct font for every language that Windows supports. You can even use multiple languages in the same app and they'll be displayed properly
- Built-in support for right-to-left reading order

Finally, the UWP provides universal templates for Adobe Illustrator and Microsoft PowerPoint, and contains everything you need to get started designing UWP apps. These templates feature universal controls and layouts for every universal device size class. To download the templates, go to the Design downloads section of the Windows Dev Center at bit.ly/1KHun6J.

Know the Devices

To provide the best possible UX in your apps, it's important to become familiar with the various device families supported by the UWP. When designing for a particular device, the main considerations include how the app will appear on that device, how the user will interact with that device, and where, when and how the app will be used on that device.

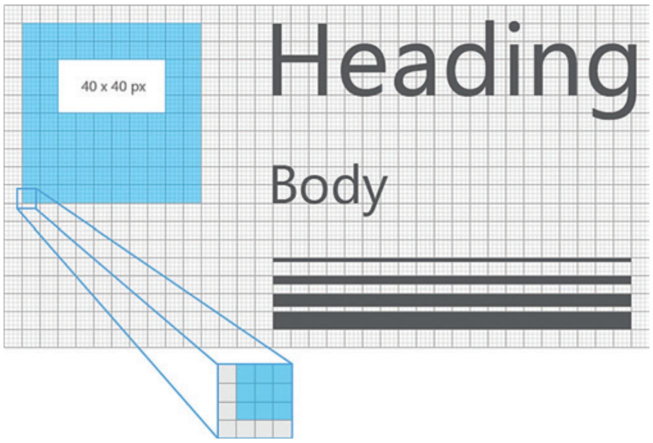


Figure 1 **Align to the 4x4 Pixel Grid for Sharp Text and Image Rendering**

Smartphones The most widely used of all computing devices, phones can do a lot with limited screen real estate and basic inputs. Phones are available in a variety of sizes, with most ranging from 4 to 6 inches diagonally, while phones with screens larger than 6 inches are often called phablets. Though app experiences on phablets are similar to those on standard-size smartphones, the larger screens do enable some important changes in content consumption.

Phones are primarily used in portrait orientation, mostly due to the ease of holding the phone with one hand while interacting with it. Experiences that work well in landscape orientation include viewing photos and video, reading a book, and composing text. Regardless of size and usage modes, phones do share a few characteristics. They're mostly used by just one person—the owner of the device—and tend to be almost always within reach, usually stashed in a pocket or a bag. Phones are also typically used for brief periods of time.

Users interact with their phones through touch and voice. Most phones provide a camera, microphone, movement sensors and location sensors.

Tablets Ultra-portable tablet computers bridge the gap between phones and laptops. Frequently equipped with touchscreens, cameras, microphones and accelerometers, tablets boast screen sizes that typically range from 7 to around 13 inches. Like phones, tablets are primarly used by a single person, the owner. They're most commonly used at home, and are used for longer periods of time than phones. Users interact with tablets through touch, stylus, and sometimes a keyboard and mouse.

PCs and Laptops Windows PCs include a wide array of devices and screen sizes. In general, PCs and laptops can display more info than phone or tablets. Typical screens sizes are 13 inches and larger. Apps on PCs and laptops see shared use, but by one user at a time and usually for longer periods. Apps can be displayed in a windowed view, the size of which is determined by the user. PC and laptop users primarily interact with apps with a mouse and keyboard, but many laptops and some PCs support touch interaction as well. PCs and desktops don't usually have as many built-in sensors as other devices, with most equipped with just a camera and a microphone.

Surface Hub Devices Microsoft Surface Hub is a large-screen, team collaboration device designed for simultaneous use by multiple users. The Surface Hub is available in 55- and 84-inch screen sizes. Apps on Surface Hub see shared use for short periods of time, such as in meetings, and can appear in one of four states: fill (a fixed view that occupies the available stage area), full (standard full-screen view), snapped (a variable view that occupies the right or left sides of the stage) and background (hidden from view while the app is still running, available in task switcher). Surface Hub supports touch, pen, voice, and keyboard interaction, and includes a camera and a microphone.

Windows Internet of Things Devices This emerging class is centered around embedding small electronics, sensors and connectivity within physical objects. These devices are usually connected through a network or the Internet to report on the real-world data they sense, and in some cases act on it. A device can either have no screen (a "headless" device) or be connected to a small screen ("headed" device) that's 3.5 inches or smaller. Inputs and device capabilities can vary greatly from device to device.

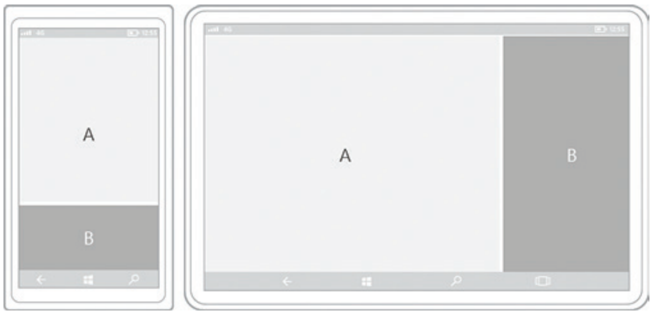


Figure 2 **Reposition Frames to Take Advantage of Larger Screens**

Designing for Specific Devices

Because Windows works behind the scenes to ensure your UI is legible and functional across all devices, you don't have to customize your app for any specific devices or screen sizes. However, there are times when you might want to do so. For example, when your app runs on a PC or laptop, you might show additional content that would clutter up the screen of a smaller device like a phone.

There are many ways to enhance your app for specific screen sizes; some of them are quick and simple, while others require some work.

Let's start by talking a bit more about effective pixels. As I mentioned earlier, when you design your Universal Windows app, you're designing in effective pixels, not actual physical pixels. Effective pixels enable you to focus on the actual perceived size of a UI element without having to worry about the pixel density or viewing distance of different devices. For example, when you design a 1-inch by 1-inch element, that element will appear to be approximately 1-inch on all devices. On an extremely large screen with a high pixel density, the element might be 200 by 200 physical pixels, while on a smaller device like a phone, it might be 150 by 150 physical pixels.

So, how does this impact the way you design your app? You can ignore the pixel density and the actual screen resolution when designing. Instead, design for the effective resolution (the resolution in effective pixels) for a size class. I'll go into detail on size class resolutions a bit later in this article. Also, a quick tip: When creating screen mockups in image-editing programs, be sure to set the PPI to 72 and set the image dimensions to the effective resolution for the size class you're targeting.

To ensure that your apps scale cleanly, it's important to obey the rule of four: Snap your designs to the 4x4 pixel grid by making your margins, sizes and positions of UI elements—including the position of text—a multiple of four effective pixels. **Figure 1** shows design elements that map to the 4x4 pixel grid. The design element will always have crisp, sharp edges. By contrast, design elements that don't map to the 4x4 grid will have blurry, soft edges on some devices.

Despite these resources and capabilities, there are certainly times when you might want to customize your app's UI for a specific device family. For instance, you might need to make the most effective use of space and reduce navigation burden on users. An app designed to look good on a device that has a small screen, such as a phone, will be usable on a PC with a much bigger display, but there will likely be some wasted space. You can customize the app to display more content when the screen is greater than a certain size. For example, a shopping app might display one merchandise category at a time

dtSearch®

Instantly Search Terabytes of Text

dtSearch's document filters support popular file types, emails with multilevel attachments, databases, web data

Highlights hits in all data types; 25+ search options

With APIs for .NET, Java and C++. SDKs for multiple platforms. (See site for articles on faceted search, SQL, MS Azure, etc.)

Visit dtSearch.com for
- hundreds of reviews and case studies
- fully-functional enterprise and developer evaluations

The Smart Choice for Text Retrieval® since 1991

dtSearch.com 1-800-IT-FINDS

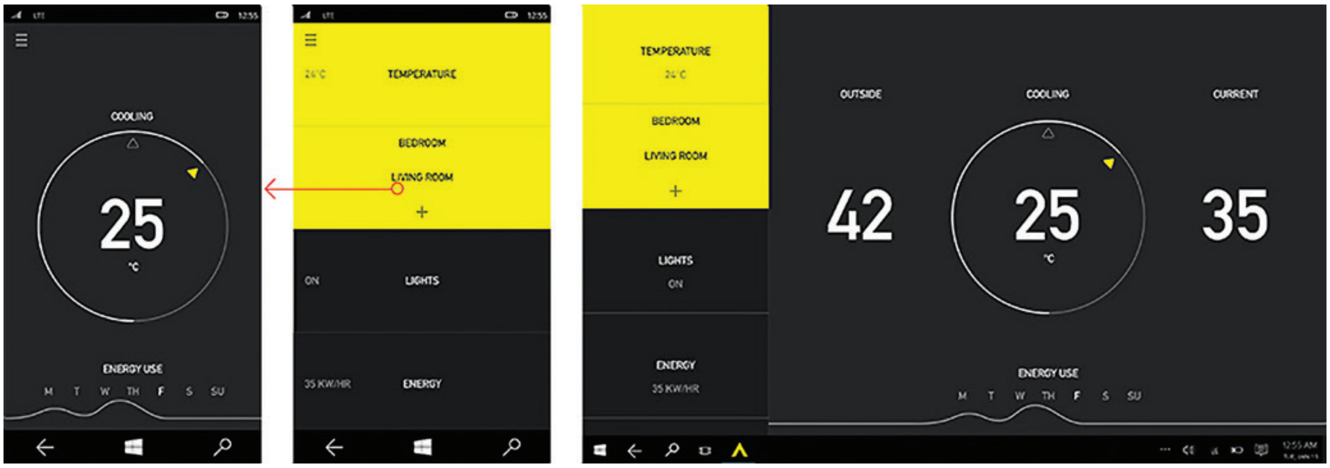


Figure 3 **Rearchitecting App Display for Different Screen Sizes**

on a phone, but show multiple categories and products simultaneously on a PC or laptop. By putting more content on the screen, you reduce the amount of navigation the user needs to perform.

Other scenarios come to mind, for instance, when you need to take full advantage of a device's capabilities. Phones are likely to have a location sensor and a camera, for instance, while a PC might not have either. Your app can detect which capabilities are available and enable features that use them.

Finally, you might want to optimize for input. The universal control library works with all input types (touch, pen, keyboard and mouse), but you can still optimize for certain input types by rearranging your UI elements. For example, if you place navigation elements at the bottom of the screen, they'll be easier for phone users to access—but most PC users expect to see navigation elements toward the top of the screen.

Responsive Design Techniques

When you optimize your app's UI for specific screen widths, you're creating a responsive design. There are a number of useful responsive design techniques that you can use to customize your app's UI and take full advantage of the screen real estate and available capabilities of different device types. There are six techniques to consider: reposition, resize, reflow, reveal, replace and rearchitect.

Repositioning app UI elements is one way to get the most out of each device. In **Figure 2**, the portrait view on a phone or phablet necessitates a scrolling UI, because only one full frame is visible at a time. When the app translates to a tablet or other device that allows two full, on-screen frames, whether in portrait or landscape orientation, frame B can shift to occupy a dedicated space. If you're using a grid for positioning, you can stick to the same grid when UI elements are repositioned.

Figure 4 **Design Breakpoints for Responsive UI**

Size Class	Small	Medium	Large
Width in Effective Pixels	320	720	1024
Typical Screen Size (Diagonal)	4" to 6"	6+" to 12"	13" and wider
Typical Devices	Phones	Tablet, phones with large screen	PCs, laptops, Surface Hubs

You can also optimize the frame by resizing the margins and UI elements to take advantage of smaller or larger screens. This could allow you to augment the reading experience on a larger screen, for example, by growing the content frame, making room for more text and reducing the amount of scrolling. By the same token, different frame sizes offer an opportunity to reflow UI elements based on device and orientation. For instance, when going to a larger screen, it might make sense to use larger containers, add columns, and generate list items in a different way than on a smartphone.

Revealing UI elements is a powerful technique that can surface functionality supported on a specific device—a smartphone camera, for example—while also providing options for taking advantage of different screen sizes and orientations. A common example of revealing or hiding UI applies to media player controls, where the button set is reduced on smaller devices and expanded on larger devices. The media player on a PC, for instance, can handle far more on-screen functionality than it can on a phone.

Part of the reveal-or-hide technique includes choosing when to display more metadata. When real estate is at a premium it's best to show a minimal amount of metadata, while a laptop or desktop PC allows for a significant amount of metadata to be surfaced. Some examples of how to handle showing or hiding metadata include:

- E-mail app: display the user's avatar
- Music app: display more info about an album or artist
- Video app: display more info about a film or a show, such as cast and crew details
- Any app: break apart columns and reveal more details
- Any app: Take something that's vertically stacked and lay it out horizontally; on larger devices, stacked list items can change to reveal rows of list items and columns of metadata

The last two responsive UI techniques are replace and rearchitect. The replace technique lets you switch the UI for a specific device-size class or orientation. For instance, a compact device might display a stacked series of buttons, while on a larger screen those controls are prelaced by tabs running along the top of the screen.

Finally, you can collapse or fork the architecture of your app to better target specific devices. In the example in **Figure 3**, going from the left device to the right device demonstrates the joining of pages. The image depicts a smart home app that, on a larger screen, combines the home controls and the settings pane on a single screen. On a smaller device, the controls and settings are shown on different screens.

Design Breakpoints

The number of device targets and screen sizes across the Windows 10 ecosystem is too great to worry about optimizing your UI for each one. Instead, it's recommended you design for three key widths (also called breakpoints): 320, 720 and 1024 effective pixels. **Figure 4** describes the breakpoints.

When designing for specific breakpoints, consider the amount of screen space available to your app (or the app window). When the app is running full-screen, the app window is the same size of the screen, but in other cases, it's smaller.

When designing for a small UI, every pixel is precious, so it's OK to hide functionality that isn't essential to the primary scenario by placing it in a menu or a toolbar. It's also a good idea to display one column or region of content at a time, and use an icon to represent search instead of showing a search box.

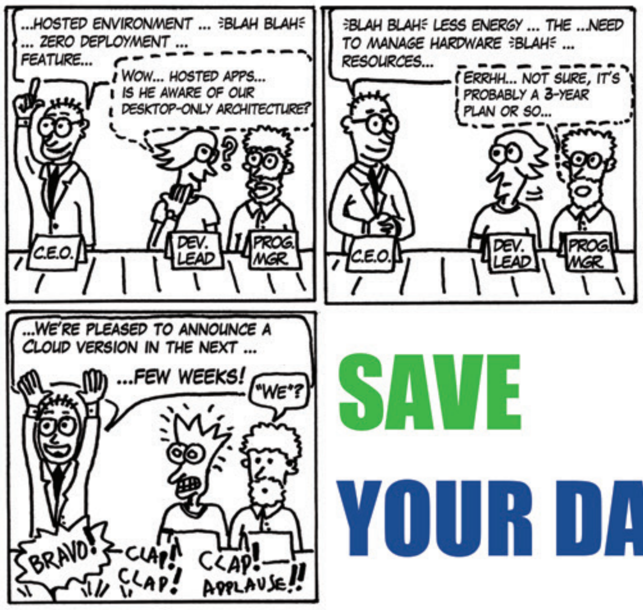
Move up to medium-sized UIs and you can take adavantage of the extra space to display a search box (if you have one) and lay out the primary content in two columns or regions. As you step up to a large UI, more functionality and content can be displayed, reducing the number of clicks and UI interactions required to get to content or perform actions.

Keep in mind that a width of 320 effective pixels could mean your app is running on a phone or in a small window on a PC with a large screen, so be sure to consider the primary input for the device—mouse or touch. On touch devices, you can ease navigation and interaction on hand-held devices by placing navigation and command elements at the bottom of the screen where they're easily reached with a thumb. When using a mouse, however, users expect navigation elements to appear at the top of the screen. Your app can use the Windows.UI.ViewManagement.UIViewSettings.UserInteractionMode property to discover the primary input device and adjust its UI accordingly.

The Universal Windows Platform opens up a spectrum of devices for your apps, from wearables with tiny screens to the enormous Surface Hub. By implementing responsive design techniques and taking advantage of the platform's built-in features and building blocks (controls and styles), it's possible to create a UI that looks great on devices of all sizes and shapes. ■

Mike Jacobs *works as a senior content developer for Microsoft. For many years, he's documented the graphics and presentation aspects of Microsoft technologies, from Direct2D to Windows Presentation Foundation and Silverlight.*

Thanks to the following Microsoft technical experts for reviewing and contributing to this article: *Jeff Arnold, Eugene Gavrilov, Ross Heeter, Chigusa Sansen, Joe Tobens, as well as to Mike Bottemiller (Murphy & Associates)*


...HOSTED ENVIRONMENT ... ⁝BLAH BLAH⁝ ... ZERO DEPLOYMENT ... FEATURE...
WOW... HOSTED APPS... IS HE AWARE OF OUR DESKTOP-ONLY ARCHITECTURE?
?
C.E.O.
DEV. LEAD
PROG. MGR.
⁝BLAH BLAH⁝ LESS ENERGY ... THE ...NEED TO MANAGE HARDWARE ⁝BLAH⁝ ... RESOURCES...
ERRHH... NOT SURE, IT'S PROBABLY A 3-YEAR PLAN OR SO...
C.E.O.
DEV. LEAD
PROG. MGR.
...WE'RE PLEASED TO ANNOUNCE A CLOUD VERSION IN THE NEXT ...
...FEW WEEKS!
"WE"?
BRAVO!
CLAP!
CLAP!
CLAP!
APPLAUSE!!

SAVE YOUR DAY!

SUPPORT FOR VISUAL STUDIO 2015

Use CodeFluent Entities

CodeFluent Entities is a unique product integrated into Visual Studio that allows you to generate database scripts, code (C#, VB), web services and UIs.

*"CodeFluent Entities is a masterpiece software product envisioned and implemented by a group of very talented and experienced people. All that is achieved upon delivering high quality and standardized code and database layers, neatly stitched together and working in unison, lifting the burden from the developers to worry about this aspect on the development process, which could be called plumbing."**

Renato Xavier - Colombaroli - Brazil

** Source : http://visualstudiogallery.msdn.microsoft.com/B6299BBF-1EF1-436D-B618-66E8C16AB410*

To get a license worth $399 for free
Go to www.softfluent.com/forms/msdn-2015

CodeFluent Entities
tools for developers, by developers

More information: www.softfluent.com **Contact us:** info@softfluent.com

Adaptive Apps for Windows 10

Clint Rutkas and Rajen Kishna

With the Universal Windows Platform (UWP) in Windows 10, apps can now run on a variety of device families and automatically scale across the different screen and window sizes, supported by the platform controls. How do these device families support user interaction with your apps, and how do your apps respond and adapt to the device on which it's running? We explore this and the tools and resources Microsoft provides in the platform, so you don't have to write and maintain complex code for apps running across different device types.

Let's start with an exploration of the responsive techniques you can use to optimize the UI for different device families. We'll then go deeper into how your app can adapt to specific device capabilities.

Before we dive into controls, APIs and code, let's take a moment to explore the device families we're talking about. Simply put: A device family is a group of devices in a specific form factor, ranging from IoT devices, smartphones, tablets and desktop PCs, to Xbox game consoles, large-screen Surface Hub devices and even wearables. Apps will work across all these device families, but it's important to consider the device families it could be used on when designing your apps.

While there are many device families, the UWP is designed such that 85 percent of its APIs are fully accessible to any app, independent of where it runs. What's more, when looking at the top 1,000 apps, 96.2 percent of all APIs used are accounted for in the base Universal Windows API set. The bulk of functionality is present and available as part of the UWP, with specialized APIs on each device available to further tailor your app.

This article discusses:

- Improvements to XAML in Windows 10 that enable responsive UI
- Employing API contract detection to enable client-appropriate features and capabilities
- The role of API contracts in enabling cross-device functionality

Technologies discussed:

Universal Windows Platform, API Contracts, XAML, Continuum

Welcome Back, Windows

One of the biggest changes in how apps are used in Windows is something you're already very familiar with: running apps in a window. Windows 8 and Windows 8.1 allow apps to run full-screen, or side-by-side with as many as four apps simultaneously. Windows 10, by contrast, lets the user arrange, resize and position apps in any way he wants. The new approach under Windows 10 gives the user better UI flexibility, but may require some work on your end to optimize for it. The improvements in XAML on Windows 10 introduce a number of ways to implement responsive techniques in your app, so it looks great no matter the screen or window size. Let's explore three of these approaches.

VisualStateManager In Windows 10 the VisualStateManager class has been expanded with two mechanisms to implement responsive design in your XAML-based apps. The new VisualState.StateTriggers and VisualState.Setters APIs allow you to define visual states that correspond to certain conditions. Visual states can change based on app window height and width, by using the built-in AdaptiveTrigger as the VisualState's StateTrigger and setting the MinWindowHeight and MinWindowWidth properties. You can also extend Windows.UI.Xaml.StateTriggerBase to create your own triggers, for instance triggering on device family or input type. Take a look at the code in **Figure 1**.

In the example in **Figure 1**, the page displays three TextBlock elements stacked on top of each other in its default state. The VisualStateManager has an AdaptiveTrigger defined at a MinWindowWidth of 720, which causes the orientation of the StackPanel to change to Horizontal when the window is at least 720 effective pixels wide. This lets you make use of the extra horizontal screen real estate when users resize the window or go from portrait to landscape mode on a phone or tablet device. Keep in mind, if you define both the width and height property, your trigger will only fire if the app meets both conditions simultaneously. You can explore the State triggers sample on GitHub (`wndw.ms/XUneob`) to view more scenarios using triggers, including a number of custom triggers.

Relative Panel In the **Figure 1** example, a StateTrigger is used to change the Orientation property of a StackPanel. The many container elements in XAML, combined with StateTriggers, let you

Figure 1 **Create Custom State Triggers**

```
<Page>
  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup>
        <VisualState>

          <VisualState.StateTriggers>
          <!-- VisualState to be triggered when window
            width is >=720 effective pixels. -->
            <AdaptiveTrigger MinWindowWidth="720" />
          </VisualState.StateTriggers>

          <VisualState.Setters>
            <Setter Target="myPanel.Orientation"
                    Value="Horizontal" />
          </VisualState.Setters>

        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>

    <StackPanel x:Name="myPanel" Orientation="Vertical">
      <TextBlock Text="This is a block of text. It is text block 1. "
                 Style="{ThemeResource BodyTextBlockStyle}"/>
      <TextBlock Text="This is a block of text. It is text block 2. "
                 Style="{ThemeResource BodyTextBlockStyle}"/>
      <TextBlock Text="This is a block of text. It is text block 3. "
                 Style="{ThemeResource BodyTextBlockStyle}"/>
    </StackPanel>

  </Grid>
</Page>
```

manipulate your UI in a large number of ways, but they don't offer a way to easily create a complex, responsive UI where elements are laid out relative to each other. That's where the new RelativePanel comes in. As shown in **Figure 2**, you can use a RelativePanel to lay out your elements by expressing spatial relationships between elements. This means that you can easily use the RelativePanel together with AdaptiveTriggers to create a responsive UI where you move elements based on available screen space.

As a reminder, the syntax you use with attached properties involves extra parentheses, as shown here:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState>

      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="720" />
      </VisualState.StateTriggers>

      <VisualState.Setters>
        <Setter Target="GreenRect.(RelativePanel.RightOf)"
                Value="BlueRect" />
      </VisualState.Setters>

    </VisualState>
```

You can check out additional scenarios using RelativePanel in the Responsiveness techniques sample on GitHub (wndw.ms/cbdL0q).

SplitView App window size impacts more than the content displayed on app pages; it may require that navigation elements respond to changes in the size of the window itself. The new SplitView control introduced in Windows 10 is typically used to create a top-level navigation experience that can be adjusted to behave differently according to the size of the app window. Keep in mind that while this is one of the common use cases for the SplitView, it's not strictly limited to this use. The SplitView is divided into two distinct areas: pane and content.

A number of properties on the control can be used to manipulate presentation. First, DisplayMode specifies how the pane is rendered in relation to the Content area, with four available modes—Overlay, Inline, CompactOverlay and CompactInline. **Figure 3** shows examples of the Inline, Overlay and CompactInline modes rendered in an app.

The PanePlacement property displays the Pane on either the left (default) or right side of the Content area. The OpenPaneLength property specifies the width of the pane when it's fully expanded (default 320 effective pixels).

Note that the SplitView control does not include a built-in UI element for users to toggle the state of the pane, like the common "hamburger" menu often found in mobile apps. If you want to expose this behavior, you must define this UI element in your app and provide code to toggle the IsPaneOpen property of the SplitView.

Want to explore the full set of features the SplitView offers? Be sure to check out the XAML navigation menu sample on GitHub (wndw.ms/qAUVr9).

Bringing the Back Button

If you developed apps for earlier versions of Windows Phone, you might be accustomed to every device having a hardware or software back button, allowing users to navigate their way back through your app. For Windows 8 and 8.1, however, you had to create your own UI for back navigation. To make things easier when targeting multiple device families in your Windows 10 app, there's a way to ensure a consistent back-navigation mechanism for all users. This can help free up some UI space in your apps going forward.

To enable a system back button for your app, even on device families that don't have a hardware or software back button (such as laptop and desktop PCs), use the AppViewBackButtonVisibility property in the SystemNavigationManager class. Simply get the SystemNavigationManager for the current view and set the back button visibility as shown in the following code:

```
SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility =
  AppViewBackButtonVisibility.Visible;
```

The SystemNavigationManager class also exposes a BackRequested event, which fires when the user invokes the system-provided button, gesture or voice command for back navigation. This means that you can handle this single event to consistently perform back navigation in your app across all device families.

Figure 2 **Express Spatial Relationships with RelativePanel**

```
<RelativePanel BorderBrush="Gray" BorderThickness="10">

  <Rectangle x:Name="RedRect" Fill="Red" MinHeight="100" MinWidth="100"/>

  <Rectangle x:Name="BlueRect" Fill="Blue" MinHeight="100" MinWidth="100"
             RelativePanel.RightOf="RedRect" />

  <!-- Width is not set on the green and yellow rectangles.
       It's determined by the RelativePanel properties. -->
  <Rectangle x:Name="GreenRect" Fill="Green"
             MinHeight="100" Margin="0,5,0,0"
             RelativePanel.Below="RedRect"
             RelativePanel.AlignLeftWith="RedRect"
             RelativePanel.AlignRightWith="BlueRect"/>

  <Rectangle Fill="Yellow" MinHeight="100"
             RelativePanel.Below="GreenRect"
             RelativePanel.AlignLeftWith="BlueRect"
             RelativePanel.AlignRightWithPanel="True"/>

</RelativePanel>
```

Benefits of Continuum

Last, but not least, we'd like to mention one of our personal favorites: Continuum on Windows 10. With Continuum, Windows 10 adjusts your experience to what you want to do and how you want to do it. If your app is running on a 2-in-1 Windows PC, for example, implementing Continuum in your app lets users use touch or a mouse and keyboard to optimize productivity. With the UserInteractionMode property in the UIViewSettings class, your app can determine whether the user is interacting with the view using touch or a mouse and keyboard with just one line of code:

```
UIViewSettings.GetForCurrentView().UserInteractionMode;
// Returns UserInteractionMode.Mouse or UserInteractionMode.Touch
```

After detecting the mode of interaction, you can optimize the UI of your app, doing things like increasing or decreasing margins, showing or hiding complex features, and more. Check out the TechNet article, "Windows 10 Apps: Leverage Continuum Feature to Change UI for Mouse/Keyboard Users Using Custom StateTrigger," by Lee McPherson (wndw.ms/y3gB0J) showing how you can combine the new StateTriggers and the UserInteractionMode to build your own custom Continuum StateTrigger.

Apps Adaptive

Apps that can respond to changes in screen size and orientation are useful, but to achieve compelling, cross-platform functionality, the UWP provides developers with two additional types of adaptive behavior:

- **Version adaptive** apps respond to different versions of the UWP by detecting available APIs and resources. For example, you might want your app to use some newer APIs that are only present on devices running the most recent versions of the UWP, while continuing to support customers who haven't upgraded yet.
- **Platform adaptive** apps respond to the unique capabilities available on different device families. So an app may be built to run on all device families, but you might want to use some mobile-specific APIs when it runs on a mobile device like a smartphone.

As noted earlier, with Windows 10 the vast majority of UWP APIs are fully accessible to any app, regardless of the device on which it's running. Specialized APIs associated with each device family then allow developers to further tailor their apps.

The fundamental idea behind adaptive apps is that your app checks for the functionality (or feature) it needs, and only uses it when it's available. In the past, an app would check the OS version and then call APIs associated with that version. With Windows 10, your app can check at runtime whether a class, method, property, event or API contract is supported by the current OS. If so, the app then calls the appropriate API. The ApiInformation class located in the Windows.Foundation.Metadata namespace contains several static methods (such as IsApiContractPresent, IsEventPresent and IsMethodPresent), which are used to query for APIs. Here's an example:

```
using Windows.Foundation.Metadata;

if(ApiInformation.IsTypePresent("Windows.Media.Playlists.Playlist"))
{
  await myAwesomePlaylist.SaveAsAsync( ... );
}
```

This code does two things. It makes a runtime check for the presence of the Playlist class, then calls the SaveAsAsync method on an instance of the class. Also note the ease of checking for the presence of a type on the current OS, using the IsTypePresent API. In the past, such a check might have required LoadLibrary, GetProcAddress, QueryInterface, Reflection, or use of the "dynamic" keyword and others, depending on language and framework. Also note the strongly typed reference when making the method call. When using either Reflection or "dynamic," you lose static compile-time diagnostics that would, for example, inform you if you misspelled the method name.

Detecting with API Contracts

At its heart, an API contract is a set of APIs. A hypothetical API contract could represent a set of APIs containing two classes, five interfaces, one structure, two enums and so on. We group logically related types into an API contract. In many ways, an API contract represents a feature—a set of related APIs that together deliver some particular functionality. Every Windows Runtime API from Windows 10 onward is a member of some API contract. The documentation at msdn.com/dn706135 describes the variety of API contracts available. You'll see that most of them represent a set of functionally related APIs.

The use of API contracts also provides you, the developer, with some additional guarantees; the most important being when a platform implements any API in an API contract, it must implement *every* API in that contract. In other words, an API contract is an atomic unit, and testing for support of that API contract is equivalent to testing that each and every API in the set is supported. Your app can call any API in the detected API contract without having to check for each API individually.

The largest and most commonly used API contract is the Windows.Foundation.UniversalApiContract. It contains nearly all of the APIs in the Universal Windows Platform. If you wanted to see if the current OS supports the UniversalApiContract, you would write the following code:

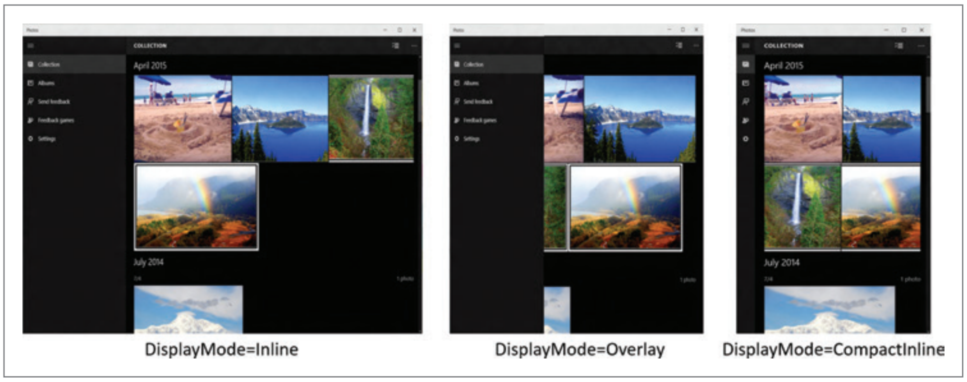


Figure 3 **DisplayMode Navigation Elements**

```
if (ApiInformation.IsApiContractPresent(
  "Windows.Foundation.UniversalApiContract"), 1, 0)
{
  // All APIs in the UniversalApiContract version 1.0 are available for use
}
```

Right now the only extant version of the UniversalApiContract is version 1.0, so this check is slightly silly. But a future update of Windows 10 could introduce additional APIs, yielding a version 2.0 of the UniversalApiContract that includes the new universal APIs. In the future an app that wants to run on all devices, but also wants to use new version 2.0 APIs, could use the following code:

```
if (ApiInformation.IsApiContractPresent(
  "Windows.Foundation.UniversalApiContract"), 2, 0)
{
  // This device supports all APIs in UniversalApiContract version 2.0
}
```

If your app only needed to call one single method from version 2.0, it could check for the method directly using IsMethodPresent. In such a case, you can use whichever approach you find the easiest.

There are other API contracts besides the UniversalApiContract. Most represent a feature or set of APIs not universally present on all Windows 10 platforms and instead are present on one or more specific device families. As mentioned earlier, you no longer need to check for a particular type of device and then infer support for an API. Simply check for the set of APIs your app wants to use.

I can now rewrite my original example to check for the presence of the Windows.Media.Playlists.PlaylistsContract, instead of just checking for the present of the Playlist class:

```
if(ApiInformation.IsApiContractPresent(
  "Windows.Media.Playlists.PlaylistsContract"), 1, 0)
{
  // Now I can use all Playlist APIs
}
```

Any time your app needs to call an API that isn't present across all device families, you must add a reference to the appropriate Extension SDK that defines the API. In Visual Studio 2015, go to the Add Reference dialog and open the Extensions tabs. There you can find the three most important extensions: Mobile Extension, Desktop Extension and IoT Extension.

All your app needs to do, however, is check for the presence of the desired API contract and call the appropriate APIs conditionally. There's no need to worry about the type of device. Now the question is: I need to call the Playlist API but it's not a universally available API. The documentation (bit.ly/1QkYqky) tells me in which API contract the class is. But which Extension SDKs define it?

As it turns out, the Playlist class is (currently) only available on Desktop devices, not on Mobile, Xbox and other device families. So you must add a reference to the Desktop Extension SDK before any of the prior code compiles.

Visual Studio team member and occasional *MSDN Magazine* author Lucian Wischik created a tool that can help. It analyzes your app code when it calls into a platform-specific API, verifying that an adaptivity check was done around it. If no check was done, the analyzer reports a warning and provides a handy "quick-fix" to insert the correct check in the code, simply by pressing Ctrl+Dot or clicking on the lightbulb. (See bit.ly/1JdXTeV for more details.) The analyzer can also be installed via NuGet (bit.ly/1KU9ozj).

Let's wrap up by looking at more complete examples of adaptive coding for Windows 10. First, here's some code that is *not* correctly adaptive:

```
// This code will crash if called from IoT or Mobile
async private Task CreatePlaylist()
{
  StorageFolder storageFolder = KnownFolders.MusicLibrary;
  StorageFile pureRockFile = await storageFolder.CreateFileAsync("myJam.mp3");
  Windows.Media.Playlists.Playlist myAwesomePlaylist =
    new Windows.Media.Playlists.Playlist();

  myAwesomePlaylist.Files.Add(pureRockFile);

  // Code will crash here as this is a Desktop-only call
  await myAwesomePlaylist.SaveAsAsync(KnownFolders.MusicLibrary,
    "My Awesome Playlist", NameCollisionOption.ReplaceExisting);
}
```

Now let's see the same code, adding a line that verifies that the optional API is supported on the target device before calling the method. This will prevent runtime crashes. Note that you'll likely want to take this example further and not display the UI that calls the CreatePlaylist method if your app detects that the playlist functionality isn't available on the device:

```
async private Task CreatePlaylist()
{
  StorageFolder storageFolder = KnownFolders.MusicLibrary;
  StorageFile pureRockFile = await storageFolder.CreateFileAsync("myJam.mp3");
  Windows.Media.Playlists.Playlist myAwesomePlaylist =
    new Windows.Media.Playlists.Playlist();

  myAwesomePlaylist.Files.Add(pureRockFile);

  // Now I'm a safe call! Cache this value if this will be queried a lot
  if (Windows.Foundation.Metadata.ApiInformation.IsTypePresent(
    "Windows.Media.Playlists.Playlist"))
  {
      await myAwesomePlaylist.SaveAsAsync(
        KnownFolders.MusicLibrary, "My Awesome Playlist",
        NameCollisionOption.ReplaceExisting);
  }
}
```

Finally, here's an example of code that looks to access the dedicated camera button found on many mobile devices:

```
// Note: Cache the value instead of querying it more than once
bool isHardwareButtonsAPIPresent =
  Windows.Foundation.Metadata.ApiInformation.IsTypePresent(
  "Windows.Phone.UI.Input.HardwareButtons");

if (isHardwareButtonsAPIPresent)
{
  Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
    HardwareButtons_CameraPressed;
}
```

Note the detection step. If I were to directly reference the HardwareButtons object for the CameraPressed event while on a desktop PC, without checking that HardwareButtons is present, my app would crash.

There's a lot going on around responsive UI and adaptive apps in Windows 10. Do you want to learn more? Check out the great talk about API Contracts that Brent Rector gave at the Build 2015 conference (wndw.ms/IgNy0I), and be sure to view the informative Microsoft Virtual Academy video on adaptive code (bit.ly/1OhZWGs) that covers this topic in more detail. ■

Clint Rutkas *is a senior product manager for Windows focusing on the developer platform. He has worked on Halo at 343 Industries and on Channel 9 at Microsoft, and has built some crazy projects using Windows technology, like a computer-controlled disco dance floor, a custom Ford Mustang, t-shirt-shooting robots and more.*

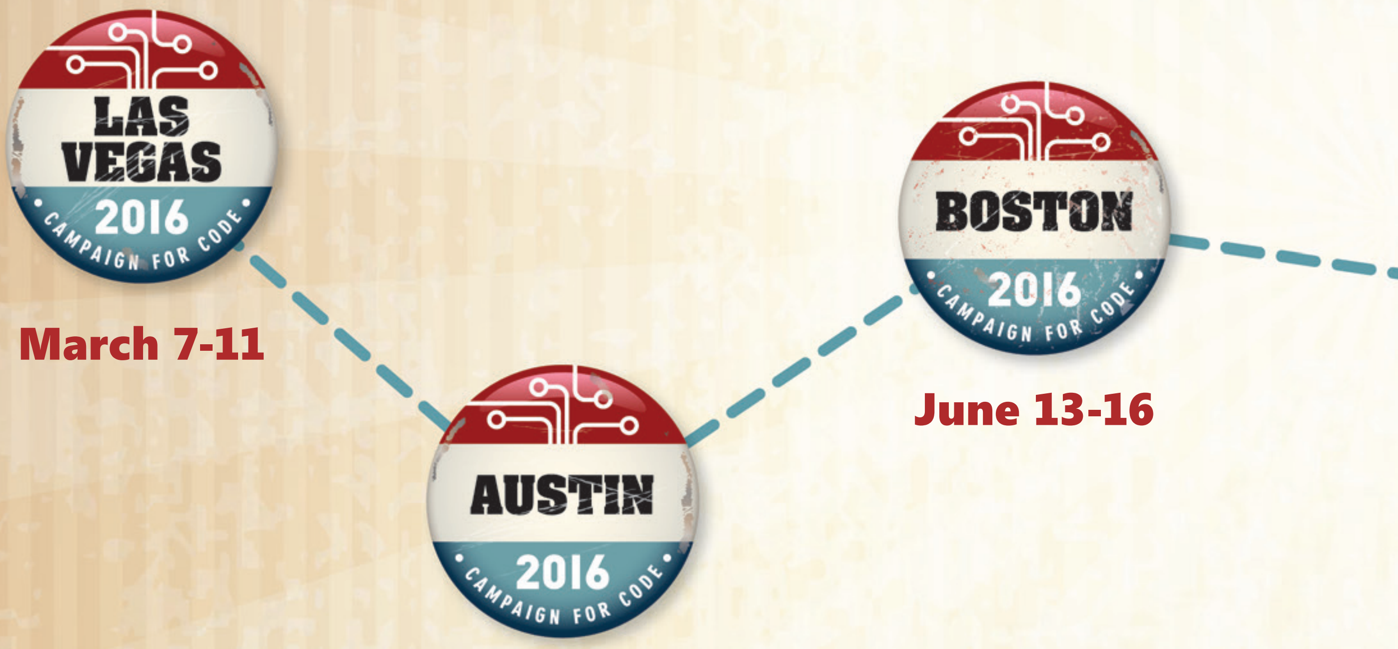
Rajen Kishna *currently works as a senior product marketing manager on the Windows Platform Developer Marketing team for Microsoft in Redmond, Wash. He previously worked as a consultant and technical evangelist for Microsoft in The Netherlands.*

Thanks to the following Microsoft technical experts for reviewing this article: *Sam Jarawan, Harini Kannan and Brent Rector*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

JOIN US



March 7-11

May 16-19

June 13-16

SUPPORTED BY

Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

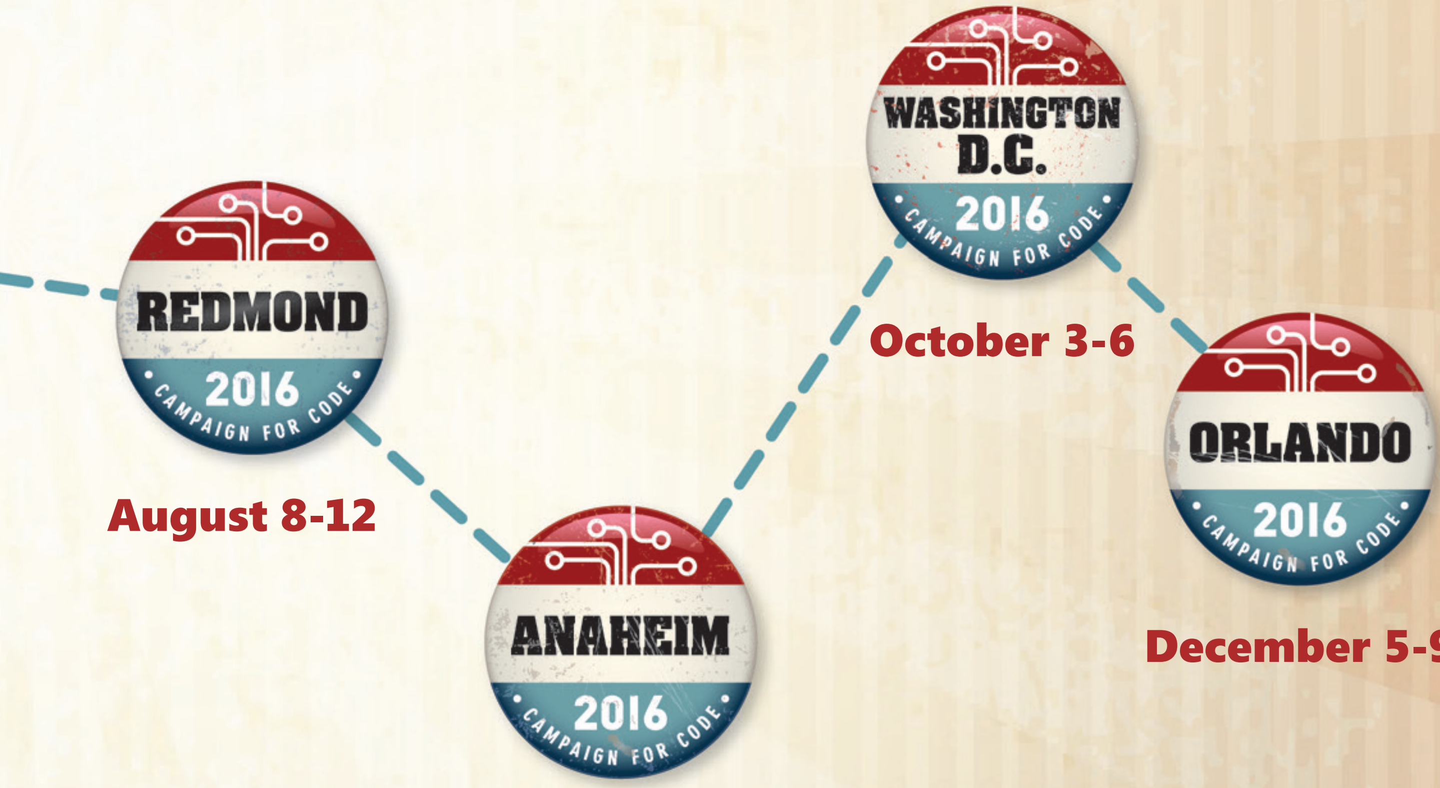
PRODUCED BY

1105 MEDIA INC

YOUR GROWTH. OUR BUSINESS.

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

ON THE 2016 CAMPAIGN FOR CODE TRAIL!



August 8-12

September 26-29

October 3-6

December 5-9

vslive.com

Ink Interaction in Windows 10

Connor Weins

In this article, I'll discuss how you can promote natural user interaction using inking. Digital ink, much like a pen on paper, flows from the tip of your digital pen device, stylus, finger or mouse and is rendered on the screen. To get started with digital ink and inking in Windows 10, I'll start off by addressing a fundamental question: Why is it important for you to use ink in your app?

Humans have been conveying thoughts and ideas through handwriting for centuries. Despite the invention of the mouse and keyboard, the pen-and-paper approach still plays a key role in our lives—from the sticky notes and whiteboards in our offices, to the notebooks in our schools and the coloring books in our children's hands.

Pen-and-paper is immediate, freeform and uniquely personal to each of us, which makes it ideal for people to use to express their creativity and emotion. The act of converting thoughts into text and diagrams into notes also makes handwriting better for thinking, remembering and learning, according to a 2013 study published in *Psychological Science* (bit.ly/1tKDrhv), where researchers from Princeton and UCLA found that handwritten notes were significantly better than typed notes for long-term comprehension.

With these advantages of ink over traditional typed keyboard input, imagine if you could make inking on devices as easy as pen-and-paper and harness the processing power of the computer to do things you can't do in the physical world. With digital ink, you can easily vary the color and appearance of the ink just like in the real world, but take it one step further, analyzing the content and shape of the ink to provide metadata, or converting ink into other content such as text, shapes or commands. This provides a whole new dimension to inking that can't be replicated on your everyday notebook, making digital ink a powerful tool for drawing, note-taking, annotating and interacting in your app. As the pen- and touch-enabled device market continues to expand, inking will become a critical method of interaction for users and app developers.

In Windows 10, it's easy for you to bring digital inking into your app through the DirectInk platform. DirectInk provides a set of rich and extensible Windows Runtime (WinRT) APIs that allow you to collect, render and manage ink in a Universal Windows Platform app. By using DirectInk, you get the same great ink and performance used by the Microsoft Edge browser, Universal OneNote and the Handwriting Panel. Here's a quick overview of the features DirectInk offers your app:

- **Beautiful Ink:** DirectInk uses input smoothing and Bezier rendering algorithms to ensure your ink always looks crisp and beautiful for both touch and pen input.
- **Low Latency, Low Memory:** DirectInk uses a high-priority background thread and input prediction to ensure ink is always immediate and responsive to users, and it manages resources effectively to keep your app's overhead low.
- **Simple and Extensible API Surface:** DirectInk offers APIs such as the InkCanvas and InkPresenter that allow you to quickly get started collecting and managing ink, and they offer advanced functionality that lets you build rich and complex functionality in your app.

This article discusses:

- Collecting ink in applications
- Editing, saving and loading ink
- Advanced inking functionality

Technologies discussed:

DirectInk, Universal Windows Platform Apps

Hopefully by now you're excited to get started using digital ink in your app. I'll now take a look at how you can leverage the DirectInk platform in your app and give users a great inking experience.

Collecting Ink in Your App

To get started using digital ink, the first step is to set up a surface where input can be collected and rendered as ink. In Windows 8.1 Store apps, bringing inking into your app was an extended process that involved creating a Canvas, listening to input events, and creating and rendering strokes piece-by-piece using your own rendering code. For a Universal Windows app, beginning to collect ink is as simple as dropping an InkCanvas into your app:

```
<Grid>
  <InkCanvas x:Name="myInkCanvas"/>
</Grid>
```

As you can see in **Figure 1**, this single line of code gives you a transparent overlay that begins collecting pen input and rendering that input as a black ballpoint pen. The pen's eraser button will also erase any collected ink it comes in contact with. While this is great for getting started with inking, what if you want to change how ink is collected or displayed?

Through the InkCanvas, you can access your InkPresenter, which exposes functionality for controlling the appearance and input configuration of your ink. While pen input provides the best UX for inking, many systems don't come equipped with a pen. The InkPresenter lets you collect ink for any combination of pen, touch and mouse input, and input types that you don't select will simply be delivered as pointer events to the InkCanvas XAML element. Through the InkPresenter, you can also manage the default drawing attributes of ink collected on the InkCanvas, allowing you to change the brush size, color and more. As an example of these features, your app could configure your InkCanvas to collect ink for pen, touch and mouse input, and emulate a calligraphy brush by doing the following:

```
InkPresenter myPresenter = myInkCanvas.InkPresenter;

myPresenter.InputDeviceTypes = Windows.UI.Core.CoreInputDeviceTypes.Pen |
                               Windows.UI.Core.CoreInputDeviceTypes.Touch |
                               Windows.UI.Core.CoreInputDeviceTypes.Mouse;

InkDrawingAttributes myAttributes = myPresenter.CopyDefaultDrawingAttributes();

myAttributes.Color = Windows.UI.Colors.Crimson;
myAttributes.PenTip = PenTipShape.Rectangle;
myAttributes.PenTipTransform =
  System.Numerics.Matrix3x2.CreateRotation((float) Math.PI/4);
myAttributes.Size = new Size(2,6);

myPresenter.UpdateDefaultDrawingAttributes(myAttributes);
```

That would produce the result shown in **Figure 2**.

DirectInk supports many more built-in configurations for input and ink rendering, allowing you to render ink as a highlighter, receive an event when a stroke is collected, access fine-grained input events and ink with multiple pointers in advanced configurations.

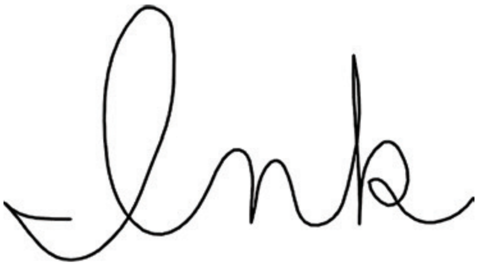


Figure 1 **Using the InkCanvas to Collect Ink Resembling a Black Ballpoint Pen**

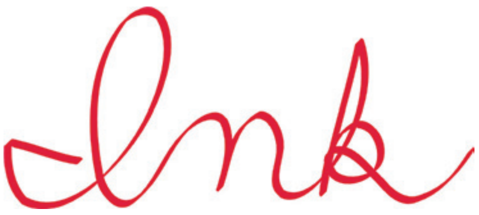


Figure 2 **Emulating a Calligraphy Brush Using the InkPresenter DrawingAttributes**

Editing, Saving and Loading Ink

So now that you've collected some ink, what can you do with it? Users frequently want the ability to erase or edit the ink they collected, or save that ink to access later. In order to provide this experience to your users, you'll have to access and modify the ink data for the strokes that DirectInk has rendered on the screen.

When ink is collected on your InkCanvas, DirectInk stores it within an InkStrokeCollection inside the InkPresenter. This InkStrokeCollection contains WinRT objects representing each of the strokes currently on your canvas, and as changes are made to this container by your app, they'll also be rendered on the screen. This lets you programmatically add, remove or modify strokes, and allows DirectInk to keep you informed of any changes it makes to the strokes on the screen. Let's take a look at some of the common user interactions you can implement using the connection between your InkPresenter and its InkStrokeContainer.

Erasing While the InkCanvas supports erasing with the pen's eraser button by default, it requires some configuration in the InkPresenter to erase ink for mouse and touch input. DirectInk

Figure 3 **Using Unprocessed Input Events To Make a Selection Lasso**

```
...
myInkCanvas.InkPresenter.UnprocessedInput.PointerPressed += StartLasso;
myInkCanvas.InkPresenter.UnprocessedInput.PointerMoved += ContinueLasso;
myInkCanvas.InkPresenter.UnprocessedInput.PointerReleased += CompleteLasso;
...

private void StartLasso(
  InkUnprocessedInput sender,Windows.UI.Core.PointerEventArgs args)
{
  selectionLasso = new Polyline()
  {
    Stroke = new SolidColorBrush(Windows.UI.Colors.Black),
    StrokeThickness = 2,
    StrokeDashArray = new DoubleCollection() { 7, 3},
  };
  selectionLasso.Points.Add(args.CurrentPoint.RawPosition);
  AddSelectionLassoToVisualTree();
}

private void ContinueLasso(
  InkUnprocessedInput sender, Windows.UI.Core.PointerEventArgs args)
{
  selectionLasso.Points.Add(args.CurrentPoint.RawPosition);
}

private void CompleteLasso(
  InkUnprocessedInput sender, Windows.UI.Core.PointerEventArgs args)
{
  selectionLasso.Points.Add(args.CurrentPoint.RawPosition);

  bounds =
    myInkCanvas.InkPresenter.StrokeContainer.SelectWithPolyLine(
    selectionLasso.Points);

  DrawBoundingRect(bounds);
}
```

offers built-in support for erasing ink for any supported input through the Mode property in the InputProcessingConfiguration. Here's an example of a button that sets Erasing mode:

```
private void Eraser_Click(object sender,
  RoutedEventArgs e)
{
  myInkCanvas.InkPresenter.
    InputProcessingConfiguration.Mode =
    InkInputProcessingMode.Erasing;
}
```

When this button is pressed, all input DirectInk collects on the InkCanvas will be treated as an eraser. If the user's input intersects with a stroke after this mode is set, that stroke will be deleted from the InkPresenter's InkStrokeContainer and removed from the screen. When using a pen in Inking mode, the eraser button will always be treated as Erase mode.

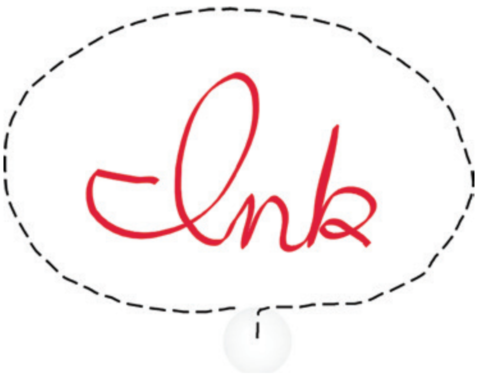


Figure 4 **Selecting Strokes with a Lasso**

Selection Unfortunately, DirectInk doesn't have support for built-in selection at this time, but it does offer a way to develop it yourself through the Unprocessed Input events. Unprocessed events are raised whenever DirectInk receives input that it has been told to listen to, but not to render into ink. This can be done for all input by setting the DirectInk processing configuration mode to None, and can also be configured to happen just for the mouse right button and pen barrel button using the RightDragAction property:

```
myInkCanvas.InkPresenter.InputProcessingConfiguration.RightDragAction =
  InkInputRightDragAction.LeaveUnprocessed;
```

For an example, **Figure 3** shows how you can use Unprocessed Input events to make a selection lasso (as shown in **Figure 4**) to select strokes on the screen.

After you've selected strokes, you can use the InkStrokeContainer.MoveSelected method to translate the strokes, or use the InkStroke.PointTransform property to apply an affine transform to the strokes. When a stroke or set of strokes managed by the InkStrokeContainer is transformed in this way, DirectInk will pick up these changes and render them to the screen.

Saving and Loading DirectInk natively supports ink saving and loading through the Ink Serialized Format (ISF), which saves ink in a vector format that makes it simple for sharing and editing. This is accessible through the InkStrokeContainer SaveAsync and LoadAsync functions.

SaveAsync takes the stroke data currently stored in the InkStrokeContainer and saves it as a GIF file with embedded ISF data. **Figure 5** shows how to save ink from your InkStrokeContainer.

LoadAsync will perform the opposite function, clearing the strokes already in your InkStrokeContainer and loading a new set of strokes from an ISF file or GIF file with embedded ISF data. After strokes are loaded into the InkStrokeContainer, DirectInk will automatically render them onto the screen.

Advanced Inking Functionality

While the ability to edit and manipulate ink on the screen is critical for developing user interaction with ink, it might not be enough to suit all your needs. The more creative the interactions you want your app to support, the more your app will have to break out of the default set of interactions DirectInk provides. Let's dig in to a few of the ways DirectInk enables you to build rich and differentiated inking features:

Inking Recognition Ink is more than just the pixels on the screen. A user's ink can be interpreted as a picture, diagram, shape or text. Recognizing the content of the user's ink allows you to associate the ink with its meaning or exchange the ink with the content it represents. For example, if a user is writing text in a note-taking app, your app can recognize the text the ink represents and use that text data for generating search results when the user enters a query into a search bar. Recognizing text in this way is powered by the InkRecognizerContainer. **Figure 6** shows how to use the InkRecognizerContainer to interpret ink as Simplified Chinese characters.

While this will allow you to recognize ink as text, the InkRecognizerContainer does have a limitation in that it currently supports recognizing text from only 33 different language packs. If you wanted to recognize text from another language, or recognize symbols, shapes or other more abstract ink interpretations, you'd have to build that logic from scratch. Fortunately, the InkStroke object offers the GetInkPoints function, which allows you to get the x/y position of each of the input points used to construct the stroke. From there, you can build an algorithm to analyze the input points of a stroke or set of strokes and interpret them however you want—as symbols, shapes, commands or anything your imagination can think of!

Independent Input DirectInk is a powerful engine for rendering ink that operates under a simple set of rules for input—either render ink for a given stroke, or don't. To make this decision, it looks at the supported input types as well as the input processing configuration provided for the mode and right-drag action configuration. This lacks a lot of the context that your app might want to provide for inking—your app might not allow inking in certain sections of the canvas, or might have a gesture where inking should stop after the gesture is recognized. To enable you to make decisions like this, DirectInk offers access to input before it begins

Figure 5 **Saving Ink from InkStrokeContainer**

```
var savePicker = new FileSavePicker();
savePicker.SuggestedStartLocation =
  Windows.Storage.Pickers.PickerLocationId.PicturesLibrary;
savePicker.FileTypeChoices.Add("Gif with embedded ISF",
  new System.Collections.Generic.List<string> { ".gif" });

StorageFile file = await savePicker.PickSaveFileAsync();
if (null != file)
{
  try
  {
    using (IRandomAccessStream stream =
      await file.OpenAsync(FileAccessMode.ReadWrite))
    {
      await myInkCanvas.InkPresenter.StrokeContainer.SaveAsync(stream);
    }
  }
  catch (Exception ex)
  {
    GenerateErrorMessage();
  }
}
```

processing it through the Independent Input events. These events allow you to inspect the input before DirectInk has rendered it, so if you receive a pressed event in an area where inking isn't allowed or a move event that completes a gesture you've been looking for, you can simply mark the event as Handled.

When the event is marked as Handled, DirectInk will stop processing the stroke, and if a stroke was already in-process, it will be canceled and removed from the screen. You need to be careful when using these events, though. Because they occur on the DirectInk background thread instead of the UI thread, any heavy processing you do in the event or waiting on activities running in a slower thread such as the UI thread can introduce lag that affects the responsiveness of your ink.

Custom Dry One of the most complex features of DirectInk is the Custom Drying mode, which allows your app to render and manage completed or "dry" ink strokes on your own DirectX surface, while letting DirectInk handle performant rendering of the in-progress or "wet" ink strokes. Although DirectInk's default drying mode can handle most scenarios you might want to enable in your app, a few scenarios require you to manage ink independently:

- Interleaving ink and non-ink content (text, shapes) while maintaining z-order
- Performant panning and zooming on a large ink canvas with a large number of ink strokes
- Drying ink synchronously into a DirectX object like a straight line or shape

In Windows 10, Custom Drying mode supports synchronization with a SurfaceImageSource (SIS), or VirtualSurfaceImageSource (VSIS). Both SIS and VSIS provide a DirectX shared surface for your app to draw into and compose, although VSIS provides a virtual surface that's larger than the screen for performant panning and zooming. Because visual updates to these surfaces are synced to the XAML UI thread, when ink is rendered to an SIS or VSIS it can be removed from the DirectInk wet layer simultaneously.

Figure 6 **Interpreting Ink as Simplified Chinese Characters Using the InkRecognizerContainer**

```
async void OnRecognizeAsync(object sender, RoutedEventArgs e)
{
  InkRecognizerContainer recoContainer = new InkRecognizerContainer();
  IReadOnlyList<InkRecognizer> installedRecognizers =
    recoContainer.GetRecognizers();

  foreach (InkRecognizer recognizer in installedRecognizers)
  {
    if (recognizer.Name.Equals("Microsoft 中文(简体)手写识别器"))
    {
      recoContainer.SetDefaultRecognizer(recognizer);
      break;
    }
  }

  var results = await recoContainer.RecognizeAsync(
    myInkCanvas.InkPresenter.StrokeContainer,InkRecognitionTarget.All);

  if (results.Count > 0)
  {
    string str = "Result:";
    foreach (var r in results)
    {
      str += " " + r.GetTextCandidates()[0];
    }
  }
}
```

Custom Drying also supports drying ink to a SwapChainPanel, but doesn't guarantee synchronization. Because SwapChainPanel isn't synchronized with the UI thread, there will be a small overlap between when the ink is rendered to your SwapChainPanel and when ink is removed from the DirectInk wet ink layer.

When you activate Custom Drying, you gain fine-grained control over much of the functionality DirectInk provides by default, allowing you to build logic for how ink is rendered and erased from your dry surface and to determine how ink stroke data is managed by your app. To help you build this functionality, many of the DirectInk components are available as standalone objects to assist your app in filling in the gaps. When Custom Drying is activated, DirectInk provides an InkSynchronizer object, which allows you to begin and end the drying process so ink is removed from the DirectInk wet layer in sync with when you add it to your custom dry layer. The DirectInk default dry ink rendering logic is also available through InkD2DRender to ensure your ink appearance is consistent between the wet and dry layers. For erasing, you can use the Unprocessed Input events to build erase logic similar to the earlier example.

For more info and examples of using Custom Drying, check out the ComplexInk sample available on GitHub at bit.ly/1NkRjt7.

Start Creating with Ink

Using what you've learned so far about the InkCanvas, InkPresenter and InkStrokeContainer, you can now collect ink for different types of input, customize how your ink appears on the screen, access stroke data and have your changes to your stroke data be reflected in the ink strokes rendered by DirectInk. With that simple level of functionality, you can build a wide range of user interactions, from simple doodling to more scenario-focused features such as note-taking and collecting a user's signature. You also have the tools to build more complex interactions through the InkRecognizerContainer, Independent Input events and Custom Drying mode.

With these tools at your disposal, your app should be able to leverage all of the benefits that digital ink can provide to give a great inking experience to your users. As the number of pen- and touch-enabled devices continues to increase, providing a great inking experience to your customers will become even more important for user satisfaction and app differentiation. Hopefully you can take some time to think about how digital inking can work in your app and start experimenting with DirectInk.

As a final note, inking continues to be an important area of investment for Microsoft, and one of the biggest keys to improving and expanding the DirectInk platform for future releases is feedback from our developer community. If you have any questions, comments or ideas while developing with DirectInk, please send them to DirectInk@microsoft.com. ■

Connor Weins *is a program manager working on the Pen, Stylus and Inking team in the Windows Developer Ecosystem Platform group. Reach him at conwei@microsoft.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Krishnan Menon and Xiao Tu*

Write Games for the Universal Windows Platform with Unity

Jaime Rodriguez and Brian Peek

Windows 10 has many new features to offer game developers and gamers. With the Universal Windows Platform (UWP), developers can target Windows 10 tablets and PCs, Windows 10 mobile devices (that is, phones), Xbox One and HoloLens with a single code base. In addition, the Windows Stores have converged into one Store, and Microsoft has brought Xbox Live Services to Windows 10, along with the Xbox app to boost gamer engagement across all the Windows device families.

For game developers, Unity is one of the most popular game development engines. With support for 21 platforms, it has the best cross-platform development support of any middleware available today. Its editor, combined with support for C# scripting, make it an extremely productive game development environment. With the Asset Store, the Unity community, and the growing capabilities of Unity with ads, engagement, and analytics, it has never been easier to develop an immersive game.

When the stars align, we must write about it! This article will give you all the details you need to get your Unity games running great on Windows 10. This isn't an introduction to Unity or an introduction to UWP apps. We assume you're already familiar with both of these topics and, instead, we'll focus on explaining what has changed with Windows 10 and the "must-know" tips required to create great Universal Windows Games (UWGs!). We'll do this in a hands-on approach, walking you through a few changes we made to a sample game called Chomp (see **Figure 1**) to optimize it for Windows 10.

Chomp started life as a Windows 8.1 game written with Unity. As you can see from the screenshot in **Figure 1**, it's a pretty simple maze game similar to Pac-Man. This sample was created as a guide for developers to demonstrate how to write a game with Unity, so simplicity was key. But, with Windows 10 becoming available and Unity supporting the new OS, Chomp had to be updated. You can find the source code for Chomp at bit.ly/ChompUnity. Download it and follow along in our journey.

To get a UWP version of our game, we could simply export the game using the Windows 10 profile in Unity, but this doesn't produce an application optimized for the way Windows 10 works. It doesn't handle running in a window, full-screen mode, touch and so on. So, let's look at what we did to take this game from Windows 8.1 to Windows 10.

This article discusses:

- Setting up your environment
- Targeting Window 10
- New project templates
- Tailoring and optimizations
- Submitting a game to the Windows Store
- New features available for immersive games

Technologies discussed:

Windows 10, Universal Windows Platform, Unity, Visual Studio

Getting Started (Prerequisites)

UWP apps (and UWGs) require that you develop and test with Windows 10 and with Visual Studio 2015. Any Visual Studio 2015 edition will have everything you need to create your games, including Visual Studio Community 2015, which is free!

On the Unity side, you need Unity 5.2.1p2 or higher. Unity 5.2 now installs Visual Studio Community 2015 and Visual Studio 2015 Tools for Unity (VSTU), so, in practice, to get started all you need to install is Unity and check the right options during the install process. See **Figure 2** for details.

Note: For developers using a Mac, a new alternative is to use the Visual Studio Code editor with Unity projects. You can find more details about that option at bit.ly/UnityVSCode.

Figure 1 **Chomp**

Building for Windows 10

Building for Windows 10 follows the exact same process with which you're already familiar. There's a new SDK to target UWP apps under the Windows Store platform (see **Figure 3**), which will export the game as a UWP app.

Here are some of the important items happening under the hood during the export for Windows 10 and UWP development:

- There are new UNITY_UWP and UNITY_WSA_10_0 pre-processors that you can use to "tailor" your game logic and experience for the UWP.
- Unity 5.2 now includes a new streamlined WinRTLegacy DLL that contains fewer types than prior versions. With Windows 10, Microsoft has brought several types back into the .NET Core profile, making some of the workarounds that WinRTLegacy included no longer necessary.
- Unity 5.2 is using the new plug-in model introduced in Unity 5. This greatly simplifies plug-ins that participate in your workflow, as you'll see later in this article.
- Unity 5.2 includes experimental support for DirectX 12, which ships with Windows 10. To try the experimental support, open Player Settings in Unity, uncheck the Auto Graphics API option and manually include Direct3D12 support.

New Project Templates

The Unity build process now generates a Visual Studio 2015 project compatible with the UWP. As you probably already know, there are some significant changes to this new project system.

For example, each UWP app now ships with its own copy of .NET Core within the app, so you always get the .NET version with which you tested. To accommodate this, Unity generates an appropriate project.json file that will pull in the proper .NET "pieces" via NuGet.

Also, UWP apps use .NET Native, which generates optimized, native machine code before your app is downloaded to customer machines and leads to faster launch times and lower battery consumption for your games. The impact of these optimizations will vary based on the complexity of your game, but they definitely won't hurt performance. The only caveat with .NET Native is that it does lead to (significantly) longer compile times within Visual Studio. For regular apps, Visual Studio only compiles with .NET Native for the "Release" configuration of the project. With Unity's generated project file, a similar approach is implemented and only the "Master" configuration is compiled using .NET Native. If you aren't familiar with the configuration targets in Unity, there are three:

- **Debug** is a full debug project with no optimizations.
- **Release** is a project compiled with optimizations, but it includes profiler support.
- **Master** is the configuration you should submit to the Store, as it has no debug code, all optimizations are enabled and profiling support is stripped out.

You definitely should be compiling and testing with .NET Native early in your development cycle. Unity does a lot of advanced intermediate language (IL) weaving, so if there's any type of app that stresses .NET Native to its fullest, it's a Unity game. To ensure things are working as expected, watch out for warnings in the output window during .NET Native compilation.

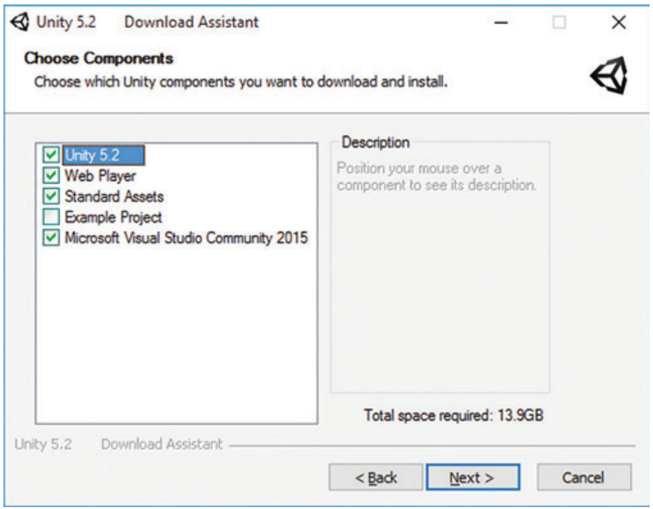
The aforementioned differences are significant at run time, but the new Unity templates make these changes transparent to you as the developer, so let's focus on how to tailor and polish your game for Windows 10.

Tailoring and Polishing Your Game for Windows 10

One code base across many form factors is a key feature of the UWP, but, when it comes to games, there might still be some tailoring and optimizations necessary for specific form-factors. Most often, these include: input mechanisms (such as touch, keyboard, mouse and gamepad); window resizing; optimizing resources and assets; and implementing native platform integration—such as live tiles, notifications or Cortana—with each specific form-factor. We'll explore what these look like for UWGs.

Windowing Universal Windows apps are now hosted in resizable windows instead of running full screen as they did in Windows 8 and 8.1, so windowing is now a consideration for your games and applications. Most of these differences are transparent to you as a Unity developer because the Screen.height and Screen.width properties still report the available space in native (pixel) sizes.

Windows 10 also includes new APIs to enter and exit full screen, and these are exposed via the Unity Screen class by setting the


Unity 5.2 Download Assistant

Choose Components

Choose which Unity components you want to download and install.

- [✓] Unity 5.2
- [✓] Web Player
- [✓] Standard Assets
- [ ] Example Project
- [✓] Microsoft Visual Studio Community 2015

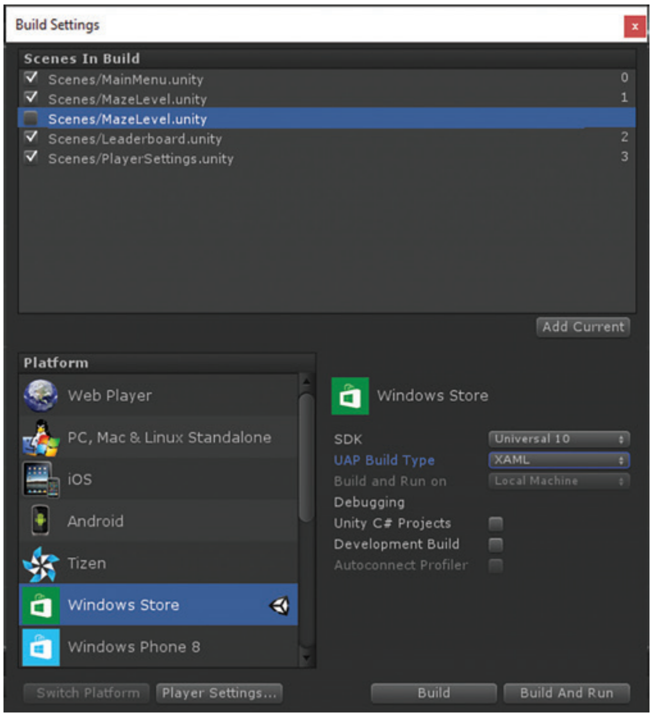
Description

Position your mouse over a component to see its description.

Total space required: 13.9GB

Unity 5.2 Download Assistant

Figure 2 **Installing Unity with the Correct Options Gives You Everything You Need to Get Started**


Build Settings

Scenes In Build

```
✓ Scenes/MainMenu.unity          0
✓ Scenes/MazeLevel.unity         1
  Scenes/MazeLevel.unity
✓ Scenes/Leaderboard.unity       2
✓ Scenes/PlayerSettings.unity    3
```

Windows Store

SDK	Universal 10
UAP Build Type	XAML
Build and Run on	Local Machine
Debugging	
Unity C# Projects	☐
Development Build	☐
Autoconnect Profiler	☐

Figure 3 **Targeting Windows 10 from Unity**

Screen.fullScreen property. A best practice around windowing is to implement the standard key accelerators to enter and exit full screen. These vary widely across publishers, but a common accelerator to toggle in and out of full screen is F11 or Alt+Enter. For Chomp, we wanted to give players this option to play in full screen, so we toggle full-screen mode by doing the following:

```
if (Input.GetKeyUp (KeyCode.F11))
{
  Screen.fullScreen = !Screen.fullScreen;
}
```

Having a multi-window desktop introduces another required change for Windows 10 games: You must handle focus changes. In a multi-window desktop, if your game window doesn't have focus, you should pause your game and your music, as the user might be interacting with a different window. Unity abstracts this interaction with the same API it exposes in other platforms: the OnApplicationPause method. This method is called on all active MonoBehaviours when focus has changed. We handle this in Chomp as shown in **Figure 4**.

Notice there's asymmetry: When a game loses focus, the game and audio are paused, but when it receives focus we un-pause the audio only. This happens because when the game pauses, we also show a paused dialog box in the game itself, and when focus is resumed, the game waits for the user to confirm he wants to resume gameplay. The dialog handles setting the game state back to Running from Pause.

With these two changes in place, we properly handle the ability to enter/exit full screen and pause the game when our window loses focus.

Input Earlier releases of Unity had great support for input in Windows games, and this hasn't changed with Windows 10. Mouse, touch and gamepad remain neatly abstracted by the Input class and the Input Manager in Unity.

The most important thing to remember regarding input is to ensure you're implementing as many input mechanisms as make sense for your games. For Chomp, we want to support keyboard, gamepad and touch. Remember that UWGs can run everywhere, so give your players the best input/gaming experience you can. The most frequent question we see around input is how to detect if you need to show touch controls (like a virtual joystick or D-pad) when you're running in a touch device such as a phone.

One way to determine if the touch joystick should be shown is to determine if the game is running on a phone. If so, it would make sense that the joystick would be displayed and enabled by default. To determine if the game is running on a specific platform (such as a phone or Xbox), you can check if the appropriate contract is implemented. This is how Chomp detects if it's running on Windows 10 Mobile:

```
public static bool IsWindowsMobile
{
  get
  {
    #if UNITY_WSA_10_0 && NETFX_CORE
      return Windows.Foundation.Metadata.ApiInformation.
        IsApiContractPresent ("Windows.Phone.PhoneContract", 1);
    #else
      return false;
    #endif
  }
}
```

Note that in this code we use the UNITY_WSA_10_0 pre-processor to determine if we're compiling for Windows 10. Without this check, the code would fail to compile on non-Windows 10 builds.

Having the virtual joystick visible at all times is one approach, but it might be a better option to only show the joystick when the

Figure 4 **Pausing Your Game When Focus Changes**

```
public void OnApplicationPause(bool pause)
{
  FocusChanged(!pause);
}

public void FocusChanged(bool isFocused)
{
  if (isFocused)
    musicSource.UnPause();
  else
  {
    musicSource.Pause();
    GameState.Instance.SetState(GameState.GameStatus.Paused);
  }
}
```

Figure 5 **Code to Determine if User Is Using Touch**

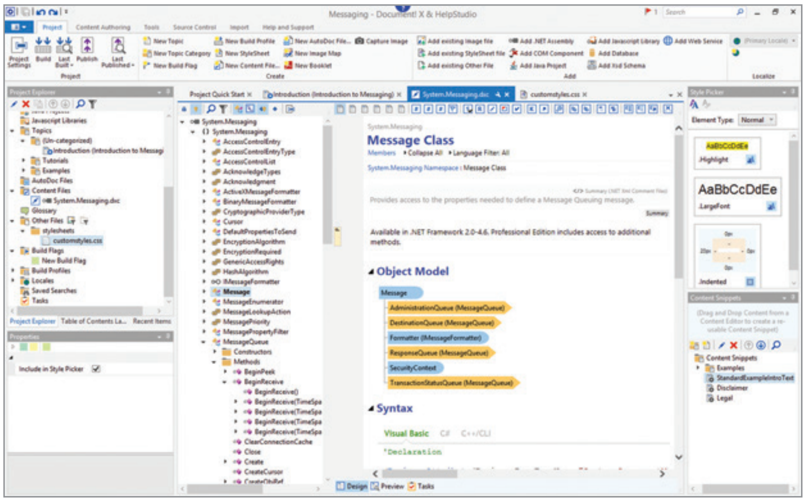
```
public static bool IsWindows10UserInteractionModeTouch
{
  get
  {
    bool isInTouchMode = false;
#if UNITY_WSA_10_0 && NETFX_CORE
    UnityEngine.WSA.Application.InvokeOnUIThread(() =>
    {
      isInTouchMode =
        UIViewSettings.GetForCurrentView().UserInteractionMode ==
        UserInteractionMode.Touch;
    }, true);
#endif
    return isInTouchMode;
  }
}
```

Innovasys

Documentation and Help Authoring Solutions from Innovasys

Q&A with Richard Sloggett, Chief Technical Officer at Innovasys

Innovasys has been a leading provider of Documentation and Help Authoring tools since 1998 and is focused on producing tools that enable developers and technical writers worldwide to produce professional quality documentation, help systems and procedures with minimum friction.



Q What is Document! X?
A Document! X is a documentation and help authoring solution that combines the ability to automatically generate documentation with a full featured and mature content authoring environment. This unique combination delivers all the time and cost benefits of automating as much of the documentation workflow as possible whilst retaining the ability to fully customize and supplement the automatically generated output.

Q What kinds of documentation can Document! X help produce?
A Document! X can document .NET assemblies, Web Services (REST or SOAP), Databases (Access, SQL Server, Oracle, OLE DB), XSD Schemas, JavaScript, COM Components, Type Libraries and Java. In addition to supplementing the automatically generated content, you can also author any number of free format conceptual Topics to round out your documentation.

The same rich and mature authoring environment and fully customizable templates are available for all of the supported documentation types.

Q How does Document! X make documentation less painful?
A With Document! X, documentation can be automatically produced throughout design, development and beyond without requiring investment of developer resources, providing development teams with an accurate and up to date reference and allowing new developers to jump the learning curve of new components, schemas or web services. Document! X makes producing documentation a natural and productive activity for developers and technical writers alike.

Q Can Document! X edit Xml comments in Visual Studio?
A The Document! X Visual Comment editor integrated with Visual Studio provides a WYSIWYG environment in which you can create and edit your Xml source code comments. Content can be authored both in source code comments and in Document! X Content Files outside of the source. This approach makes it easy to make post-development edits and combine the efforts of developers and technical writers.

Q Why choose Document! X?
A Document! X has been trusted by developers and technical writers worldwide since 1998 to produce accurate and comprehensive professional quality documentation. We believe that Document! X is the most comprehensive and mature documentation tool available today, backed with first class support, and we encourage you to try our free trial version to document your own components and see if you agree.

For more information and a free trial version, visit → www.innovasys.com

Figure 6 **Listing Items Available for In-App Purchase**

```
void Start()
{
  Store.LoadListingInformation(OnListingInformationCompleted);
  ...
}

void OnListingInformationCompleted(
  CallbackResponse<ListingInformation> response)
{
  if (response.Status == CallbackStatus.Success)
  {
    ListingInformation result = response.Result;
    PopulatePrices(result.ProductListings);
  }
}
```

user is actually using touch on a device. Windows 10 includes a new API that will determine whether a user is running Windows 10 in Tablet mode (touch) or traditional desktop mode (mouse/keyboard): Windows.UI.ViewManagement.UIViewSettings.UserInteractionMode. This API must run on the Windows UI thread, so from within Unity you must marshal the call to the UI thread. We added the code shown in **Figure 5** to Chomp to determine if the user is interacting with the app using touch.

Now with these two methods implemented, we can update Chomp to make an educated guess on when to show the joystick. If the game is running on a mobile device, or the user interactive mode is touch, UseJoystick will return true, and we display the joystick:

```
public static bool UseJoystick
{
  get
  {
    return (IsWindowsMobile || IsWindows10UserInteractionModeTouch) ;
  }
}
```

With windowing and input handled, we can now move on to take advantage of new native Windows APIs to add more functionality to our game.

Native Integration with Windows 10: Platform Light-Up Native integration with the OS from within a Unity game is handled the same way it was previously: If you're compiling using the .NET Core compilation option in Unity (using the NETFX_CORE pre-processor), you can inline native code, as our previous examples showed.

If the code you want to add is more than you want to inline, or the behavior needs to be abstracted (across platforms), you can still use plug-ins. For Windows 10, the Microsoft Developer Experience Games Evangelism team (that we're a part of) is open sourcing several new plug-ins to make it easier for you to integrate with Windows 10. You can find these plug-ins at bit.ly/Win10UnityPlugins. As of today, the following plug-ins have been shared:

- **Store**: A Store plug-in for in-app purchases with support for app simulator, consumables, durables and receipts. Everything you need to transact with the Windows Store is here.
- **AzureMobile**: A Microsoft Azure plug-in with basic Create, Read, Update, Delete (CRUD) operations for Azure Storage.
- **Core**: A "core" plug-in that offers native integration to core OS features such as live tiles, local notifications, push notifications and Cortana.
- **Ads**: A plug-in that wraps the new Windows 10 Microsoft Ads SDK, now with support for video interstitials.

Previously, Chomp didn't have in-app purchases, so we decided to add them to the game using the Store plug-in. Now, Chomp has support for purchasing consumables (boosters) and durables (mazes).

The easiest way to use these plug-ins is to download the Unity package from GitHub and import it into Chomp using the Assets | Import Package | Custom Packages in Unity Editor. Once the package is imported, you'll have the proper binaries located in the Unity Plugins folder. Once installed, the binaries can be found in the Plugins\WSA directory; Unity requires placeholder assemblies (compatible with the Microsoft .NET Framework 3.5) to be used within the editor, so those are also included and copied to the Plugins folder itself.

After importing the package, we can now reference the Store APIs from within Chomp. We first call Store.LoadListingInformation, providing a callback method that runs upon completion, to get a listing of all the items available for purchase. If the call succeeds, we iterate over the ProductListing items returned and use those to populate the prices in our UI, as shown in **Figure 6**.

Once the user has selected the item he wants to purchase, executing the transaction is just a few more lines of code. **Figure 7** shows the code we added to purchase a durable (new mazes for the game).

As you can see, it's straightforward to use these new plug-ins to implement native functionality.

Submitting to the Store

Submitting to the Windows Store is easier than ever. You can now submit one single package that includes all binaries, or you can submit a package for each platform/architecture, if desired.

If you want to split your packages or support your game only on specific platforms, you can manually edit the package.appxmanifest file and tweak the TargetDeviceFamily element:

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.10240.0"
    MaxVersionTested="10.0.10240.0" />
</Dependencies>
```

The three possible device families are:

- **Windows.Universal**: This allows your binaries to be deployed everywhere that meets your hardware requirements.

Figure 7 **Making an In-App Purchase**

```
public void OnDurablePurchase(GameObject buttonClicked)
{
  string productId = GetProductId(buttonClicked.name);
  if (!string.IsNullOrEmpty (productId))
  {
    Store.RequestProductPurchase(productId, (response) =>
    {
      if (response.Status == CallbackStatus.Success)
      {
        // response.Status just tells us if callback was successful.
        // The CallbackResponse tells us the actual Status
        // as returned from store.
        // Check both.
        if (response.Result.Status == ProductPurchaseStatus.Succeeded)
        {
          EnableLevels(productId);
          return;
        }
      }
    });
  }
}
```

- **Windows.Mobile**: This should be used for binaries that go into the Windows 10 Mobile SKUs, namely Windows Phone, though in the future there will likely be other small devices (six inches or smaller) that aren't phones and run this SKU, so don't assume it's just phones.
- **Windows.Desktop**: This should be used for games that can only be played on desktop and tablets.

If you want to target mobile and desktop, but not console or other later families of Windows, you can have two device families in your manifest, like so (replace "x" and "y" as appropriate):

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Mobile" MinVersion="10.0.x.0"
    MaxVersionTested="10.0.y.0"/>
  <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.x.0"
    MaxVersionTested="10.0.y.0"/>
</Dependencies>
```

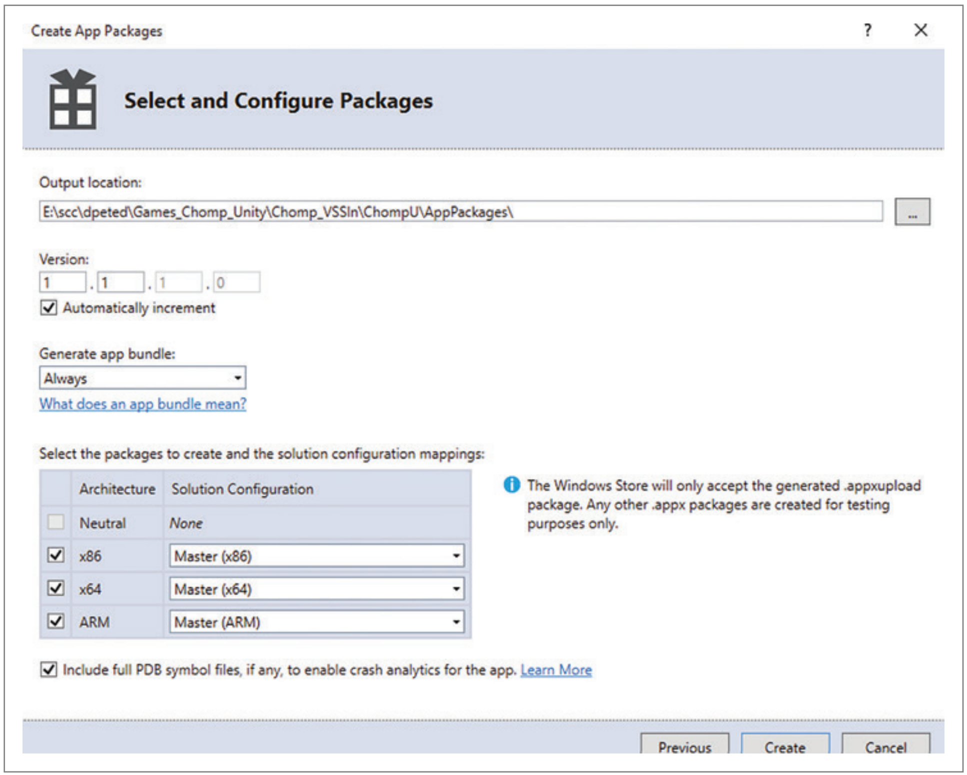
Notice that you can have different MinVersion and MaxVersions for each device family. This will be handy in the future as Windows 10 Mobile will ship with a later version number than Windows 10 desktop. However, for now, we recommend you leave the default versions (10.0.10240.0).

As mentioned earlier when we discussed .NET Native, when submitting your packages, make sure you submit the Master configuration. Also, we recommend you include the full program database (PDB) symbol files for crash analytics. The new Store doesn't have the option to download cabs to analyze them locally. Instead, you must submit your PDBs to the Windows Store, and the Store does the work for you, giving you stack traces for the crashes (see **Figure 8**).

Finally, when you submit to the Windows Store via the developer portal, make sure you choose the right hardware configuration. The Store now lets you specify hardware requirements such as touch or keyboard, so your game is only installed on the right devices (see **Figure 9**).

There's Lots More!

This article gives you a brief "what's new under the hood" for Unity developers by showing how we ported our simple game to Windows 10 and added a few new features. With our limited space, we focused on the foundations and game-specific "must-dos" instead of showing you every new feature in the Windows 10 platform. If you want to have a great, immersive game, consider taking advantage of additional features such as the new notification center (to drive engagement) on desktop, Cortana, the new live tile templates and the new Xbox Live Services APIs. With Windows 10, Microsoft is gradually opening up access to Xbox Live Services. You don't need to write a console game—you just need to have a great game running on Windows 10 and want to take advantage of the services such as leaderboards, achievements, cross-device play and so on. To apply for access, sign up for the ID@Xbox program at xbox.com/id. ■



Output location:
E:\scc\dpeted\Games_Chomp_Unity\Chomp_VSSln\ChompU\AppPackages\

Version:
1 . 1 . 1 . 0
[✓] Automatically increment

Generate app bundle:
Always
What does an app bundle mean?

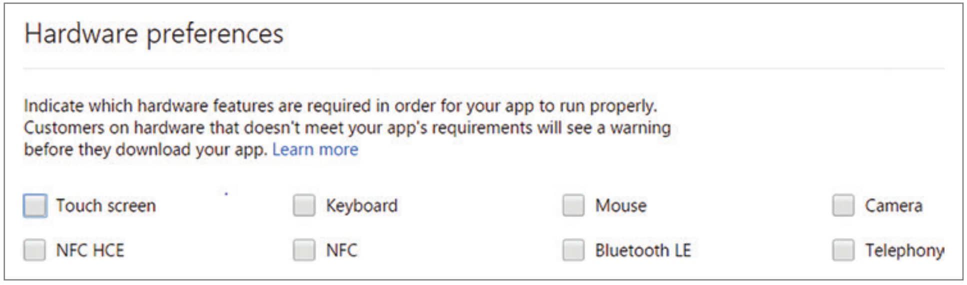
Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
☐	Neutral	None
✓	x86	Master (x86)
✓	x64	Master (x64)
✓	ARM	Master (ARM)

The Windows Store will only accept the generated .appxupload package. Any other .appx packages are created for testing purposes only.

[✓] Include full PDB symbol files, if any, to enable crash analytics for the app. Learn More

Figure 8 **Including Program Database Symbol Files**



Indicate which hardware features are required in order for your app to run properly. Customers on hardware that doesn't meet your app's requirements will see a warning before they download your app. Learn more

☐ Touch screen ☐ Keyboard ☐ Mouse ☐ Camera
☐ NFC HCE ☐ NFC ☐ Bluetooth LE ☐ Telephony

Figure 9 **Hardware Preferences Available in Developer Portal**

Jaime Rodriguez *leads the Microsoft Developer Experience Games Evangelism team. You can find him on Twitter: @jaimerodriguez and via his blog at jaimerodriguez.com.*

Brian Peek *is a senior game developer evangelist at Microsoft. He's a hardcore gamer and has been developing games on Windows, consoles and anything else that can be programmed for as long as he can remember. You can spot him around the country speaking at developer conferences, or follow him on Twitter: @BrianPeek and via his blog at brianpeek.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Sanjeev Dwivedi and Adam Tuliper*

Welcome to Windows 10 App Development

I hope you are enjoying this special edition of *MSDN Magazine* focusing on app development for the Universal Windows Platform (UWP). I'm Kevin Gallo, vice president of the Developer Platform team within Windows. My team is responsible for creating the UWP, and we're anxious for you to use it to deliver great UXes and to give us feedback to help us make it better in the future.

One of the best and most satisfying things about creating a platform like the UWP is getting to see the amazing things that people do with it. Your apps and games bring the platform to life and create the experiences that attract, engage and delight users. We mean it when we say that we love developers and their awesome creations, and we look forward to seeing what that you'll come up with next.

With Windows 10 and the UWP, it's possible for you to target—with a single app—the wide range of Windows devices. This is possible because the platform has one shared set of APIs, a single set of languages and frameworks, one set of tools, and a unified Windows Store.

This means that your apps can reach a huge audience. That audience includes users of Windows 10 PCs—of which there are nearly 100 million and counting—Internet-of-Things (IoT) devices, Windows Phone handsets (coming later in 2015), as well as the Surface Hub, Xbox and HoloLens (coming in 2016) platforms.

With Windows 10 and the UWP, it's possible for you to target—with a single app—the wide range of Windows devices.

With the UWP, it's easy to use our adaptive controls to make apps look great across a wide range of device types, input methods and screen sizes. You can also tailor your apps to create customized experiences for specific devices, ranging from the smallest IoT sensor to the largest Surface Hub interactive whiteboard and display. That means you can use a single UWP app to target this diverse audience with experiences that are tailored to each device and usage scenario in ways that are the most natural and comfortable for users.

This issue of *MSDN Magazine* provides an introduction and a practical guide to many of the new and exciting capabilities of the UWP, ranging from design guidance for multiple screens and optimizing communication between apps, to improvements in background processing and multitasking, and more.

For those of you who already have apps on Windows 8.1 or Windows Phone 8.1, I want to emphasize that the UWP is just an evolution of the same modern platform you've been using. You can continue to write apps for the Windows platform in your language and using the framework of your choice, whether that's C/C++, C#, Visual Basic, JavaScript, XAML or Silverlight. You can reuse significant portions of your existing code with minimal effort and then focus your time enhancing the app experience with the unique features of Windows 10.

Beyond this special issue of *MSDN Magazine*, we offer a trove of online content that we think will be useful to you as a UWP app developer. Here are just a few of the resources you might be interested in checking out:

- **Building Apps for Windows:** Our developer blog, with new content added every week (bit.ly/1KZUio1)
- **Build 2015:** Session videos and slides from the conference are on the Channel 9 site (bit.ly/1NHlnz7)
- **A Developer's Guide to Windows 10:** An 18-part video series from Microsoft Virtual Academy that teaches with practical examples (bit.ly/1LrDVmM)
- **Windows Dev Center:** Downloadable tools, design guidance, how-to guides, plus complete API reference documentation
  - Dev Center (dev.windows.com)
  - Tools Download (bit.ly/1MVjblj)
  - Design Documentation (bit.ly/1LC6eej)
  - How-to Guides (bit.ly/1WdJkkG)
- **Code and App Samples:** Available in a GitHub repository (bit.ly/1RhG46l)

Are you planning to work with the UWP and Windows 10? If so, we'd love to hear your feedback. We rely on developer input to help shape our plans and priorities. Make your voice heard by providing feedback via our Windows Developer User Voice. Go to wpdev.uservoice.com and leave us a suggestion, or vote on the existing suggestions that are important and relevant to you.

I hope you find this issue of *MSDN Magazine* useful. I can't wait to see the great things you and the Windows developer community will build with the Universal Windows Platform! ■

KEVIN GALLO *is corporate vice president of the Windows Developer Platform at Microsoft and is responsible for designing and shipping the platform components through which developers build meaningful apps and services for the Windows family of devices. Gallo holds a Bachelor of Engineering from Carnegie Mellon University in Computer Engineering. In his spare time he enjoys biking, hiking and boating with his family.*

```
// Create an OCR Engine for each processor
on the machine. This
// allows for optimal use of thread during
recognition and processing.
ocrEngines = new List<IOcrEngine>();
for (int i = 0; i <
Environment.ProcessorCount; i++)
{

ocrEngines.Add(OcrEngineManager.CreateEngin
e(
```

MILLIONS OF LINES OF CODE AT YOUR FINGERTIPS

Document Imaging	**Medical** Imaging	**Multimedia** Imaging
Document Viewer	*DICOM, CCOW & HL7*	*Play, Capture, Convert & DVR*
Document Converter	*PACS Client & Server Framework*	*Stream - MPEG2-TS & RTSP*
OCR, MICR, OMR & ICR	*DICOM Storage Server Framework*	*Media Streaming Server*
Barcode & Forms Recognition	*Web & Desktop Viewers*	*MPEG-4, H.264 & H.265*
Create, Save, Edit & View PDF	*Image Processing & Annotations*	*Media Foundation & DirectShow*
Annotations, TWAIN & SVG	*Medical 3D (MPR, MIP, VRT)*	*Distributed Transcoding*

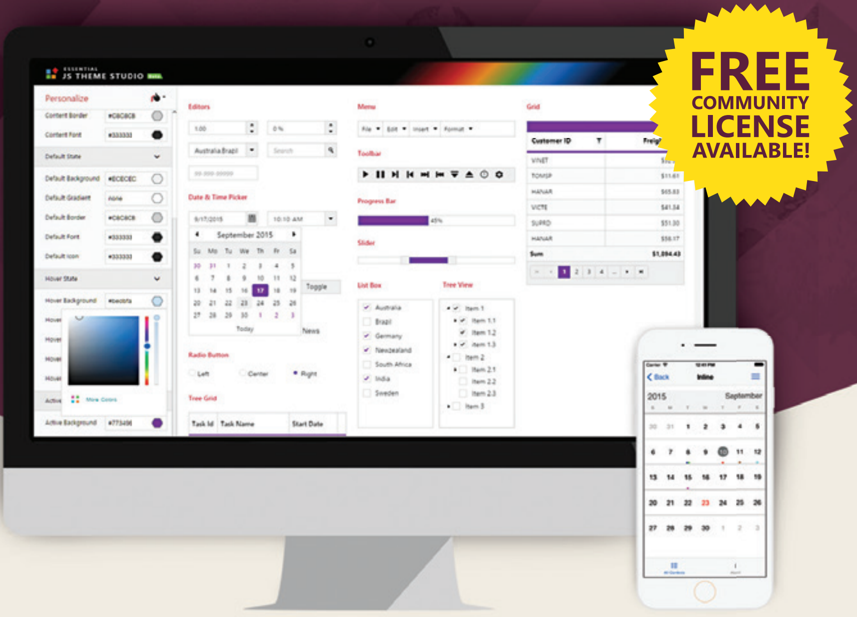
.NET   Windows API   WinRT   Linux   iOS   OS X   Android   HTML5

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

MORE THAN **650** CUSTOMIZABLE
CONTROLS AND FRAMEWORKS



NOW WITH WINDOWS 10 COMPATIBILITY ON APPLICABLE PLATFORMS!

WEB

ASP.NET MVC

ASP.NET Web Forms

HTML5/JavaScript

LightSwitch

Silverlight

MOBILE

Android

HTML5/JavaScript

iOS

Orubase

Universal Windows Platform

Windows Phone

WinRT

Xamarin

DESKTOP

Windows Forms

WPF

Universal Windows Platform

FILE FORMATS

Excel

PDF

Word

PowerPoint

DATA SCIENCE

Big Data Platform

Predictive Analytics

ENTERPRISE SOLUTIONS

Report Server

DOWNLOAD A FREE TRIAL AT

syncfusion.com/msdnevaluation

Syncfusion®