THE MICROSOFT JOURNAL FOR DEVELOPERS

DECEMBER 15, 2014 VOL 29 NO 12A

msdn magazine

Connect(); Special Issue

THE MICROSOFT JOURNAL FOR DEVELOPERS

DECEMBER 15, 2014 VOL 29 NO 12A

msdn magazine

Connect(); Special Issue

Visual Studio 2015 & Microsoft Azure

Microsoft

Version 19 has been crafted with developers in mind. It has been optimized to give you an all-in-one development solution for all your document, medical and multimedia needs.

Document
Imaging SDKs

- *Document Viewer* New
- *Document Converter* New
- *OCR, MICR, OMR & ICR*
- *Barcode & Forms Recognition*
- *Create, Load, Save, Edit & View PDF*
- *Annotations, TWAIN & SVG* New

Medical
Imaging SDKs

- *DICOM, CCOW & HL7* New
- *PACS Client & Server Framework*
- *DICOM Storage Server Framework*
- *Web & Desktop Viewers*
- *Image Processing & Annotations*
- *Medical 3D (MPR, MIP, VRT)*

Multimedia
Imaging SDKs

- *Playback, Capture & Convert*
- *Streaming - MPEG2-TS & RTSP*
- *MPEG-4, H.264 & H.265* New
- *DVD & DVR*
- *Media Foundation & DirectShow*
- *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

msdn magazine

DECEMBER 15, 2014 VOLUME 29 NUMBER 12A

MSDN Magazine (ISSN 1528-4859) is published 13 times a year, monthly with a special issue in December 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, Inc., 9201 Oakdale Ave., Ste 101, Chatsworth, CA 91311, www.1105media.com

Media Kits: Direct your Media Kit requests to Matt Morollo, VP Publishing, 508-532-1418 (phone), 508-875-6622 (fax), mmorollo@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

All customer service inquiries should be sent to MSDNmag@1105service.com or call 847-763-9560.

Microsoft

BPA WORLDWIDE

Printed in the USA

Editor's Note

MICHAEL DESMOND

Building on Connect();

Back in mid-November, before the holiday rush made mincemeat of everyone's work schedules, Microsoft did a rather remarkable thing. A bunch of things, actually.

At the Connect(); event in New York City, the company rolled out a series of products and initiatives that portend the next generation in Microsoft-borne software development, and point to a world that's becoming significantly more open, mobile and cross-platform than before. From the release of Preview editions of Visual Studio 2015 and .NET 2015, to the open sourcing of the .NET Core Runtime and Base Class Libraries, the message is clear: Microsoft aims to equip developers with the best tools and solutions, whether they're writing code in an enterprise .NET shop or banging out mobile games in a home office.

In this special issue of *MSDN Magazine*, you'll have an opportunity to explore the deep and broad innovation coming out of the Visual Studio, Microsoft Azure, and other teams at Microsoft. You'll learn about the latest versions of Visual Studio, the .NET Framework and languages such as C# and C++. You'll be introduced to the exciting realm of Visual Studio Extensions, which enable developers to customize and tune the IDE—including the new, freely available Community Edition. And you'll gain a foothold in powerful, cross-platform development frameworks, such as Xamarin, Apache Cordova and Unity.

Microsoft likes to say that it's a "mobile-first, cloud-first world," and the events at the Connect(); conference put emphatic action to those words.

"Announcements of this magnitude don't happen very often, and if this doesn't get you excited as a developer using Microsoft platforms, tools and services, I'm not sure what will," says Microsoft Principal Director Keith Boyd.

Visual Studio 2015 is at the heart of the evolving developer experience, and this special issue directly addresses that. Cross-platform-themed articles like Adam Tuliper's dive into Unity game development and Kraig Brockschmidt and Mike Jones' look at building hybrid apps for Apache Cordova show how Visual Studio integration enables developers to reach new markets. Doug Erickson and Susan Norwood explore Visual Studio Extensions, while Alex Turner dives into the .NET Compiler Platform (aka, "Roslyn") and the new diagnostic analyzer functionality in Visual Studio 2015. There's plenty of Visual Studio insight, as well, in articles that explore the latest versions of C# and C++.

Microsoft Azure is also on full display in this issue. Mohit Srivastava and Saurabh Bhatia offer an in-depth look at the new Azure SDK for .NET, which enables .NET-based cloud applications to consume Azure resources. And Glenn Gailey explores how developers can leverage the Azure platform to send push notifications to hybrid Cordova apps.

There's a lot more going on here than I can explain in one page, and even more than we can publish in a single issue. Just days after this special issue reaches readers, you can expect to receive our regular January issue. In it you'll find additional coverage of the new tools and technologies debuted by Microsoft in November, including the latest on Visual Basic 14, an exploration of new Visual Studio tooling for HDInsight, and a glimpse at how Test Hubs enable management of manual tests with Visual Studio Online and Team Foundation Server.

Dmitry Lyalin, senior product manager for Visual Studio and Azure, has been instrumental in putting this special issue together. He describes the Connect(); event as "an extremely important developer moment," and one that will echo forward in the coming years.

"One would think that releasing Visual Studio 2015 Preview, a Microsoft-built Android emulator, Visual Studio 2013 Update 4 and Azure SDK 2.5 would be enough. But we also made two huge announcements, open sourcing the full, server-side .NET Core stack and shipping a free version of Visual Studio called Visual Studio Community 2013," he says. "It's just an amazing time to be a developer on the Microsoft stack."

Indeed it is.

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

GROUPDOCS
Your Document Collaboration APIs

30 days free trial

Your Document Collaboration APIs

Professional APIs that allow developers to empower their apps with document collaboration capabilities.

GroupDocs.Viewer

Native-text, high-fidelity HTML5 document viewer with support for over 50 file formats.

GroupDocs.Signature

Electronic signature API that allows users to place signatures on documents within your apps.

GroupDocs.Annotation

A powerful API that lets developers annotate Microsoft Office, PDF and other documents within their apps.

GroupDocs.Comparison

A diff view API that allows end users to quickly find differences between two revisions of a document.

GroupDocs.Conversion

Universal document converter for fast conversion between more than 50 file formats.

GroupDocs.Assembly

Merges data entered by users through online forms Into both Microsoft Office and PDF documents.

100% Standalone - No Office Automation

.NET Libraries | Java Libraries | Cloud APIs | Cloud Apps

SALES INQUIRIES
+1 214 329 9760
sales@groupdocs.com
groupdocs.com

Write Cross-Platform Hybrid Apps in Visual Studio with Apache Cordova

Kraig Brockschmidt and Mike Jones

Web developers with expertise in standard HTML, CSS and JavaScript already know that Web applications reach the greatest breadth of devices and customers with a single code base. And with responsive Web design, Web applications can also deliver great mobile experiences.

Yet the wind appears to be blowing elsewhere: Data shows that mobile customers spend about 80 percent of their time in apps and 20 percent in browsers. Put simply, even the best mobile Web solutions just don't get the visibility and usage of native apps that are discoverable through platform app stores. Furthermore, although the W3C and browser vendors strive to surface native platform capabilities through new HTML5 standards, it's a slow process. Compared with native apps and native APIs, the mobile Web and HTML5 will generally be at a disadvantage when it comes to delighting customers with new innovations.

This article discusses a prerelease version of Visual Studio Tools for Apache Cordova. All related information is subject to change.

This article discusses:

- Using responsive cross-browser CSS
- Structure of a Cordova project
- Accessing native platform APIs
- Implementing geolocation
- Working with the Cordova camera plug-in
- App project lifecycle options
- Future improvements in Visual Studio

Technologies discussed:

Visual Studio Tools for Apache Cordova, HTML5, CSS, JavaScript

Code download available at:

bit.ly/1yo8Hr3

> The Cordova plug-in model provides full access to native platform APIs.

Must you then bite the bullet and write native apps? Must you learn different languages and UI paradigms, shoulder the high cost of managing projects for each platform, and subsequently miss potential market opportunities?

Not at all! Enter the open source framework known as Apache Cordova (`cordova.apache.org`), managed by the Apache Software Foundation (ASF). Cordova lets you use your HTML, CSS and JavaScript skills—and vast community resources—to create apps for Android, Amazon Fire, iOS, Mac OS X, Windows, Windows Phone, BlackBerry, Web browsers and more. Better still, whereas Web

applications have limited browser-supported capabilities only, the Cordova plug-in model provides full access to native platform APIs.

Thus, Cordova gives you the cross-platform benefits of the Web without sacrificing functionality, offering an attractive path for entering the mobile app marketplace. We'll show how it all works by building the simple selfie app shown in **Figure 1**.

Cordova in Visual Studio

Working with Cordova is typically done through the Node Package Manager (npm) and the Cordova command-line interface (CLI). Depending on your target platform, you also need to install other components such as platform SDKs as described in the "Platform Guides" section of the Apache Cordova Documentation (bit.ly/1tU19fq). For many developers wanting to target Android, iOS and Windows, this can be a lot of work, and it can take some time to learn all the intricacies of the different tools.

To make it easier and to bring the power and productivity of Visual Studio to Cordova, Microsoft is building the Visual Studio Tools for Apache Cordova. This toolkit—presently in preview and available as an extension for Visual Studio 2013 Update 4 (bit.ly/11m4vKH) and built into Visual Studio 2015 Preview (bit.ly/1u1dsSQ)—correctly installs and configures each third-party dependency for Android and iOS targets (Windows comes automatically with Visual Studio). It then automates Cordova processes into the Visual Studio IDE, such as installing plug-ins, running builds, and deploying to devices and emulators. Furthermore, it gives you all the goodness you expect from Visual Studio such as IntelliSense, integrated debugging, project and configuration management, integration with Visual Studio Online, and the ability to use TypeScript.

To target Android and Windows using Visual Studio, you need only a Windows 8 or higher machine (and you can do Android from Windows 7). To target iOS, you need a Mac with Xcode 5.1 or later, and you can run Visual Studio within Windows on Parallels or VMware Fusion. (See the "Install Tools to Build for iOS" MSDN Library page at bit.ly/1GqnEuK for details, including information about the remote build agent, vs-mda-remote, which runs on Macs and lets you build, run and debug from Visual Studio over a local network or VPN connection.)

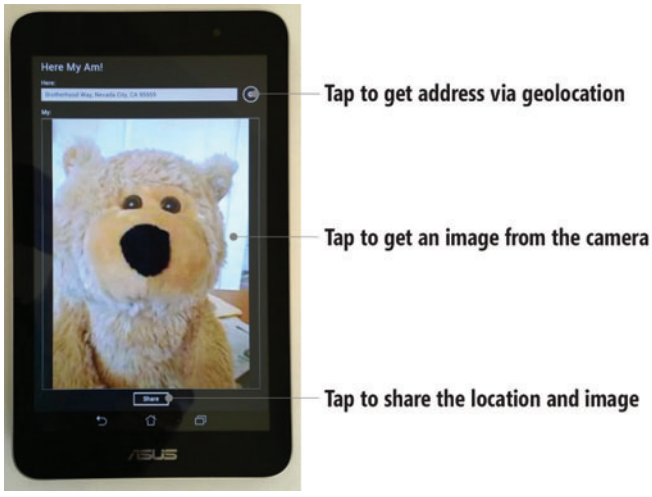


Figure 1 **The Here My Am! App on an Android Device**

Figure 2 **The HTML Markup for the Here My Am! App**

```
<body>
  <div class="mainContainer">
    <div class="title mainChild">
      <h1 class="titlearea ">Here My Am!</h1>
    </div>
    <div id="locationSection" class="mainChild subsection">
      <h2 class="section-title">Here:</h2>
        <div id="locationControls">
          <input id="txtLocation" type="text" placeholder="tap to edit" />
            <button id="btnLocate" class="iconGlobe"></button>
        </div>
    </div>
    <div id="photoSection" class="mainChild subsection">
      <h2 id="photoHeading" class="section-title">My:</h2>
        <div id="photo">
          <img id="photoImg" src="#" draggable="false"
          alt="Tap to capture image from camera" />
        </div>
    </div>
    <div id="shareSection" class="mainChild">
      <button id="btnShare" class="shareButton" disabled>Share</button>
    </div>
  </div>
</body>
```

Of Web Hosts and Responsive Cross-Browser CSS

We'll call our selfie app Here My Am! (a phrase that Kraig's son used at age 3), whose HTML5 markup is straightforward, as shown in **Figure 2**.

Next, we need the appropriate styling to match **Figure 1**, which gives us an opportunity to explore the core nature of Cordova.

> To bring the power and productivity of Visual Studio to Cordova, Microsoft is building the Visual Studio Tools for Apache Cordova.

Native mobile apps generally run from compiled executables: The code you write in languages such as Objective-C, Swift, Java, C# and C++ is compiled into binary formats that mobile OSes understand. HTML, CSS and JavaScript, on the other hand, must be fed to a *host*—such as a browser—that parses, renders and executes that source code at run time. Some platforms, such as Windows 8/8.1 and Windows Phone 8.1, provide a system-level host that renders HTML, CSS and JavaScript directly as a native app. For more on this, refer to the free ebook, "Programming Windows Store Apps with HTML, CSS, and JavaScript, Second Edition" (Microsoft Press, 2014), by Kraig Brockschmidt at bit.ly/1jyOYLC. Cordova uses such a capability where possible. Otherwise, as on iOS and Android at present, it uses a browser-like WebView element inside a native app wrapper, as shown in **Figure 3**. Either way, you have an environment in which your Web-standards code runs in the context of an app.

The rub is that these host environments aren't created equal. Each is a variant of the native browser in the OS, as shown in **Figure 4**.

Therefore, *a Cordova app's CSS must work in each target platform's Web host*, just like Web applications must work in different browsers by using a variety of prefixed and nonprefixed styles, and you can use similar approaches. It's helpful, for example, to drop your app's files on a Web host and open them up in Internet Explorer, Chrome and Safari (on a Mac) and then use browser developer tools to test your styling at different window sizes. Also, cross-reference your styling at caniuse.com, which identifies exactly which HTML5 and CSS3 features are supported in which browsers and mobile environments (and remember to look at the Android Browser and iOS Safari, specifically). Of course, simpler layouts will be more adaptable, and you can use CSS frameworks such as jQuery Mobile, Ionic and Bootstrap. You can also use CSS preprocessors such as LESS or SASS, which are supported as of Visual Studio 2013 Update 2, but need to be run prior to a Cordova build.

Along similar lines, mobile apps will also encounter many combinations of screen sizes and orientations, as described in **Figure 5**.

All the principles of responsive Web design—using media queries to adjust margins, font sizes, layouts, animations and so on—are thus essential with Cordova apps (and to keep things simple, the Here My Am! app supports portrait mode only). Be mindful, too, that the physical screen dimensions of a device aren't necessarily those reported to the CSS rendering engine. High-DPI displays typically have a scaling factor applied, meaning the sizes in **Figure 5** apply to *[min | max]-[width | height]* media queries rather than *[min | max]-device-[width | height]* queries. In some cases the scaling might not be exact, either; a nominal 768-px dimension on an iPad might appear in CSS as 767. Test early and often, and again consider a CSS framework or preprocessing tool.

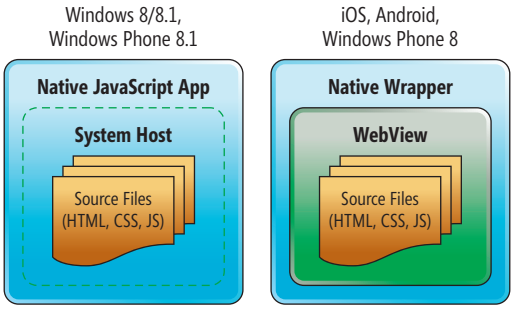


Figure 3 **Cordova Runs HTML, CSS and JavaScript Within a Native App Environment**

Figure 4 **Basis for Host Environments**

Mobile Platform	Host Basis
Windows 8/8.1, Windows Phone 8/8.1	Internet Explorer 10/11
Android 4.x, 5.0	Android Browser 4.x, 5.0 (4.4 or later is recommended for the best implementation)
iOS7, iOS8	iOS Safari 7.1/8 or later, depending on updates

Figure 5 **Common Screen Size Combinations**

Portrait Widths/ Landscape Heights	Portrait Heights/ Landscape Widths
240 (low-end phones)	320, 400, 480
320 (mid-level phones)	462, 480, 470, 568 and greater heights on Windows in a narrow view.
480 (mid-level phones)	640, 800, 854 and other heights on Windows in a greater-than-500-px portrait orientation.
720, 768 (tablets)	960, 1024 or larger. The iPad is 1024x768, as are some views on Windows.
Greater than 768	iPad 2 and later; most Windows laptops and tablets and many Android tablets.

The Cordova App Project Structure

After our markup and CSS are in decent shape, we can bring them into a Cordova app. In Visual Studio, a Cordova app created with File | New | Project, selecting JavaScript | Apache Cordova Apps | Blank App (Apache Cordova), gives the project structure shown in **Figure 6** (note that this is subject to change as the Tools for Cordova approach RTM).

For our app, we'll drop the markup into index.html and our styles into css/index.css. We'll also bring in the cross-platform WinJS 3.0 library for various goodies, which we've copied into css/frameworks, css/fonts and scripts/frameworks to keep everything organized (and of course you can use any number of other libraries such as AngularJS and Bootstrap). Here's the <head> element of index.html:

```
<!-- Recommended to prevent pinch zoom in webviews -->
<meta name="viewport" content="user-scalable=no" />

<link href="css/frameworks/ui-dark.css" rel="stylesheet" /> <!-- WinJS -->
<link href="css/index.css" rel="stylesheet" /> <!-- App styles -->

<script src="cordova.js"></script> <!-- Cordova standard -->

<!-- WinJS -->
<script src="scripts/frameworks/base.js"></script>
<script src="scripts/frameworks/ui.js"></script>
<script src="scripts/frameworks/winjs.js"></script>

<!-- Project references -->
<script src="scripts/platformOverrides.js"></script>
<script src="scripts/index.js"></script>
```

Note the reference to cordova.js, which you won't find anywhere in your project; it's brought in automatically as part of the build process.

As for platformOverrides.js, the file by this name in the root scripts folder is empty, but there's another in merges/windows/scripts that loads up another file called winstore-jscompat.js (to help accommodate third-party libraries). At build time, all files and folders under merges/<platform> are copied into the final folder structure, overwriting any files with the same names in the root project. On Windows, scripts/platformOverrides.js is replaced by merges/windows/scripts/platformoverrides.js. This is how you can incorporate platform-specific code, if needed, and the same file-level merging also works for CSS, images and anything else. Just remember that it's a file-level merge, not a merging of file contents! Also note that the merges folder might be deprecated in favor of feature-detection approaches.

The contents of the res folder have a similar but more varied behavior because these resources are needed for the app package, rather than the code running inside the WebView or host. Remember that Cordova builds a native app wrapper around a WebView when necessary, meaning that it creates a folder structure for the native app package as required by the target platform. **Figure 7**

illustrates where different parts of the Visual Studio project end up in a build. You can see the results if you build a project and then open the bld/debug/platforms folder. We will use this behavior later on to customize the Windows/Windows Phone app manifests.

Native APIs and Plug-Ins

We're now ready to add behaviors in JavaScript to our app. By default, the Cordova app template in Visual Studio supplies the code shown in **Figure 8** (most comments omitted), where the Cordova deviceready, pause and resume events abstract similar app lifecycle events on various platforms (see the "Events" section of the Apache Cordova Documentation at bit.ly/1u1hT1n for the full roster).

For simplicity, we won't be persisting any data in Here My Am!, so we've removed the pause and resume handlers.

Note that deviceready typically fires after DOMContentLoaded, so you can do DOM-related initialization within your deviceready handler. In some debugging scenarios, however, such as using Visual Studio remote debugging over a wireless network, it's possible for deviceready to fire before DOMContentLoaded, leading to null object exceptions. Here My Am! uses some flags to handle this variance.

Here's what we now need to do in our app:

- Initialize the Here field with the user's location and wire up the locate button.
- Initialize the photo area with the default message and wire it up to camera capture.
- Enable sharing the location and image.
- Lock the display orientation to portrait.

Figure 6 **Project Structure in Visual Studio**

Folder or file	Can rename?	Description
css	Yes	Folder for stylesheets. The css, images and scripts folders can all be renamed arbitrarily and referenced from your HTML files.
images	Yes	Default (empty) folder for platform-neutral graphics.
merges	No	Folder for platform-specific JavaScript. The build tools depend on this folder name, as with res.
res	No	Folder for platform-specific resources such as icons, splash screens and manifests.
scripts	Yes	Default folder for JavaScript files.
config.xml	No	The global configuration file that conforms to the W3C Packaged Web Apps (Widgets) specification. It serves as a generic manifest whose settings translate into platform-specific manifest entries. Visual Studio provides a designer for config.xml, and you can edit the raw XML, as some settings aren't shown in the designer.
index.html	Yes	Default start page for the app, referred to in the Common | Start Page field of config.xml. If you rename it, update config.xml accordingly.

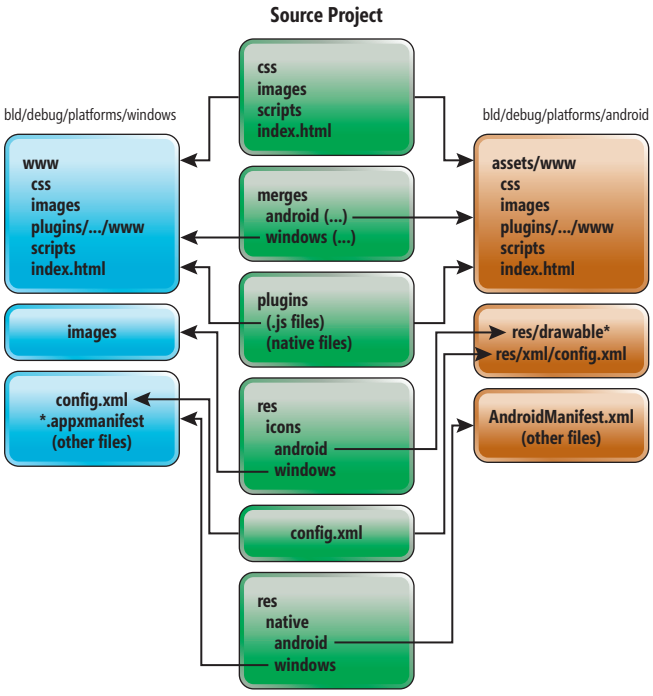


Figure 7 **How Different Parts of a Source Project End Up in Platform-Specific Build Folders (Subject to Change)**

Although we could use the HTML5 geolocation API and write a bunch of code to share to specific social networks, we start running into problems with camera capture because of two primary WebView limitations that can affect Cordova development (for example, on iOS and Android):

1. WebViews block JavaScript from accessing any native platform APIs because untrusted JavaScript code loaded from a potentially remote source poses a security threat. (On Windows platforms, in-package JavaScript is trusted and can call native APIs, but remotely loaded script can't.)
2. WebViews typically support only a subset of HTML5 APIs (varied by platform, check caniuse.com), and many native capabilities are far from having suitable HTML5 standards.

Fortunately, a native app and any WebView it contains can communicate through special mechanisms provided by the OS.

Figure 8 **Default Code from the Cordova App Template**

```
(function () {
  "use strict";

  document.addEventListener('deviceready', onDeviceReady.bind(this), false);

  function onDeviceReady() {
    document.addEventListener('pause', onPause.bind(this), false);
    document.addEventListener('resume', onResume.bind(this), false);

    // Perform other initialization here.
  };

  function onPause() {
  };

  function onResume() {
  };
} )();
```

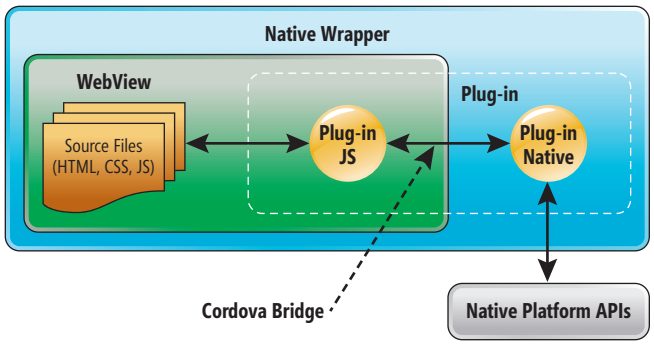


Figure 9 **The Structure of Cordova Plug-ins When Bridging Is Required, as on iOS and Android**

Cordova abstracts these mechanisms through plug-ins as illustrated in **Figure 9**. Simply put, a plug-in contains a universal JavaScript interface that's loaded into the WebView, which in turn uses the platform's communication channel to talk to a piece of native code that's compiled for that platform and can use native APIs. Again, where the underlying platform supports native JavaScript apps, it's not necessary to go through such a bridge, but plug-ins still serve to abstract platform differences.

The ASF provides a core set of Cordova plug-ins for common native APIs such as storage, media, networking, geolocation, sensors, dialogs, media capture, notifications, device information and globalization. When you select these plug-ins within the Visual Studio config.xml editor, the appropriate code is pulled into your builds automatically.

> Fortunately, a native app and any WebView it contains can communicate through special mechanisms provided by the OS.

You can also browse the healthy ecosystem of third-party plug-ins on the Cordova Plugins Registry (plugins.cordova.io) and other sites such as plugreg.com. These cover all manner of features including advertising, Microsoft Azure Mobile Services, Bluetooth communication, social sharing and much more. And because plug-ins aren't obligated to simply surface native APIs on a 1-to-1 basis, plug-in developers often add value by building higher-level capabilities such as barcode scanning and integration with Web services. A great example is the plug-in for Cortana, the vocal assistant for Windows Phone. Developed by Microsoft Open Technologies Inc., a wholly owned subsidiary of Microsoft focusing on open source technologies, it was warmly received by developers at a PhoneGap Day in October 2014. Be mindful, though, that platform support varies widely among plug-ins. For example, the Cortana plug-in supports only Windows Phone 8.1.

Whatever the case, the Visual Studio config.xml editor helps you easily pull plug-ins into your project from sources such as the Cordova registry or GitHub, as described in the MSDN Library article, "Manage Plugins for Apps Built with Visual Studio Tools for Apache Cordova," at bit.ly/10ov6Fo. We'll present an example shortly. Visual Studio automatically pulls in the right files so the appropriate methods get added to the global namespace without you having to explicitly reference a plug-in's JavaScript file.

Overall, when thinking about native APIs, the convention with Cordova is to first look for a suitable plug-in rather than adding platform-specific code to your app. If you find one that's close to what you want, consider extending it—after all, many are written by developers such as you, and contributions are welcome on GitHub. Otherwise, refer to the "Plugin Development Guide" section of the Apache Cordova Documentation to create a plug-in of your own that you can then share with the Cordova community.

Completing the App

Knowing how we access native APIs through plug-ins, we can complete the features of our app. For geolocation, we pull in the Cordova Geolocation plug-in and add this code to our deviceready handler to initialize the location field and wire up the locate button:

```
locate();
document.getElementById("btnLocate").addEventListener("click", locate);
```

Figure 10 **Implementing Geolocation**

```
function locate() {
  navigator.geolocation.getCurrentPosition(function (position) {
    App.lastPosition = {
      latitude: position.coords.latitude, longitude: position.coords.longitude,
      address: "(" + position.coords.latitude + ", "
      + position.coords.longitude + ")"
    };

    // Go translate the coordinates into an address using the Bing Map Web API.
    updatePosition();
  }, function (error) {
    WinJS.log && WinJS.log("Unable to get location: " + error.message, "app");
  }, {
    maximumAge: 3000, timeout: 10000, enableHighAccuracy: true
  });
}
```

Figure 11 **Working with the Cordova Camera Plug-In**

```
function capturePhoto() {
  var photoDiv = this;

  // Capture camera image into a file.
  navigator.camera.getPicture(cameraSuccess, cameraError, {
    quality: 50,
    destinationType: Camera.DestinationType.FILE_URL,
    encodingType: Camera.EncodingType.JPEG,
    mediaType: Camera.MediaType.PICTURE,
    allotEdits: true
    correctOrientation: true // Corrects Android orientation quirks.
  });

  function cameraSuccess(imageFile) {
    // Save for share and enable Share button.
    App.lastCapture = imageFile;
    document.getElementById("btnShare").disabled = false;

    // Do letterboxing and assign to img.src.
    scaleImageToFit(photoDiv.querySelector("img"), photoDiv, App.lastCapture);
  };

  function cameraError(error) {
    WinJS.log && WinJS.log("Unable to obtain picture: " + error, "app");
  };
}
```

Then we implement the locate method, as shown in **Figure 10**.

Wait a minute—isn't navigator.geolocation just the HTML5 API? Well, maybe: The plug-in intelligently uses the HTML5 implementation if it's available, otherwise it adds the same functions to the global namespace that are implemented using native APIs (or fails gracefully if there's no platform support). This way, code migrated from a Web site doesn't have be rewritten.

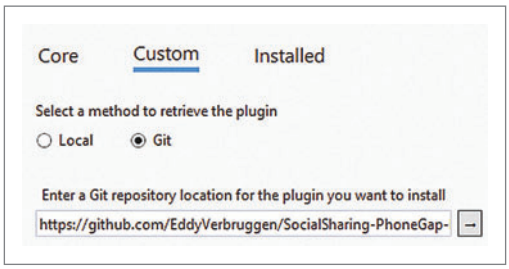


Figure 12 **Adding a Custom Plug-in from GitHub**

Having obtained the location, we store the coordinates in App.lastPosition and create a default App.address string we use for the txtLocation UI element and sharing. Our updatePosition function (not shown) then attempts to use the Bing Maps Web API to translate those coordinates into a meaningful address before setting the txtLocation value. Refer to the accompanying code download for details.

> The last piece of our app is to enforce a portrait-only view, which gives us an opportunity to see how we have full control over platform-specific manifests when necessary.

For camera capture, we initially create a default image and wire up the photo area to do the capture:

```
setPlaceholderImage();
document.getElementById("photo").addEventListener("click",
  capturePhoto.bind(photo));
```

Adding the Cordova Camera plug-in adds the method navigator.camera.getPicture to the global namespace. If successful, this call provides a URI to an image on the device's local file system, which we can just assign to an img.src attribute. In the code shown in **Figure 11**, the assignment happens within our scaleImageToFit function that does letterboxing to preserve the image's aspect ratio.

To implement sharing, which is enabled when we have an image, we found Eddy Verbruggen's SocialSharing plug-in on plugreg.com, which works with both text and images. When we add it as a custom plug-in directly from GitHub, as shown in **Figure 12**, Visual Studio downloads everything we need automatically. (This process downloads everything from the repository, but the only essential parts are the plug-in's plugin.xml file and the src and www folders; everything else can be removed.)

Now we can wire up the Share button and call the plug-in's socialsharing.share API (in the window.plugin namespace) to get the lovely UI shown in **Figure 13** that provides a variety of sharing choices similar to the Share charm on Windows 8/8.1:

```
document.getElementById("btnShare").addEventListener("click", App.share);
function share() {
  var txtLocation = document.getElementById("txtLocation").value;
  plugins.socialsharing.share("At " + txtLocation, "Here My Am!",
    App.lastCapture);
}
```

But there's a snag: SocialSharing supports Android, iOS and Windows Phone 8, but not Windows 8/8.1 and Windows Phone 8.1. Quite a few third-party plug-ins have limited platform support, in fact, presenting clear opportunities to help improve the plug-in for the community's benefit. For this article, though, it gives us a chance to demonstrate using the merges folder, because a Cordova app on Windows runs within the system host where we *can* call native APIs directly from JavaScript and bypass the plug-in altogether.

First, we'll move the aforementioned share function into a share.js file and expose it as App.share:

```
WinJS.Namespace.define("App", {
  configurePlatformSharing: function () { },
  share: share
});

function share() {
  var txtLocation = document.getElementById("txtLocation").value;
  plugins.socialsharing.share("At " + txtLocation, "Here My Am!",
    App.lastCapture);
}
```

Next, in merges/windows/share.js, which will replace the platform-neutral share.js, we provide a different App.share function that invokes the Share charm:

```
function share() {
  Windows.ApplicationModel.DataTransfer.DataTransferManager.showShareUI();
}
```

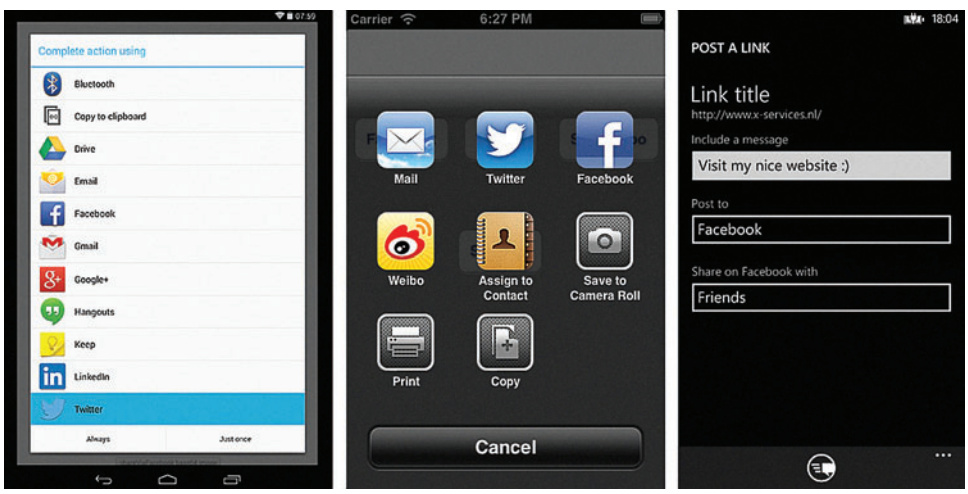


Figure 13 **Sharing on Android, iOS and Windows Phone 8 (Images Courtesy of the SocialSharing Plug-In)**

Tips, Tricks and Notes

Moving targets: To easily switch between target platforms, add the Solution Platforms control to your toolbar as described in the MSDN Library article, "Create Your First Hello World App," at **bit.ly/1th836k**.

PhoneGap and Cordova: Cordova began as a mobile framework and build system called PhoneGap produced by Nitobi Software Ltd., designed to bridge the gap between different mobile phone platforms by abstracting differences behind a framework. Adobe Systems Inc. acquired Nitobi in 2011 and subsequently open sourced the framework to the Apache Software Foundation as Apache Cordova, to which Microsoft Open Technologies Inc. is a significant contributor. Adobe runs the cloud-based PhoneGap Build system that creates native app packages from Web code so you don't need native SDKs on your own machine.

Android Emulators: There's near-universal agreement in the developer community that the raw Android SDK emulators are worth avoiding like dial-up Internet unless you configure virtual machine (VM) acceleration. Visual Studio 2015 Preview includes the fast Hyper-V/x86-based Visual Studio Emulator for Android (**bit.ly/1xlRwFT**), which is compatible with the Windows Phone Emulator (and which, once installed, works with Visual Studio 2013 Update 4). You can also use the GenyMotion Emulator for Android if you don't want to install the Visual Studio 2015 Preview.

Cordova's Sweet Spot: Developers generally realize that running HTML, CSS and JavaScript inside a Cordova app won't always produce a truly native experience on every platform. Cordova apps, then, are ideal for highly branded experiences that needn't conform to native UI patterns, and for enterprise apps where differentiation across platforms isn't desired. Cordova is also an efficient means to quickly build production apps that test ideas in the marketplace, helping you understand where to make deeper business investments.

Referencing Remote Script: Web applications often load JavaScript libraries from remote sources, which makes sense because a user has to be online to see the Web site at all. Mobile apps, however, should provide a great experience without connectivity, so it's best to download remote script and include it directly in your project. This is required on Windows 8/8.1 and Windows Phone 8.1 where the system host disallows remote script; loading a library that attempts to load other remote script can be problematic here, and you might need to do some manual patching to make it all work.

One Platform at a Time: Using a cross-platform tool such as Cordova doesn't mean you must ship an app on multiple platforms simultaneously. You might target only a single platform initially, knowing that your investments will apply to other platforms later. This way you can optimize your UI and CSS for one platform at a time.

To make this work, though, we must wire up a handler during initialization for the DataTransferManager.ondatarequested event. This is the purpose of the stub method configurePlatformSharing, which does nothing on non-Windows platforms when we call it in the deviceready handler:

```
App.configurePlatformSharing();
```

In the Windows-specific share.js, however, we do more:

```
WinJS.Namespace.define("App", {
  configurePlatformSharing: configurePlatformSharing,
  share: share
});

function configurePlatformSharing() {
  var dataTransferManager =
    Windows.ApplicationModel.DataTransfer.
    DataTransferManager.getForCurrentView();
  dataTransferManager.addEventListener("datarequested", provideData);
}
```

In the preceding code, provideData creates the appropriate data package, the result of which is a native Share charm experience on Windows 8.1 and Windows Phone 8.1, as shown in **Figure 14**.

The last piece of our app is to enforce a portrait-only view, which gives us an opportunity to see how we have full control over platform-specific manifests when necessary via the res/native folder. In this case, setting the Common | Orientation to Portrait in the config.xml editor is sufficient for Android, but as of this writing doesn't affect Windows and Windows Phone. No problem—we can just make some custom entries in the app manifests.

To do this, run a build and go to bld/platforms/windows where you'll find generated manifests named package.windows.appxmanifest (Windows 8.1), package.phone.appxmanifest (Windows Phone 8.1), and package.windows80.appxmanifest (Windows 8). Copy whichever ones you need into res/native/windows and make your changes. To enforce portrait orientation, this is all we need within the Application | Application | VisualElements node (shown for Windows 8.1; use the namespace prefix m3: for Windows Phone 8.1 and no prefix for Windows 8):

```
<m2:InitialRotationPreference>
  <m2:Rotation Preference="portrait" />
  <m2:Rotation Preference="portraitFlipped" />
</m2:InitialRotationPreference>
```

At build time, Cordova translates the start page setting and individual preferences in config.xml to the platform-specific manifest, along with platform-specific details from plug-ins. Specifically, if you look in the plugin.xml file in any plug-in folder, you'll see entries such as this:

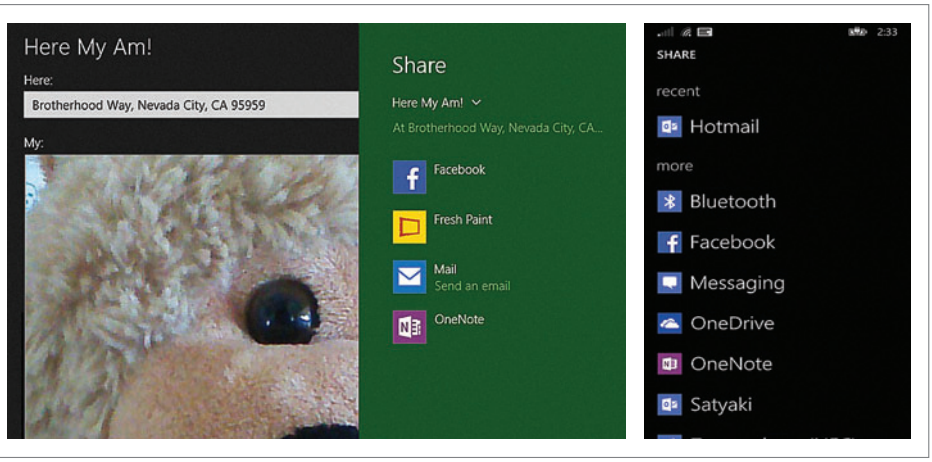


Figure 14 **Sharing on Windows 8.1 and Windows Phone 8.1**

```
<platform name="windows">
  <config-file target="package.windows.appxmanifest"
    parent="/Package/Capabilities">
    <Capability Name="picturesLibrary" />
    <DeviceCapability Name="webcam" />
  </config-file>
  <!-- ... -->
</platform>
```

The preceding code says, "For Windows targets, in the package.windows.appxmanifest file, add these Capability and DeviceCapability elements under the Package | Capabilities node." So, clearly, an orientation plug-in could do the same job we've done manually. And with that, we have a functional app!

Debugging, Deployment and Other App Lifecycle Concerns

As a developer, you clearly know that writing code is just one part of the overall process of producing market-ready software, and Visual Studio has quite a bit to offer here, as shown in **Figure 15**.

It's helpful to know as you step into building Cordova apps that you'll find quirky behaviors on every platform. This is just the nature of cross-platform development at present, which means you should realistically expect to do as much per-platform debugging as you would writing native apps for each target. Similarly, deployment happens with each app store individually, using tools specific to those platforms. Cordova doesn't shortcut those processes.

The platform-specific builds will pull in the appropriate resources in your project's res folder: Be sure to review that content and confirm you have all the right graphics and other files for each platform. When getting your final package together, also double-check the manifests to make sure your app and your company are correctly identified and that you're not carrying over data from the default project template.

> You should realistically expect to do as much per-platform debugging as you would writing native apps for each target.

And when your app is ready to go out to real customers, dedicate yourself to responding to issues with timely app updates. There's so much device variance within the realm of even a single platform that customers will effectively be your field testers—be sure to honor that role with quick updates.

Figure 15 **Additional Visual Studio Features for Cordova Apps**

Feature	Description/Notes
Debugging	At the time of writing, the following devices and emulators support debugging in Visual Studio with the JavaScript Console and DOM Explorer: • Android 2.3.3 and later (4.4 or later is best) • iOS 6, 7, 8 • Windows 8 and 8.1 • Apache Ripple (Chrome-based emulator) for Android and iOS Debugging on Windows Phone 8 and 8.1 isn't presently supported, but you can use the Web Inspector Remote (weinre) as described in a Microsoft Open Technologies blog post at bit.ly/1y9k0zr. When running in Ripple, you might encounter the infamous message, "I Haz Cheeseburger?!?!" For information on this, refer to the Visual Studio Tools FAQ at bit.ly/1x71P0q, and when in doubt, just hit the Success button. There's also a note in the default index.js file that says to run the window.location.reload function in the JavaScript Console to debug initialization code; be mindful that this reloads the *most recent* build and doesn't rebuild the app or apply any changes to source files.
IntelliSense	Visual Studio gives you IntelliSense on HTML, CSS and JavaScript code, including a growing number of third-party libraries such as JQuery, Backbone.js and Require.js.
Test	Appium and Chutzpah are popular open source automation tools. The Chutzpah Test Runner for Visual Studio integrates with the Visual Studio Test Explorer and Team Foundation Build. For broad device testing you can use services such as Perfecto Mobile.
Application Lifecycle Management	Visual Studio ALM features provide for agile planning, source control (hosted on Team Foundation Server or GitHub), team collaboration, and release management for cross-platform projects and team members using other IDEs.

What to Expect in Upcoming Visual Studio Releases

In closing, we've already seen how Visual Studio Tools for Apache Cordova go a long way to simplifying cross-platform Cordova development. Microsoft Open Technologies is a significant contributor to the Cordova project itself, and the Visual Studio Tools team is committed to making the experience even better—what we've covered in this article is just a preview! Moving forward, the team is looking at app features such as push notifications, greater IntelliSense, better debugging on more platforms, HTML/CSS designers, improved support for unit testing, support for Team Build, and speeding up and supporting the standard Web developer workflow. Are there other features you'd like to see? Give us your feedback through the Microsoft UserVoice site (visualstudio.uservoice.com) or @VSCordovaTools on Twitter, and join the discussion about the toolkit on StackOverflow (bit.ly/1yCReYa). ■

Kraig Brockschmidt *works as a senior content developer in the Microsoft Developer Division and is focused on providing developer guidance for mobile cloud-connected apps on Windows, iOS and Android. He's the author of "Programming Windows Store Apps with HTML, CSS and JavaScript" (two editions) from Microsoft Press and shares other insights on kraigbrockschmidt.com/blog.*

Mike Jones *is a programmer writer who's been working on mobile at Microsoft for 10 years, from Compact Framework to Silverlight for Windows Phone to Visual Studio Tools for Apache Cordova. He's been enjoying JavaScript for the last few years after a long, fun fling with C#, et al.*

Thanks to the following Microsoft technical experts for reviewing this article: *Chuck Lantz, Eric Mittelette, Parashuram Narasimhan, Ryan Salva, Amanda Silver, Cheryl Simmons and Priyank Singhal*

Cross-Platform Game Development with Visual Studio Tools for Unity

Adam Tuliper

I've noticed some interesting things in my foray into game development. A friend created an Android game that recently hit No. 1 on the Google Play Store. I have some other friends at another indie game company who have reached a top spot on iOS. I know yet another who has more than 500,000 players on a Windows Store game. They all share something in common regardless of the platform they build for: They all use Unity on Windows to develop their games. (That's Unity the game engine, not the dependency injection package from Microsoft.)

Unity on Windows accounts for a whopping 88.6 percent of Unity projects, far exceeding OS X (stats.unity3d.com). I find that stat quite interesting because Unity is the most popular cross-platform game development middleware. On the mobile side, iOS takes up about 26 percent of the mobile platform usage category, with Android at 71 percent, yet Unity on Windows attracts the largest number of developers, demonstrating Windows is still the No. 1 platform on which developers want to code.

This article discusses a prerelease version of Visual Studio 2015. All related information is subject to change.

This article discusses:

- The advantages of using VSTU for game development
- How VSTU works
- Using the Unity Editor
- Developing for the Mac
- Cross-platform development

Technologies discussed:

Visual Studio Tools for Unity

Visual Studio Tools for Unity (VSTU) is a software package (formerly known as UnityVS) that Microsoft obtained through the acquisition of SyntaxTree in 2014. This product used to cost about $100 and allowed Unity developers to use Visual Studio to not only edit code, but also to debug code. Why is this a big deal? First, because Microsoft made this great tool 100 percent free. Second, it's being actively developed by the SyntaxTree team that joined Microsoft. Third, it's great because I can use the familiar code editor/debugger to develop my games. As soon as Visual Studio 2015 Preview bits were released, the update to VSTU was released to support it, as well.

In Unity, for any GameObject you want code assigned to, simply add a script component to it. To edit that code, you double-click it. Unity then opens MonoDevelop by default (which installs with Unity) to edit and debug code. For me, there's a bit of friction here when developing this way. I think the MonoDevelop project has done some great work, but Visual Studio is a more full-featured tool. I want to use Visual Studio to write and debug code, just like I have for the past 18 years. Previously in Unity, you could use any editor (such as Visual Studio, Sublime and so on) to edit your Unity code, but you couldn't debug your code from Visual Studio. That means you would edit code in Visual Studio and switch back to MonoDevelop to debug, or drop Debug.Log statements everywhere, which isn't the most productive way to debug.

Many readers of *MSDN Magazine* are enterprise developers. Many I speak to have wondered for a long time about writing games, especially after XNA went the way of the dodo. For those folks, I recently did a four-part series for the magazine on game development with Unity (msdn.microsoft.com/magazine/dn759441) to help new audiences get introduced to the experience of game development on Windows with Unity. Most enterprise developers I know use Visual Studio at work,

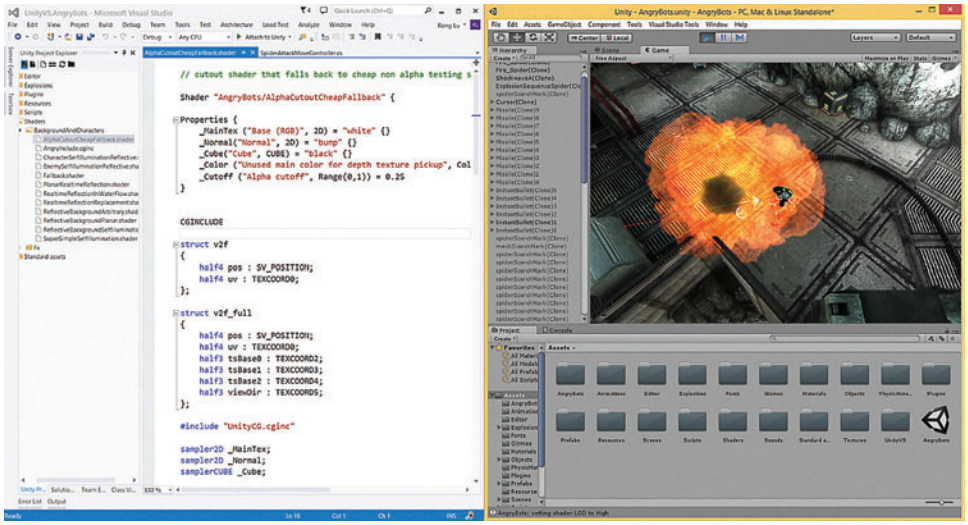


Figure 1 **New Syntax Highlighting for Shaders in Visual Studio 2015 Preview**

but can't use it at home for game development because their organization is the license holder. MonoDevelop works in this case, but it doesn't provide the Visual Studio experience. Microsoft recently released Visual Studio Community Edition, a free version virtually the same as Visual Studio Professional. It's awesome for small teams or the individual developer because it's free and can run VSTU. Coupled with the free version of Unity, you have a completely free multiplatform game development system using Unity and Visual Studio. For those wondering about Visual Studio Express, that product doesn't support plug-ins, so you can't run VSTU in it. There are no future versions of Visual Studio Express planned because Visual Studio Community Edition is the future—and a great future it is.

Why Use VSTU?

If you've developed great games with Unity and MonoDevelop, why would you use something different? What does VSTU give in addition to MonoDevelop? Well, the IntelliSense is better. The code formatting works better. You can use plug-ins such as ReSharper to refactor and enhance your code. If you're used to Visual Studio, you're surely used to the Solution Explorer, of which MonoDevelop has a limited version, but which doesn't provide nearly the same experience. There's a better debugging experience through the Output window in Visual Studio, as well, so you can see the result from commands such as Debug.Log without having to switch back and forth to Unity to view the output as you normally would. VSTU also enhances the debugging experience by providing better visualizations for collections, such as lists, dictionaries and the often-used (in Unity) Hashtable. Unity contains functionality for its own SerializedProperty, which allows your custom editor extensions to read values in the inspector. This type displays as expected with VSTU, but when being visualized in MonoDevelop, it spews lines and lines of errors in the Unity console.

Also, with the release of Visual Studio 2015 Preview, there's a great new feature added to VSTU: syntax highlighting for shaders (see **Figure 1**). While it might not be considered a huge productivity booster, it's significantly easier on the eyes when writing shaders, and it was a top-requested feature on the UserVoice site used to gather feature suggestions and feedback. Anyone who has used modern tools with syntax highlighting and then had to develop without it quickly misses the functionality, so a lot of you will cheer this release.

How Does It All Work?

Before further exploring new features, I'll discuss how the "magic" behind the tool works. Unity contains a virtual environment inside of its editor that's controlled by a soft debugger. When you debug a game, you aren't debugging Unity.exe, as you might expect, because you can attach a debugger to a process from Visual Studio. Instead, you're connecting over a port to Unity and then sending commands to the Unity soft debugger running inside of the editor. MonoDevelop ships with a MonoDevelop.Debugger.Soft library that understands the binary format used to talk back to the Unity soft debugger. Unity adds a thin wrapper around this library to support custom actions and distributes that with MonoDevelop, as shown in **Figure 2**. The plug-in can launch Unity or just connect over a port to talk to the soft debugger (to check out the

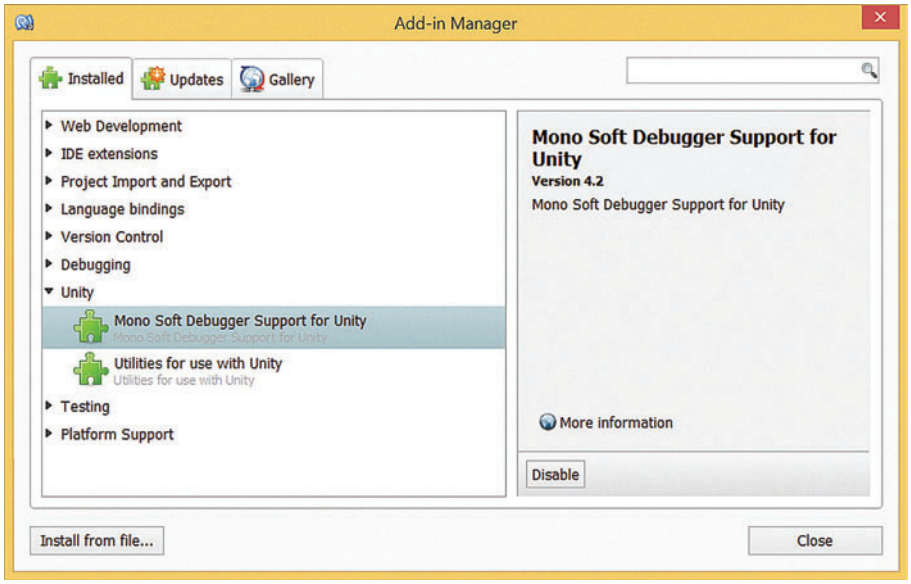


Figure 2 **The Unity Plug-in for Debugging from MonoDeveloper**

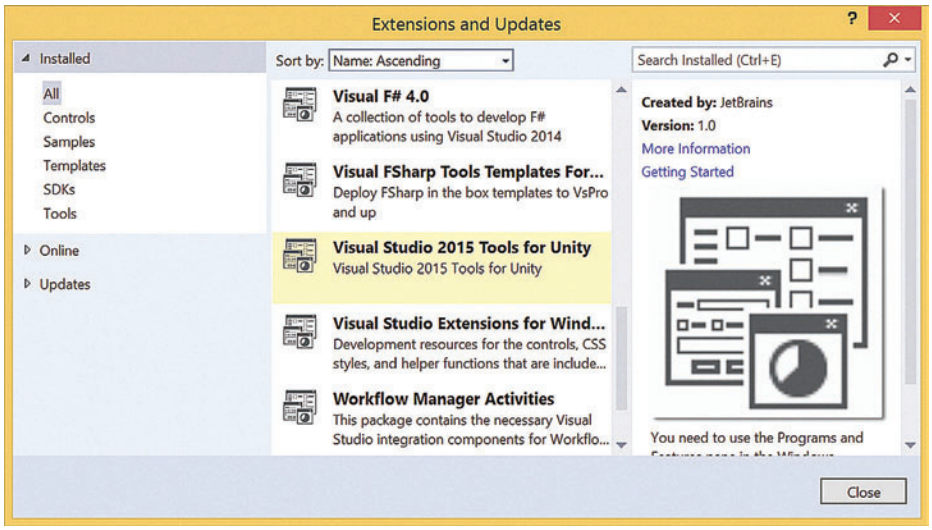


Figure 3 **The Microsoft Visual Studio Tools for Unity Extension in Visual Studio 2015 Preview**

code, see the GitHub project at bit.ly/1zY6K2a). The good folks working on VSTU have in turn written an extension for Visual Studio that uses this library, as well (see **Figure 3**). It allows you to not only use Visual Studio in all its code-editing, refactoring, syntax-highlighting and IntelliSense glory, but to also debug Unity code using the debugging experience you're used to in Visual Studio—and it works better than what's built in to the default Unity distribution.

Unity creates its own .csproj files to manage its Mono compilation and some other .csproj files for adding to Visual Studio if you want to use Visual Studio as an editor only. I covered a bit of this Unity architecture in the first article of my Unity series earlier this year. VSTU creates its own versions of the .csproj files, as shown in **Figure 4**. It might seem like there can be many .csproj files to manage, but the details are handled behind the scenes so you don't have to bother with them. The usual F5 debugging in MonoDevelop doesn't always work. Unity will always do its own compilation on your code, but when doing an F5 to debug in MonoDevelop, it will do an additional compilation on the binaries and might error out on some syntax that Unity actually didn't mind and, thus, fail to debug. Every time you start to debug in MonoDevelop, you have to attach to the Unity.exe process (which in turn connects to the port for debugging). This differs from the "press F5 to debug" you've grown used to in Visual Studio. I've also locked up Unity quite a few times while using MonoDevelop to debug, something that hasn't been an issue with VSTU on Windows.

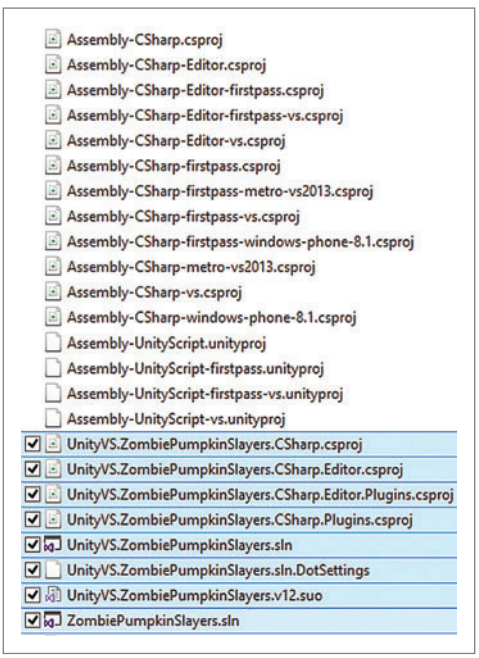


Figure 4 **Visual Studio Tools for Unity Creates Its Own Versions of the .csproj Files**

Using VSTU

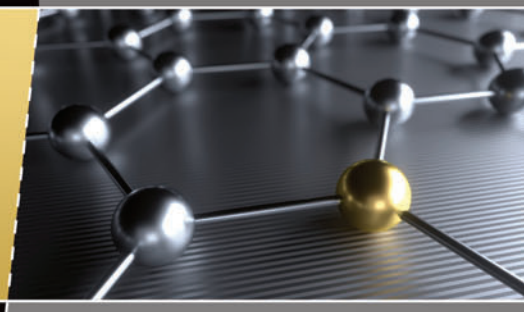
VSTU is simple to use, but it does warrant a brief explanation of its Unity Editor because it might differ from what you expect compared with something such as a Visual Studio extension. A Visual Studio extension can just be installed once and then be active for each project, as with VSTU. The Unity environment is extremely scriptable with C#, which makes it powerful, but for each project you want the functionality in, the architecture requires you to import the required VSTU package. This is a one-time operation for each project. Simply choose the VSTU package from Assets/Import Package, as shown in **Figure 5**. This decompresses the package and tells you the files about to be imported. That's it.

You can also install any .unitypackage file into Unity by double-clicking on it, so for VSTU, you could also just double-click on the "Visual Studio 2015 Tools.unitypackage" file in your file system. Once installed, to edit code you simply double-click on a script file in Unity and it will open in Visual Studio with VSTU active. You *must* remember to install the VSTU package into your Unity project, or Visual Studio will display an ambiguous error that might leave you a little confused. It's also worth mentioning that if your code doesn't compile when you attempt to open it for the first time, the Visual Studio projects won't generate and, therefore, no project loads in Visual Studio, so you'll receive the same ambiguous error. This also happens in Unity with your MonoDevelop projects—they just won't generate if your code initially has an error prior to first-time project generation (which, again, happens behind the scenes). These are just a few things I've experienced of which you should be aware.

Mac Usage

What about you Mac users out there? Because Visual Studio runs on Windows and not OS X, how do the other roughly 11 percent of developers not using Windows enhance their development experience with Unity? You can run Windows in a virtual machine (VM) and share folders so you can switch back and forth for debugging. There's a little friction with that method, too, though. There's a better way. Because VSTU connects to Unity over a port, technically you can do cross-machine development. With virtualization software such as Parallels and VMware on a Mac, you can run software from your guest VM and have it show up in your host OS desktop just like it was running natively there in a window. This means Visual Studio can run on your OS X desktop just like any other OS X

Switch to Amyuni PDF



Create & Edit PDFs in .Net - ActiveX - WinRT

- 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 WinRT Component enables publishing C#, C++CX or Javascript apps to Windows Store
- New Postscript/EPS to PDF conversion module

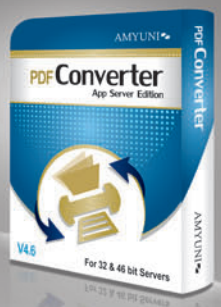
Complete Suite of Accurate PDF Components

- 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

Advanced HTML to PDF & XAML

- Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver
- Easy Integration and deployment within developer's applications
- WebkitPDF is based on the Webkit Open Source library and Amyuni PDF Creator

High Performance PDF Printer For Desktops and Servers



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2012 R2 and Windows 8.1
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language
- Easy licensing and deployment to fit system administrator's requirements

CERTIFIED FOR Windows Server 2008 · Windows 7 · Windows 8 · Windows Server 2012 Certified

AMYUNI

All development tools available at

www.amyuni.com

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

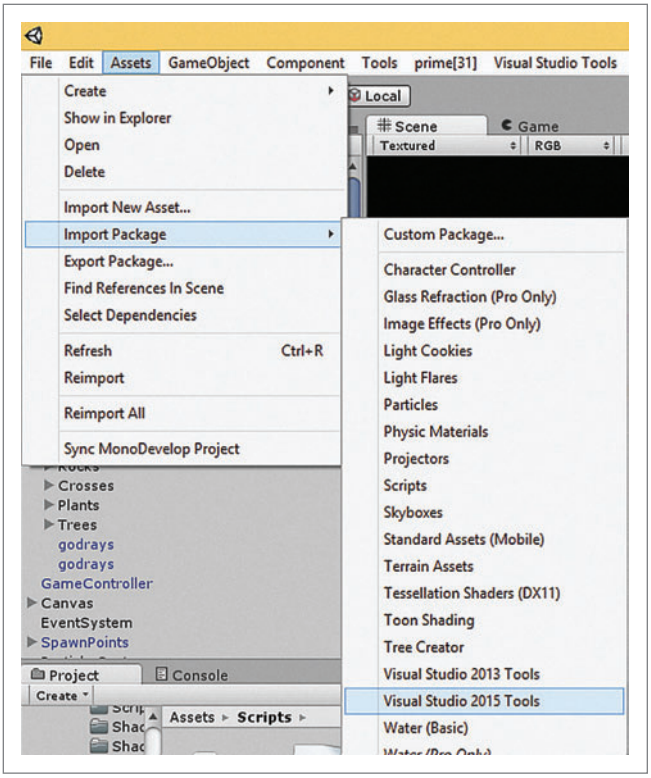


Figure 5 **Importing the Visual Studio Tools for Unity Package into Unity**

application and connect to Unity *running on your Mac to debug it from Windows*. Yeah, that's pretty cool.

The Cross-Platform Approach

We certainly live in an amazing time of multiplatform app development. Sometimes, though, this app dev landscape feels like we're living in a single land with many different currencies, and some vendors take some and not another, and everyone speaks a different language. There are so many different tools, platforms and devices, you could keep busy for life just learning them all. At Microsoft, we've worked closely with many partners to bring cross-platform development to our tooling and platform for an integrated experience. I can spin up a Linux VM on Microsoft Azure. I can use Visual Studio to debug Python scripts (even running remotely in the cloud), and I can read great documentation on microsoft.com about setting up push notifications on an iPhone device. On top of all that, .NET is now open source, ASP.NET runs on Linux and OS X, and Visual Studio has an Android emulator. I certainly didn't foresee this day, and I believe I just saw a pig fly by my window outside.

Additional Learning

Visual Studio Tool for Unity: UnityVS.com

Adam Tuliper's Channel 9 Blog: bit.ly/AdamChannel9

Microsoft Virtual Academy—Developing 2D and 3D Games with Unity for Windows: bit.ly/FreeUnityTraining

Unity Resources: unity3d.com/learn

Unity, of course, is my favorite cross-platform game development tool, although I have seen impressive implementations of business apps with advanced 3D capabilities that have been done in Unity. Unity supports *16* different platforms—yes, you read that correctly. That's quite impressive and you're likely curious what makes up that number, so here you go: iOS, Android (including devices such as OUYA), Windows, Windows Store, BlackBerry, Windows Phone, OS X, Linux, Web (via plug-ins previously and now HTML5/JavaScript/WebGL), PS3, PS4, PS Vita, PS Mobile, Xbox One, Xbox 360 and WiiU.

You also have several great choices for developing cross-platform applications solely with Visual Studio. The flagship Microsoft IDE now has integration with Apache Cordova, allowing cross-platform HTML-based applications (not Web sites—these are apps) to be developed from within its familiar environment. You can debug these applications from within Visual Studio for Windows, Android and even iOS. For an example of this, check out the MSDN Library article, "Run Your Apache Cordova App on iOS," at bit.ly/1ycNUVD.

In the cross-platform app space, the leader is Xamarin (pronounced zam-a-rin for those who still are unsure). Xamarin isn't gaming-focused, although you definitely can create games with it. The tooling can integrate with Visual Studio, as well, for excellent cross-platform app development, which even allows you to design a UI for iOS and Android all from within Visual Studio. Xamarin has had since its inception part of the original Mono team, so that expertise has served the company well. Mono is also what powers your custom game code and extensibility in Unity—it has come an incredibly long way since its creation in 2001.

For those who love C++ (you know who you are), I touched briefly on the Android emulator earlier, and it will support C++ apps. That's right: Visual Studio now lets you create a native Android app via the Cross Platform | Native-Activity Application template. This allows development and debugging of Android code from Visual Studio starting with Visual Studio 2015 Preview. It was all I could do to not italicize that entire last line!

Wrapping Up

To review, Microsoft has been adding to its cross-platform story for some time, but now it's in overdrive. VSTU is, of course, free and has been going through some great updates; you can find a history of them at bit.ly/15pC1RN. Visual Studio Community Edition is free. Microsoft is known for having great development tools and is committed to bringing the best development and debugging experience to game developers using Visual Studio. So I say to all: Use Unity, develop on Windows and use Visual Studio for an awesome development and debugging experience. ■

Adam Tuliper *is a senior technical evangelist with Microsoft living in sunny Southern California. He's an indie game dev, co-admin of the Orange County Unity Meetup and a Pluralsight author. He and his wife are about to have their third child, so reach out to him while he still has a spare moment at adamt@microsoft.com or on Twitter at twitter.com/AdamTuliper.*

Thanks to the following Microsoft technical expert for reviewing this article: *Jb Evain*

msdn magazine VP

MSDN MAGAZINE VENDOR PROFILE

Syncfusion®
Deliver innovation with ease®

Big Data and Predictive Analytics Solutions from Syncfusion

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. 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 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.
- 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/BigData
- A complete production environment is also coming soon.

Why Predictive Analytics?

- Predictive modeling solutions enable enterprises to use accumulated data to their advantage.
- Analyzing data from past transactions can reveal a lot about the future, like which customers will pay on time, or which are most likely to stop using your service or product.
- The down side is that until now, commercial modeling solutions have been very expensive. Additionally, the integration of modeling solutions with Windows-based .NET solutions has been complicated.

How can Syncfusion help developers with Predictive Analytics?

- Essential Predictive Analytics combines the powerful R® modeling environment with Syncfusion's own deployment framework to provide smooth integration with .NET. Our deployment framework can also be used with other modeling environments, such as SAS® and SPSS®.
- Deployment is completely independent of the modeling framework. You can save on expensive deployment fees.
- Essential Predictive Analytics ships with an easy installer for Windows. Check it out at Syncfusion.com/PredictiveAnalytics.

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.
- We offer the unique ability to develop with the ease of Windows, and then deploy solutions at a fraction of the price of comparable solutions.
- There are no per-node, per-user, or other deployment fees.

What kind of support does Syncfusion provide?

- Unlimited commercial support to enable customers to build big data and predictive modeling solutions.
- Samples, patches, and workarounds where applicable, all delivered to our SLA.
- Stress-free deployment to the Microsoft Azure cloud is guaranteed.

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/datascience**

Expand Visual Studio 2013 with Extensions

Susan Norwood and Doug Erickson

One of the best features in the higher-level versions of Visual Studio is now available to everybody for free through the Visual Studio Community 2013 IDE: the ability to use extensions from the Visual Studio Gallery (find the extensions at bit.ly/115mBzm). So what are extensions? They're plug-ins that let you extend Visual Studio Community 2013 (which you can download at bit.ly/1tH2uyc) and features of Visual Studio to perform new tasks, add new features and refactor code—or even support new languages.

The Microsoft developer community provides a wide variety of extensions on the Visual Studio Gallery. Take a look and you'll be surprised at the kinds of extensions available. Some of them, like the Productivity Power Tools 2013 extension (download at bit.ly/1xE3EAT), you may not want to live without. You can install these extensions from the Web, or you can search within Visual Studio for online extensions using the Extensions and Updates window in the Tools menu. Look at the Online category to find more popular tools like Visual Assist and ReSharper. You can even find more extensions at the Code Gallery (bit.ly/11nzi9Q).

This article discusses:

- How to extend the Visual Studio IDE
- Adding extensions to standard menus
- Sharing custom extensions

Technologies discussed:

Visual Studio 2013 Community, Visual Studio 2013 SDK, Visual Studio Gallery

The Microsoft developer community provides a wide variety of extensions on the Visual Studio Gallery.

Even the best tools may not map perfectly to specific tasks you want to perform or automate within the IDE. You probably have your own personal scripts you've customized and tweaked to make your work easier. Maybe you have one that checks a build directory for a successful build, or runs a transform on XML files or cleans up all the detritus of a complex build process. How would you like to run your tool within Visual Studio as part of the IDE or as part of your build process? Now you can, using Visual Studio 2013 Community and the Visual Studio 2013 SDK.

You can get started by downloading the Visual Studio SDK. It provides all the libraries, tools and project templates for creating a variety of different extensions. Go ahead and install it and you'll be ready.

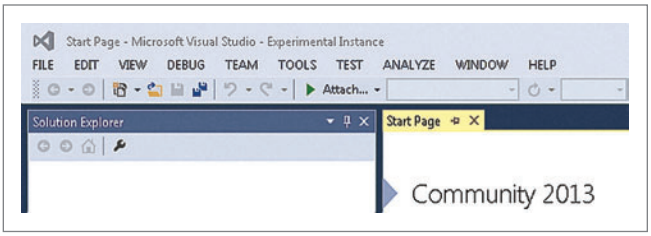


Figure 1 **The Visual Studio Experimental Instance**

Set up Tools to Run in Visual Studio

Once you have the SDK installed, it's pretty easy to make your tool or executable script run from a Visual Studio menu. The ubiquitous Notepad.exe will be the example for this article, but you can use any executable or integrate your own code in the command handler.

A basic extension is a Visual Studio Package binary. You can find the Visual Studio Package template in the New Project dialog under Visual Basic | Extensibility, C# | Extensibility, or Other Project Types | Extensibility.

This article will demonstrate how to make a simple extension that launches Notepad. We'll use the template under C#/Extensibility to create a C# Visual Studio Package project. We'll put it in the D:\Code directory and call it StartNotepad. This extension will ultimately start Notepad from a Visual Studio IDE menu item.

Once you double-click the template, Visual Studio launches a wizard to help you configure the Extension. For starters, just accept the defaults on the first two dialog pages.

Next, take the following steps:

1. On the Select VSPackage Options page, select the Menu Command option.
2. On the Command Options page, set the command name to Start Notepad and the command ID to cmdidStartNotepad.
3. On the Test Options page, uncheck the two checkboxes.
4. Click Finish.

At this point you can build and run the Package project. When the project opens, start debugging (press the F5 key, or use the Start command on the toolbar). A new instance of Visual Studio Community 2013 will start up. When that's complete, you'll see the IDE, with a default to Start Page – Microsoft Visual Studio – Experimental Instance on the title bar (see **Figure 1**). This is your test instance of Visual Studio. It runs separately from your working instance of Visual Studio, so you don't have to worry about contaminating your own development environment if something goes wrong.

The "experimental instance" is really just a fancy way of announcing that you've launched a sandbox instance of Visual Studio for testing your extension. It has all the functions of Visual Studio Community 2013, but it doesn't jeopardize your work in the original Visual Studio instance. For an example this simple, it may seem like overkill. However, if you build an entire design framework or a complex interdependent set of build tools, you don't want to risk your code by running it in the same instance from which you're developing.

On the Tools menu of the experimental instance, open the Extensions and Updates window. You should see the StartNotepad extension here (see **Figure 2**). (If you open Extensions and Updates in your working instance of Visual Studio, you won't see StartNotePad.)

You'll also see Start Notepad under the Tools menu (see **Figure 3**).

Now go to the Tools menu in the experimental instance. You should see the Start Notepad command. At this point it just brings up a message box that says "StartNotepad – Inside MSIT.StartNotepad.StartNotepadPackage.MenuItemCallback()." You'll see how to actually start Notepad from this command in the next section.

> Once you have the SDK installed, it's pretty easy to make your tool or executable script run from a Visual Studio menu.

Set up the Menu Command

Stop debugging and go back to your working instance of Visual Studio. Open the StartNotepadPackage.cs file, which contains the derived Package class. This is the starting point for all VSPackage extensions. The Initialize method of this class actually sets up the command:

```
// Add our command handlers for menu (commands must exist in the .vsct file)
OleMenuCommandService mcs =
  GetService(typeof(IMenuCommandService)) as OleMenuCommandService;
if ( null != mcs )
{
  // Create the command for the menu item.
  CommandID menuCommandID = new CommandID(GuidList.guidStartNotepadCmdSet,
    (int)PkgCmdIDList.cmdidStartNotepad);
  MenuCommand menuItem = new MenuCommand(MenuItemCallback, menuCommandID );
  mcs.AddCommand( menuItem );
}
```

Don't worry about the details of this code right now. You should note this is how the menu command is instantiated. The menus and other UI for VSPackage extensions are defined, as the code comment says, in the .vsct file.

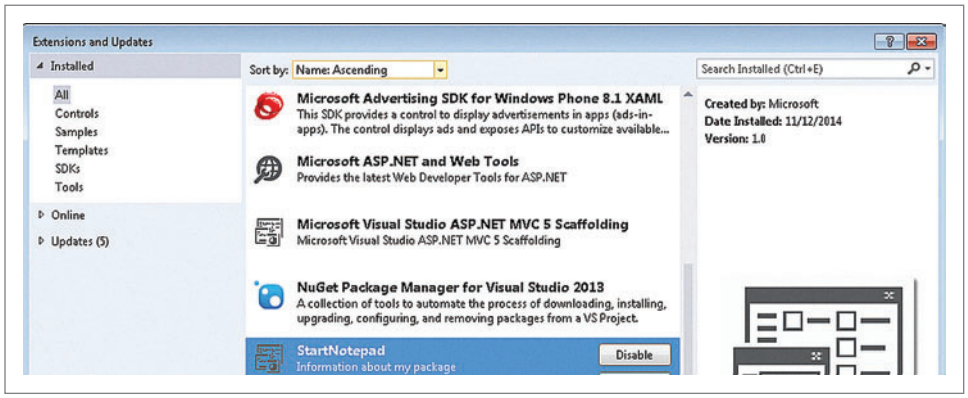


Figure 2 **Extensions and Updates Showing StartNotepad**

The command handler is named MenuItemCallback (in the same StartNotepadPackage class). Delete the existing method and add the following:

```
private void MenuItemCallback(object sender, EventArgs e)
{
  Process proc = new Process();
  proc.StartInfo.FileName = "notepad.exe";
  proc.Start();
}
```

Now try it out. When you start debugging the project and click Tools | Start Notepad, you should see an instance of Notepad. In fact, you'll get a new instance of Notepad every time you click Start Notepad.

> The .vsct file is a handy place to find all the definitions of the Visual Studio UI.

You can use an instance of the System.Diagnostics.Process class to run any executable, not just Notepad. Try it with calc.exe, for example.

Define the Interface

Now you'll use the .vsct file to define the UI for your extension. Take a look at the StartNotepad.vsct file. This is a handy place to find all the definitions of the Visual Studio UI you'll use in your extension. It's an XML file, so brace yourself for angle braces.

Find the <Groups> block. Every menu command must belong to a Group, which tells Visual Studio where to put the command. In this case the command is on the Tools menu, and its parent is the Main menu:

```
<Groups>
  <Group guid="guidStartNotepadCmdSet" id="MyMenuGroup" priority="0x0600">
    <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
  </Group>
</Groups>
```

Don't worry too much about the details yet. This is just to show you where the important elements are. The command itself is defined within the <Buttons> block:

```
<Button guid="guidStartNotepadCmdSet" id="cmdidStartNotepad"
  priority="0x0100" type="Button">
  <Parent guid="guidStartNotepadCmdSet" id="MyMenuGroup" />
  <Icon guid="guidImages" id="bmpPic1" />
  <Strings>
    <ButtonText>Start Notepad</ButtonText>
  </Strings>
</Button>
```

You can see the button is defined with a GUID (guidStartNotepadCmdSet, which is the same as the Group GUID) and an ID (cmdidStartNotepad, which is the ID you specified in the package wizard). The command is parented to the Group. It has an icon and some text. The icon is one of a default set of icons included in the solution. The priority specifies the location of this button on the menu, if there are multiple commands in the command group. The GUIDs and IDs for the groups and buttons are defined in the <Symbols> block:

```
<!-- This is the package GUID. -->
<GuidSymbol name="guidStartNotepadPkg"
  value="{18311db7-ca0f-419c-82b0-5aa14c8b541a}" />
<!-- This is the GUID used to group the menu commands together -->

<GuidSymbol name="guidStartNotepadCmdSet"
  value="{b0692a6d-a8cc-4b53-8b2d-17508c87f1ab}">
  <IDSymbol name="MyMenuGroup" value="0x1020" />
  <IDSymbol name="cmdidStartNotepad" value="0x0100" />
</GuidSymbol>
```

The bitmaps are also defined in the <Symbols> block:

```
<GuidSymbol name="guidImages" value="{b8b810ad-5210-4f35-a491-c3464a612cb6}" >
  <IDSymbol name="bmpPic1" value="1" />
  <IDSymbol name="bmpPic2" value="2" />
  <IDSymbol name="bmpPicSearch" value="3" />
  <IDSymbol name="bmpPicX" value="4" />
  <IDSymbol name="bmpPicArrows" value="5" />
  <IDSymbol name="bmpPicStrikethrough" value="6" />
</GuidSymbol>
```

Try changing the icon to one of the other ones defined here. None of the choices are particularly appropriate for Notepad, so choose the strikethrough because it shows another part of the setup. Although the icons are defined in the <Symbols> block, they must also be listed in the usedList attribute of the <Bitmaps> block:

```
<Bitmap guid="guidImages" href="Resources\Images.png" usedList="bmpPic1,
  bmpPic2, bmpPicSearch, bmpPicX, bmpPicArrows"/>
```

You see bmpPicStrikethrough isn't listed, although it has been defined. If you change the icon to this bitmap, it won't appear on the menu. So add bmpPicStrikethrough:

```
<Bitmap guid="guidImages" href="Resources\Images.png" usedList="bmpPic1,
  bmpPic2, bmpPicSearch, bmpPicX, bmpPicArrows, bmpPicStrikethrough"/>
```

Now you can change the icon to bmpPicStrikethrough for the menu button:

```
<Button guid="guidStartNotepadCmdSet" id="cmdidStartNotepad"
  priority="0x0100" type="Button">
  <Parent guid="guidStartNotepadCmdSet" id="MyMenuGroup" />
  <Icon guid="guidImages" id="bmpPicStrikethrough" />
  <Strings>
    <ButtonText>Start Notepad</ButtonText>
  </Strings>
</Button>
```

Now you can try it out. When the experimental instance comes up, you should see something like **Figure 4** on the Tools menu.

Now add a keyboard shortcut for the Start Notepad menu command. This example will use Ctrl+1. If you add a keyboard shortcut, be sure to pick an obscure key combination so your command doesn't collide with a more standard combination. You can do this entirely in the .vsct file. Keyboard shortcuts are called KeyBindings in the .vsct file. Add the following block to the .vsct file:

```
<KeyBindings>
  <KeyBinding guid="guidStartNotepadCmdSet" id="cmdidStartNotepad"
    editor="guidVSStd97" key1="1" mod1="CONTROL"/>
</KeyBindings>
```

The key1 attribute declares the standard key and the mod1 attribute declares the modifier or accelerator key (typically Ctrl, Alt or Shift) pressed in conjunction with the standard key. If you add mod2="ALT" to add a second modifier key, you would have Ctrl+Alt+1 as the shortcut. Stick with Ctrl+1 for now. Keep in mind, though, the .vsct file is a good place to start your customizations.

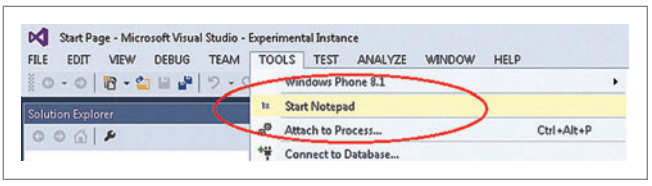
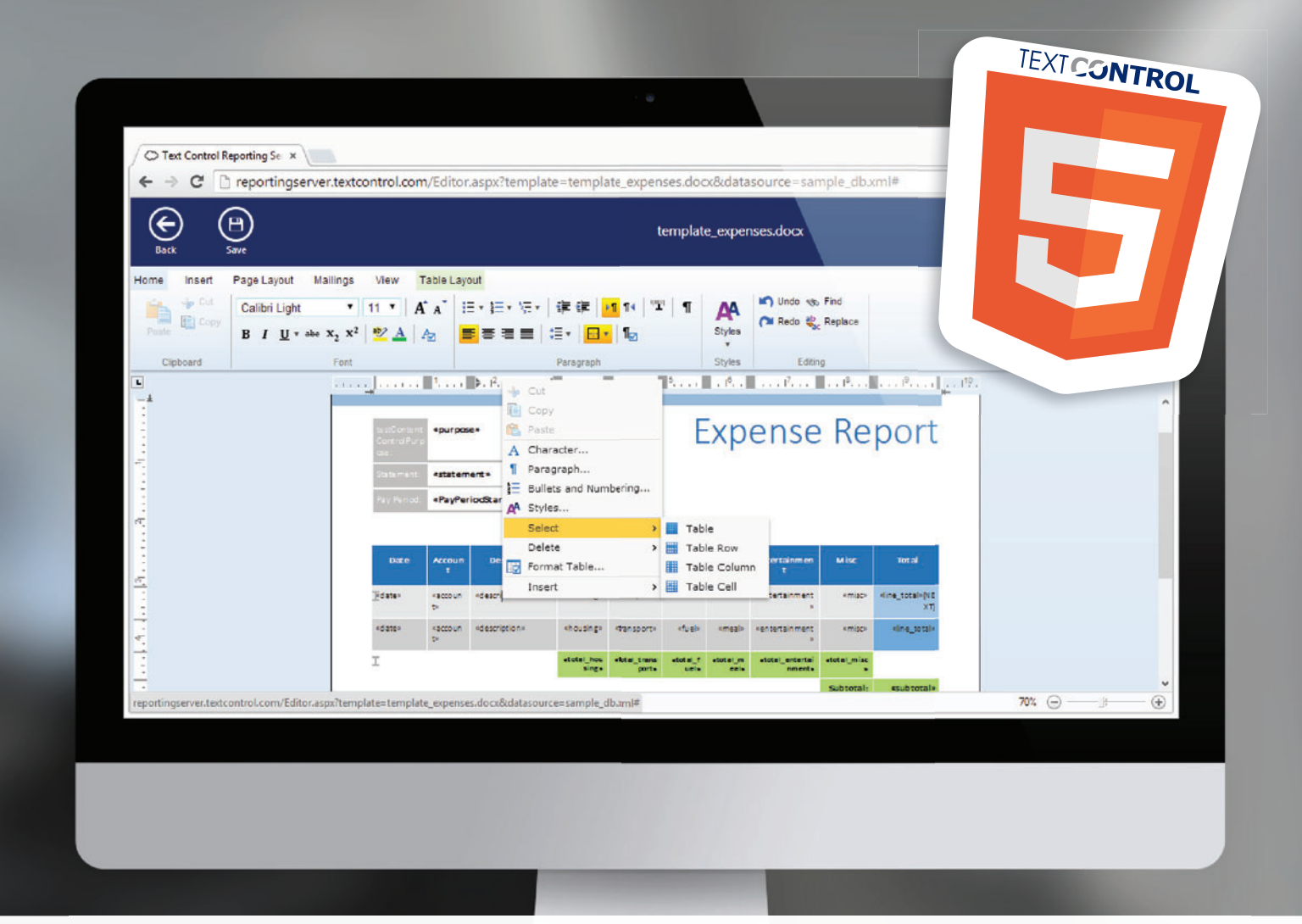


Figure 3 **New Menu Item Confirmation for StartNotepad Under Tools**



Cross-browser, cross-platform document and template editing

EDIT MS WORD DOCUMENTS IN ANY BROWSER

The first true WYSIWYG, HTML5-based Web editor and reporting template designer for ASP.NET.

Give your users an MS Word compatible editor to create powerful reporting templates anywhere - in any browser on any device.

Download your 30-day trial version and test a live demo today:
www.textcontrol.com/html5

Visual Studio Partner

the new TEXT CONTROL

Now, you can start debugging. Relaunch the package. When the Experimental Instance comes up, hit Ctrl+1. You should get an instance of Notepad.

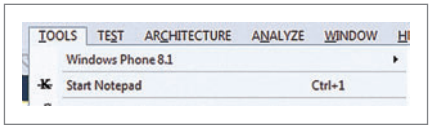


Figure 4 **The Command to Start Notepad**

Keep the UI Responsive

The MenuItemCallback method as it currently exists doesn't block the UI thread. You can still start Notepad with the Start Notepad command (or Ctrl+1). You can also still do things in the Visual Studio IDE. You can move the window around, click other commands on the menus and so on. Imagine your tool needs to complete its work before you exit the handler. In other words, you need to call Process.WaitForExit.

Unfortunately, if you call WaitForExit on the Visual Studio UI thread, as MenuItemCallback is doing at this point, the whole Visual Studio UI freezes. Check this out in action. In the MenuItemCallback method, call WaitForExit:

```
private void MenuItemCallback(object sender, EventArgs e)
{
  Process proc = new Process();
  proc.StartInfo.FileName = "notepad.exe";
  proc.Start();
  proc.WaitForExit();
}
```

Start debugging, and when the experimental instance comes up, hit Ctrl+1. You should see the instance of Notepad. Now, try moving the Visual Studio window (the experimental one). It won't move. You can't do anything else in the Visual Studio UI, either. You may even see the popup that says, "Microsoft Visual Studio is Busy." Clearly, this is something you need to fix.

> Learning how to create extensions lets you make the Visual Studio experience truly your own.

Fortunately, the fix is simple. If you have a tool or a process that will take a noticeable amount of time to complete, wrap it in a Task in MenuItemCallback, like so:

```
private void MenuItemCallback(object sender, EventArgs e)
{
  ThreadHelper.JoinableTaskFactory.RunAsync(async delegate
  {
    Process proc = new Process();
    proc.StartInfo.FileName = "notepad.exe";
    proc.Start();
    await proc;
  });
}
```

Now you should be able to run your tool and work in Visual Studio at the same time.

Clean up Your Experiment

If you're developing multiple extensions, or just exploring outcomes with different versions of your extension code, your experimental environment may stop working the way it should. In this case, you should run the reset script.

This script is called Reset the Visual Studio 2013 Experimental Instance and it ships as part of the Visual Studio 2013 SDK. This script removes all references to your extensions from the experimental environment so you can start from scratch. You can get to this script in one of two ways:

- From the desktop, find Reset the Visual Studio 2013 Experimental Instance
- From the command line, run the following:

```
<VSSDK installation>\VisualStudioIntegration\Tools\Bin\
  CreateExpInstance.exe /Reset /VSInstance=12.0 /RootSuffix=Exp && PAUSE
```

Deploy Your Extension

Now that your tool extension is running as it should, it's time to think about sharing it with your friends and colleagues. That's easy, as long as they have Visual Studio 2013 installed. All you have to do is send them the .vsix file you built. Be sure to build it in Release mode.

You can find the .vsix file for this extension in the StartNotepad bin directory. Assuming you've built the Release configuration, it will be in \D:\Code\StartNotepad\StartNotepad\bin\Release\StartNotepad.vsix.

To install the extension, the user needs to close all open instances of Visual Studio, then double-click the .vsix file to bring up the VSIX Installer. The files are copied to the %LocalAppData%\Microsoft\VisualStudio\12.0\Extensions directory.

When the user brings up Visual Studio again, he'll find the StartNotepad extension in Tools | Extensions and Updates. He can go to Extensions and Updates to uninstall or disable the extension.

Learning how to create extensions lets you make the Visual Studio experience truly your own. It also lets you share your best productivity enhancements and features with the community. Microsoft also loves it when you publish your extensions.

Wrapping Up

This article has described just a small part of what you can do with Visual Studio extensions. To find out more about Visual Studio extensions in general, see the "Integrate Your App or Service with Visual Studio" page at bit.ly/1zolt59.

For a more in-depth view, see the MSDN Library articles under the "Extending Visual Studio Overview" section at **bit.ly/1xWoA4k**. And for some good code samples to use to build off this knowledge, check out the collection of samples on the MSDN samples gallery at **bit.ly/1xWp5eD**. Once you've built and shared a few extensions, you can truly customize your Visual Studio environment. ■

Susan Norwood *has worked at Microsoft, mostly writing about the Visual Studio SDK. She has helped many people get their tools integrated into Visual Studio.*

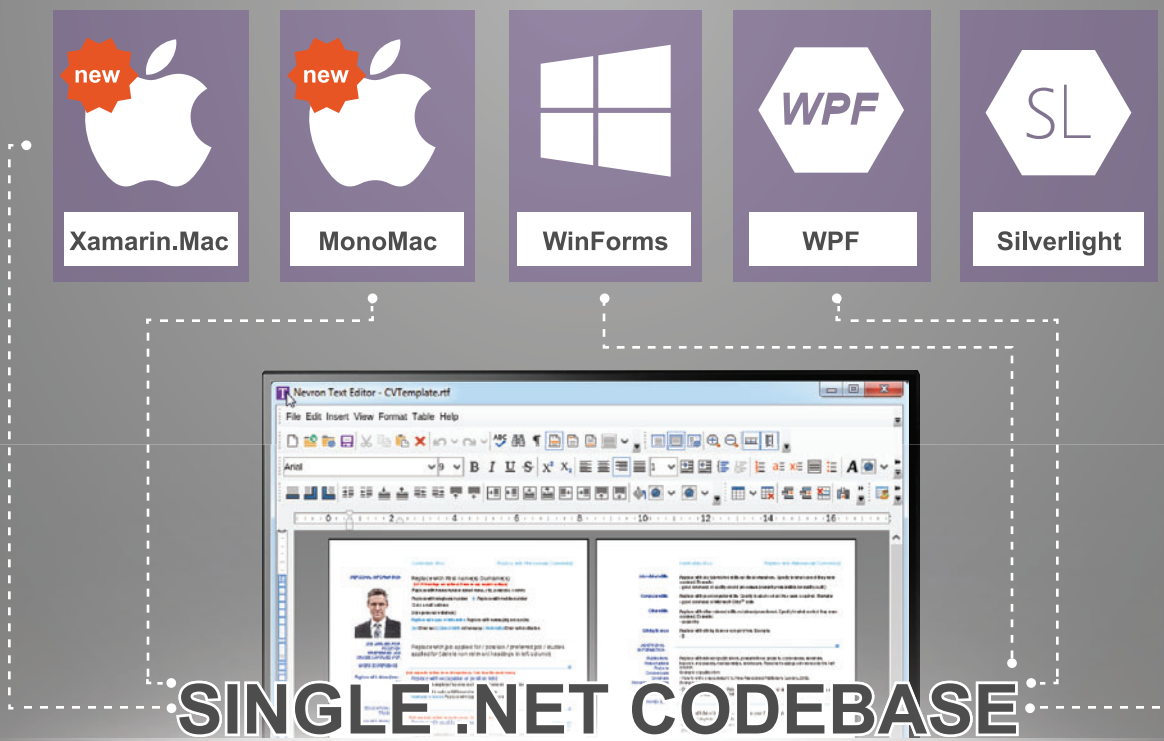
Doug Erickson *has worked as a developer and technical writer at Microsoft for 13 years. He looks forward to what developers will do with Visual Studio Community to develop apps and games for all platforms.*

Thanks to the following Microsoft technical experts for reviewing this article: *Anthony Cangialosi and Ryan Molden*

NEVRON
OPEN VISION for .NET

www.nevron.com

Write Once, Run Everywhere



Nevron Open Vision is the first suite of .NET controls for building cross-platform user interfaces and applications from a single code base.

Nevron Open Vision includes:

- NOV WIDGETS for .NET
- NOV TEXT EDITOR for .NET
- NOV GAUGE for .NET
- NOV BARCODE for .NET

COMING SOON:

- NOV CHART for .NET
- NOV DIAGRAM for .NET
- NOV GRID for .NET

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

Azure SDK 2.5 for .NET and Visual Studio 2015 Overview

Saurabh Bhatia and Mohit Srivastava

The Azure SDK for .NET makes it easy for your .NET-based cloud application to consume Microsoft Azure resources. Moreover, Visual Studio is integrated into the cloud development experience. You can view and manage cloud resources directly from Visual Studio, add cloud services to your projects, and publish to the cloud from the IDE.

This article gives a quick tour of some of the new features in Azure SDK 2.5 for .NET. Many of the features shown here support Visual Studio 2012 and Visual Studio 2013, but some require Visual Studio 2015 Preview, where noted.

The flow of this article is intended to follow a typical development workflow, while focusing on the new features of Azure SDK 2.5:

- Sign in to Visual Studio and Azure
- Create a cloud project
- Add cloud services and features, such as Azure Storage, WebJobs, and single sign-on
- Enable diagnostics and code analysis

This article refers to the Preview version of Visual Studio 2015. All information is subject to change.

This article discusses:

- Signing in to Visual Studio and Microsoft Azure
- Creating a cloud project
- Adding cloud services and features
- Enabling diagnostics and code analysis

Technologies discussed:

Visual Studio 2015 Preview, Azure SDK 2.5, Microsoft .NET Framework

Sign in to Visual Studio and Azure

When you launch Visual Studio, you're prompted to sign in with a Microsoft account. In Visual Studio 2015 Preview, if the account is also associated with your Azure Subscription, you're automatically signed in to Azure. To view your Azure resources, select Server Explorer from the View menu.

Visual Studio 2015 Preview lets you sign in with multiple accounts at once. For example, you might have a Microsoft account associated with an MSDN subscription, and also sign in with your organization account for your company. Server Explorer shows the resources for all of your accounts.

To add another account, right-click on the Azure node in Server Explorer and choose Manage Subscriptions. This brings up a dialog that shows the accounts you've added to Visual Studio. To add a new account, click Add an account. In the Manage Subscriptions dialog, you can also filter which Azure resources appear in Server Explorer by unchecking accounts, subscriptions, regions or certificates.

Another nice improvement is that you can now view your SQL Databases in Server Explorer without importing a certificate. Simply sign in to your Azure account and view your SQL Databases in Server Explorer. (This feature is also available in Visual Studio 2013 Update 4.)

QuickStart Templates

To get started with Azure quickly, you can use the new QuickStart templates. These templates provide sample code, showing how to use the Azure SDK and other libraries to interact with Azure Services. Find the QuickStart templates under File | New | Project | C# | Cloud | QuickStarts.

In the current release, there are QuickStart templates for Azure Storage (Blobs, Tables, Queues), DocumentDB, Azure Redis Cache Service, Azure Media Services, Azure WebJobs and Service Bus (Topics and Queues). There are also QuickStarts that show how to use the Microsoft Azure Management Libraries (MAML) to programmatically manage Azure resources.

Creating Resource Groups and Cloud Deployment Projects

A typical cloud app might use several different Azure resources. For example, an app might be hosted in an Azure Website, use a SQL Database for relational data, use Azure Storage for blobs or queues, and use Azure Redis Cache for low-latency data access.

Resource groups are a way to manage all of your resources as a logical group. With Azure SDK 2.5, you can create and deploy a set of resources in a resource group, using the new Cloud Deployment Project template. Currently, there are templates for:

- Azure Website
- Azure Website + SQL
- Azure Website + SQL + Redis Cache

In future releases, more templates will be added for application scenarios that use other Azure features such as networking, storage and virtual machines. Select the template closest to the solution you're building.

Here's how it works. In Visual Studio, select File | New | Project | Cloud | Cloud Deployment Project. Name the project MyAzureCloudApp and click OK. Then select from the list of common Azure Gallery templates, as shown in **Figure 1**.

For example, select the Azure Website template and name the project MyAzureCloudApp. After you select the template, you're prompted for an ASP.NET project type (Web Forms, MVC, Web API and so on). Visual Studio creates a solution with two projects:

- MyAzureCloudApp is the ASP.NET application.
- MyAzureCloudApp.Deployment is the deployment project.

The deployment project, shown in **Figure 2**, includes the following files:

- WebSiteDeploy.json: A deployment template.
- WebSiteDeploy.param.dev.json: Template parameters.
- Publish-AzureResourceGroup.ps1: A Windows PowerShell script that can be used to deploy your resources to Azure.

Together, the deployment template (WebSiteDeploy.json) and the parameters file (WebSiteDeploy.param.dev.json) specify the details for deploying and provisioning resources.

From Visual Studio, it's easy to deploy your resources to Azure. In Solution Explorer, right-click the deployment project, and select Deploy | New Deployment. In the New Deployment dialog, select Resource Group. This brings up the Deploy to Resource Group dialog.

To create a new resource group, click on the Resource group combo box and select Create New. Name the resource group, select a region, and click Create. Clicking the Create button provisions your Azure Resource Group, but does not spin up any Azure resources yet. (That will happen when you deploy.)

Next, click Edit Parameters to edit the parameters for the deployment template, such as Web site name, Web hosting plan and Web site location. If any required values are missing, they're shown with a red warning icon. When you click Save, the parameter values are saved back to the WebSiteDeploy.param.dev.json file. The next time you deploy, you don't need to reenter this information. Being able to customize and store the local JSON reduces the chance of errors when redeploying the resources.

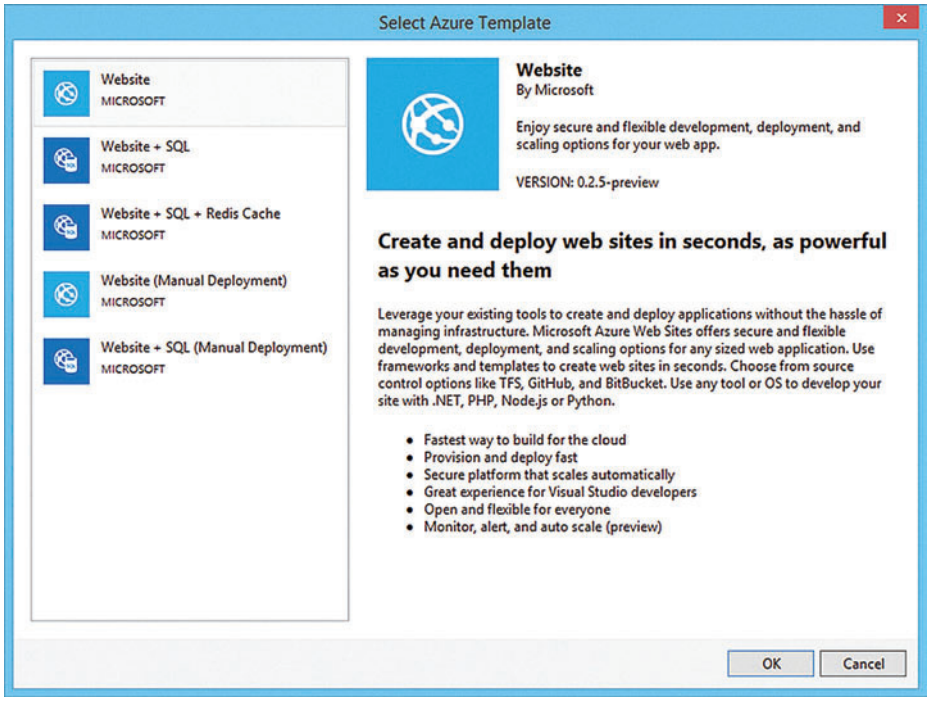


Figure 1 **Azure Galley Templates**

Once all the parameters have been specified, click Deploy to provision resources and deploy the application. You can see detailed progress in the Output window in Visual Studio. You can also view the resource group and the deployed resources in the new Azure Management Portal.

The deployment template and parameter files make it easy to work in a cloud lifecycle pattern and treat configuration as code. For example, you could create several parameter files, such as *.test.json and *.staging.json, so that a single template can deploy to multiple environments for testing, staging and so on. Visual Studio has

a built-in JSON editor, and includes IntelliSense based on the published JSON schemas, which makes it easy to edit the JSON files.

Enterprise Single Sign-On

A common task for business applications is to configure the application for enterprise single sign-on (SSO) using Azure Active Directory. In Visual Studio 2015, you can do this easily at any point during development. Right-click the project in Solution Explorer and select Configure Azure AD Authentication.

This will bring up a wizard to guide you through the process of configuring your application to work with Active Directory. Specifying an Active Directory Domain in the wizard will register your application with that Active Directory and configure your application to prompt for sign-in. Registering your application in a domain is a great way to get started in a dev-test environment. You can easily reconfigure the application's config files to use other domains as you move from one environment to the next in the application's lifecycle. You can find out more about this feature in the Azure Active Directory Overview video at bit.ly/1xCRpoc.

Add Connected Services

Once you've created a cloud application, Visual Studio 2015 makes it easy to incorporate additional cloud services into your app, including Azure Storage, Azure Mobile Services, Office 365 or Salesforce.

> Azure WebJobs provide an easy way to run scripts or programs as background processes on Azure Websites.

Here's how you connect the ASP.NET application to an Azure Storage account. In Solution Explorer, expand the Web application project. Next, right-click the References node and select Add Connected Service. Select Azure Storage from the list of services and click Configure.

Visual Studio shows a list of your existing storage accounts. Choose one of these, or provision a new one. Then click Add to connect to that storage account from your application. This adds Azure Storage references using the latest NuGet packages, and adds a connection string for your selected storage account in web.config. Visual Studio also shows a Getting Started page to help you get started with Azure Storage. To revisit the Getting Started page, right-click the GettingStarted.html file in Solution Explorer and choose View in Browser.

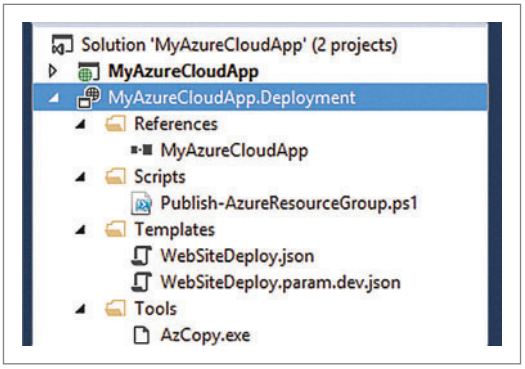


Figure 2 **Deployment Project Files**

Blob Folders in Storage Explorer

Blob storage is persistent file storage for the cloud. A blob can be any type of text or binary data, such as a document, media file or application installer. Blob folders are a way to group blobs into logical units. This is particularly useful if you're using one blob container to store all your information. In terms of implementation, a blob folder is really just a prefix for the blob name, delimited with the / character. For example, if you have a blob named file.png inside a folder named pictures, the URL for that blob will be in the following format:

```
https://<storageaccountname>.blob.core.windows.net/<blobcontainername>/
pictures/file.png
```

Starting with this release, you can create blob folders from inside Visual Studio, and navigate through the contents of blob folders.

In Server Explorer, expand the Storage node. Under this node are your storage accounts. Expand [storage account] | Blobs | [container name]. Right-click the container name and select View Blob Container. You can also create new containers from Server Explorer.

Navigate through folders by double-clicking on a folder to go inside it, and using the up arrow to go up a level. You can create a new folder when you upload a blob to Azure Storage using the upload button.

WebJobs

Azure WebJobs provide an easy way to run scripts or programs as background processes on Azure Websites. You can upload and run executable files such as .cmd, .bat, .exe (.NET), .ps1, .sh, .php, .py, .js and .jar. These programs run as WebJobs on a schedule (cron) or continuously. Azure WebJobs and the WebJobs SDK are now Generally Available. To learn more about WebJobs, see aka.ms/webjobs.

You can also add a WebJob project to an existing Web application. In Solution Explorer, right-click the Web application project and click Add | New Azure WebJob Project. As **Figure 3** shows, you can select whether you want the WebJob to run continuously, on a fixed schedule or on demand. For scheduled WebJobs, select the start and end times and the frequency.

The Program.cs file for an on-demand or scheduled job contains the following code for executing the WebJob explicitly:

```
static void Main()
{
  var host = new JobHost();
  host.Call(typeof(Functions).GetMethod("ManualTrigger"), new { value = 20 });
}
```

Continuous WebJobs get a different Program.cs file, which includes a call to start the WebJob host and block it from exiting:

```
static void Main()
{
  var host = new JobHost();
  // The following code ensures the WebJob will be running continuously
  host.RunAndBlock();
}
```

For more information about creating and deploying WebJob projects, see bit.ly/1pGVplo.

With Azure SDK 2.5, your WebJobs now appear in Server Explorer. They're listed under the Azure Website where they're deployed, and

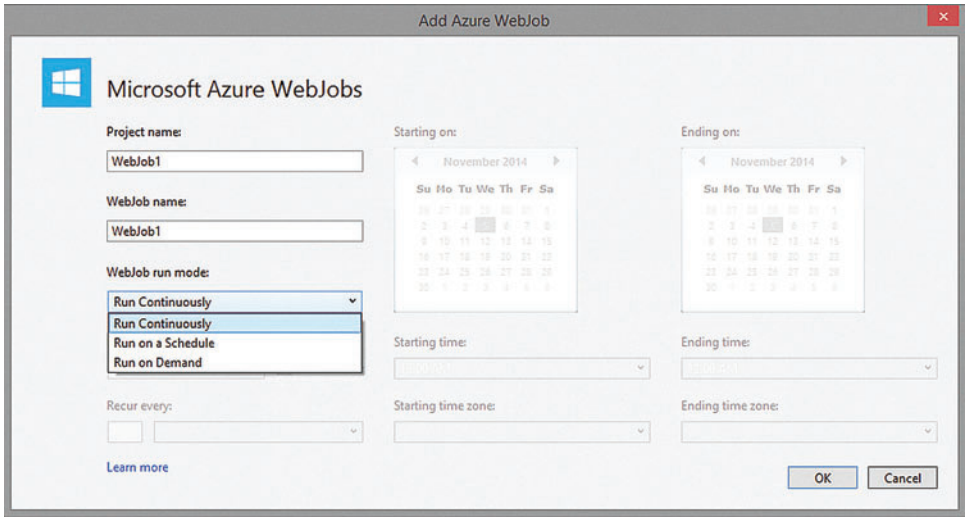


Figure 3 **Adding an Azure WebJob to a Project**

are grouped by run mode (Continuously, On Demand or Schedule). By right-clicking a WebJob in Server Explorer, you can open the dashboard for that WebJob. You can also run an on-demand job, or stop and start continuous jobs.

Continuous WebJobs now support remote debugging. Once a continuous WebJob has started, you can attach the Visual Studio debugger to the WebJob running in the cloud. To attach the debugger, right-click the WebJob in Server Explorer and select Attach Debugger. By placing a breakpoint and attaching the debugger, you can be stepped through your code just as if it were running locally in Visual Studio.

HDInsight

The SDK includes new tools in Server Explorer that make you more productive working with your Big Data using HDInsight hosted Hadoop services and using Hive to query your datasets.

Like any other Azure service, you can browse the HDInsight resources in your subscription and do common dev-test tasks. You can use the Server Explorer to browse your Hive tables, examine schemas and even query the first 100 rows of a table. This helps you understand the shape of the data with which you're working.

Also, there's tooling to create Hive queries and submit them as jobs. Use the context menu against a Hadoop cluster to immediately begin with "Write a Hive Query" for quick scripts. The Job Browser tool helps you visualize the job submissions and status. Double-click on any job to get a summary and details in the Hive Job Summary window. In the example in **Figure 4**, the table is queried with geographic info to find the count of all countries and then sort by country.

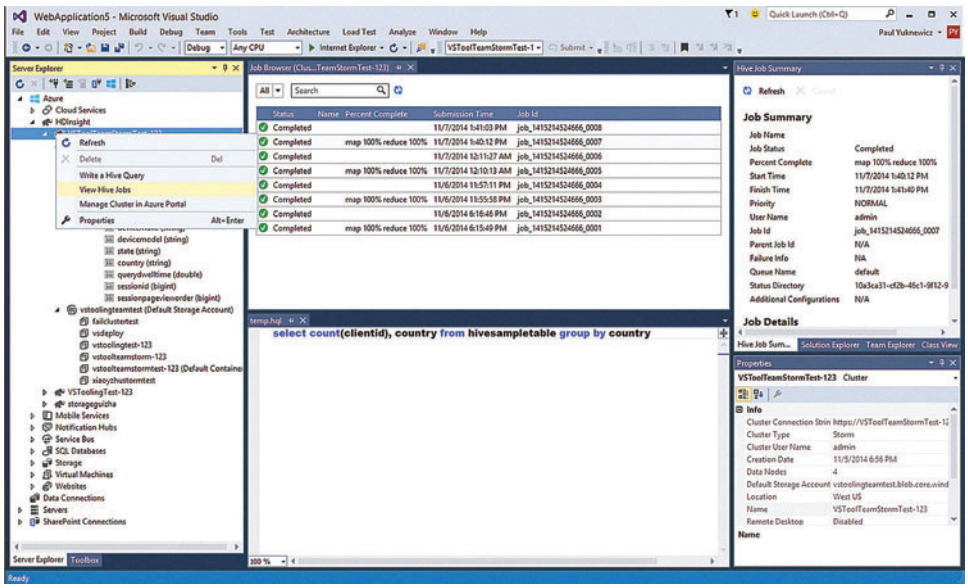


Figure 4 **Viewing Hive Jobs in Visual Studio Server Explorer**

The backing blob storage used by HDInsight is listed in Server Explorer. You can navigate to any blob container and open it to work with files. The backing store is associated with the Hadoop cluster during cluster creation in the Azure dashboard. Management of the Hadoop cluster is still performed in the same Azure dashboard.

For more complex script development and lifecycle management, you can create Hive projects within Visual Studio. The New Project dialog now has an HDInsight Template category. A helpful starting point is the Hive Sample project type. This project is pre-populated with a more complex Hive query and sample data for the case of processing Web server logs.

Improved Diagnostics Logging for Cloud Services and Virtual Machines

Azure SDK 2.5 includes improvements for diagnostics logging in both Azure Cloud Services Platform as a Service (PaaS) and Virtual Machines Infrastructure as a Service (IaaS).

For example, you can now perform structured and semantic logging using Event Tracing for Windows (ETW) event sources. ETW event sources and event IDs allow you to define logical events and stages of the application workflow. Then you can track events across multiple tiers, to help diagnose issues in your application workflow. You can also collect crash dumps. By default, the most common processes for cloud roles are pre-selected for crash dump collection. One very nice feature is that you can update the diagnostic configuration for a running service, after the cloud service is published.

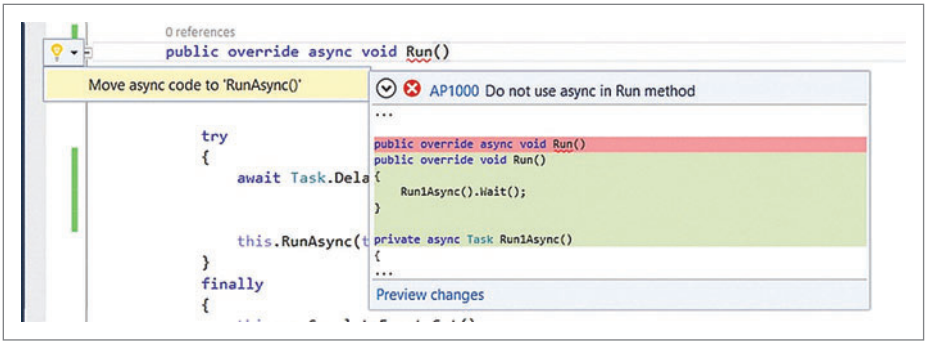


Figure 5 **Code Analysis with a Suggested Code Fix**

Microsoft Code Analysis for Azure Preview

What if you could find problems in your code as you type? Not just compilation errors, but subtler coding issues.

The .NET Compiler Platform in Visual Studio 2015 Preview provides new code analysis capabilities that detect code issues in real time as you write your code. Leveraging these capabilities, the Azure team has put together a small set of code analysis rules for Azure. These rules identify issues in your code and, when possible, recommend automatic fixes. They're available through the Microsoft Code Analysis for Azure NuGet package.

To install the package, right-click your solution and choose Manage NuGet Packages for Solution. Make sure the Include Prerelease checkbox is checked and search for Azure Code Analysis.

This should find the Microsoft.VisualStudio.Azure.CodeAnalysis package. Select projects in your solution and click Install to enable code analysis for those projects. Once an analyzer is added to the project, code analysis is always running. To run code analysis on the entire solution, build the solution. Any code analysis warnings or errors will automatically show up in the error list as you make edits to a source file.

As you code, any line that has an issue will display a red underline for errors or a green underline for warnings. Hover over the underlined code to get more information. Click on the light bulb next to the issue to get additional details on fixes. For example, **Figure 5** shows a code issue with the Run method of a worker role being declared as async. This is a common scenario where you might want to perform some asynchronous operations in a worker role. However, declaring the Run method itself as async will cause the role to restart in an infinite loop. The code analysis detects the issue and offers an automatic fix, which refactors the Run method and moves the async operations to another method.

Not all code issues have automatic fixes. If no automatic fix can be provided, the light bulb menu will point you to documentation describing how to fix the issue.

The code analysis rules provided in the NuGet package are just an initial set of rules designed to showcase the code analysis capabilities in the new .NET compiler. As the team receives feedback, we will provide more rules and continue to iterate on the experience.

Figure 6 **Summary of New Features in Azure SDK 2.5 for .NET Framework**

Feature	Visual Studio 2015 Preview	Visual Studio 2013/2012
Connect Visual Studio to Multiple Azure Accounts	x	
QuickStart Templates	x	x
Cloud Deployment Projects	x	x (Visual Studio 2013 only)
Add Connected Service Dialog	x	
Blob folders in Storage Explorer	x	x
WebJobs: Server Explorer Support and Remote Debugging	x	x
HDInsight: Server Explorer and Hive Query Support	x	x
Configure Enterprise Single Sign-On (SSO)	x	
Diagnostics Improvements	x	x
Code Analysis for Azure (Preview)	x	

Wrapping Up

Figure 6 shows a summary of the new features in Azure SDK 2.5 for .NET. Azure SDK 2.5 and Visual Studio 2015 Preview make it easier than ever to get started developing rich cloud applications.

If you don't already have an Azure account, you can sign up for a free trial and start using all of the features discussed here. There are two ways to get started with the SDK:

- Install the SDK locally: Download from the Azure .NET Developer Center (bit.ly/1uR4Z5h)
- VM Image in Azure: The fastest way to get started with Azure SDK 2.5 is by using a VM from the Azure Virtual Machine gallery. You can create a VM that has Azure SDK 2.5 installed, with either a Visual Studio 2015 Preview or Visual Studio Community 2013 image. Both of these images are available to all Azure customers. If you're an MSDN subscriber, you can take advantage of additional Visual Studio 2013 images based on Windows 8.1 and Windows Server 2012, which have been updated to Visual Studio 2013 Update 4 and Azure SDK 2.5.

You can then visit the Azure .NET Developer Center (bit.ly/1uR4Z5h) to learn more about how to build apps with it. ■

Saurabh Bhatia *is a program manager at Microsoft working on Visual Studio Azure Tools. Before this, Bhatia worked on developer tools for Office 365 APIs and apps for Office and SharePoint. You can reach him at saurabh@microsoft.com.*

Mohit Srivastava *is a lead program manager at Microsoft responsible for Azure and Web Developer Tools and Services. In prior roles, he worked "lower in the stack" on core Azure and Windows services and "higher in the stack" as cofounder at a microblogging startup. You can reach him at mohisri@microsoft.com*

Thanks to the following Microsoft technical experts for reviewing this article: *Brady Gaster and Michael Wasson*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

vslive.com

2015 Dates Announced

Visual Studio Live! is hittin' the open road on the ultimate code trip! For 22 years, we have helped tens of thousands of developers navigate the .NET Highway, featuring code-filled days, networking nights and the best in independent training:

- Multi-track Events
- Focused, Cutting-edge .NET Education
- Covering the Hottest Topics
- Relevant and Useable Content
- Expert Speakers, Including Many Microsoft Instructors
- Interactive Sessions
- Discuss Your Challenges
- Find Actionable Solutions

MAKE PLANS TO JOIN US IN 2015!

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA

NAVIGATE THE .NET HIGHWAY

NOW WITH 6 Locations to Choose From!

New York

The Code That Never Sleeps
September 28 – October 1
NY Marriott at Brooklyn
Bridge Brooklyn, NY

NEW YORK Code Trip

AUSTIN Code Trip

Austin

Don't Mess with Code
June 1 - 4
Hyatt Regency
Austin, TX

ORLANDO Code Trip

Orlando

Code in the Sun

PART OF LIVE! 360
Tech Events with Perspective

SharePoint Live!
SQL Server Live!
Modern Apps Live!
TechMentor

November 16 – 20
Loews Royal Pacific Resort
Orlando, FL

Push Notifications to Cordova Apps with Microsoft Azure

Glenn Gailey

As mobile devices proliferate and become a ubiquitous part of our daily existence, consumers expect more and more from their mobile apps. Whereas it was once acceptable for your mobile app to simply connect to the cloud to get the latest stock quote or to store user-generated data, users now expect your app to alert them when events occur—such as when their stock hits a certain price or when their team scores. Today, all native device platforms support push notifications, which enables you to interact with your users nearly in real time.

I'll demonstrate the basics of leveraging the Microsoft Azure platform to send push notifications to apps developed by using the Visual Studio Tools for Apache Cordova. I'll start with an existing Cordova sample app project, create and configure the needed services, and then modify the app to register for push notifications, all in Visual Studio. I'll focus on sending push notifications to an Android app, but the completed Cordova app sample supports the Android and iOS platforms. I'll show how to use the Android emulator, not because it's great (it's not), but because it does support push notifications. I recommend using an Android device rather than the emulator if you can. (Note that sending push notifications to an iOS device requires both a physical device and an Apple Developer account, and that the Mobile Services plug-in doesn't currently support Windows Phone.)

To use the Visual Studio 2013 tooling highlighted in this article, you need to have Update 4 installed. You also need Visual Studio Tools for Apache Cordova (aka.ms/uopu9r), which is also included in Visual Studio 2015 Preview. For a more comprehensive overview of these tools, see the article, "Write Hybrid Cross-Platform Apps in Visual Studio with Apache Cordova" (p. 4), in this issue.

Finally, you'll need an Azure account. If you don't have an account, you can sign up for an Azure trial and get up to 10 free mobile services that you can keep using even after your trial ends. For details, see the Azure Free Trial page (aka.ms/qepjcc).

This article discusses a prerelease version of Visual Studio Tools for Apache Cordova. All related information is subject to change.

This article discusses:

- How push notifications work
- Benefits of using Microsoft Azure for notifications
- Extending an existing Todo sample
- Creating back-end Azure services
- Provisioning push notifications
- Using the Android emulator
- Generating and testing notifications

Technologies discussed:

Visual Studio Tools for Apache Cordova, Microsoft Azure

Code download available at:

aka.ms/kop2o2

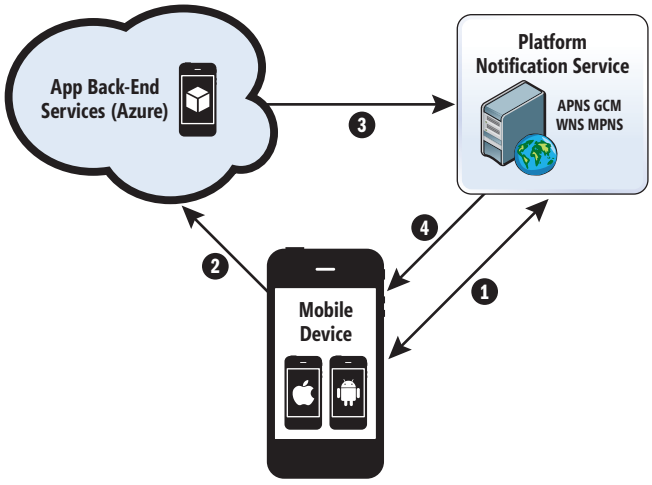


Figure 1 **Push Notification Services Architecture**

How Push Notifications Work

Regardless of the native device platform, all push notification architectures are REST-based, whereas the details are platform-specific. **Figure 1** shows the service architecture required to send push notifications using Azure.

As you can see in **Figure 1**, the key to a push notification architecture is the platform notification service (PNS). This device-platform-specific service is responsible for sending messages to the native device, and it's actually the device that handles the notification, not the app. You can see in **Figure 1** that a back-end service is also required to send notifications to the device (this is where Azure comes in). In most cases, the back-end service must authenticate with the PNS to be able to send notifications, which is done to prevent malicious use of the PNS. The basic process of sending a push notification goes like this:

1. The mobile app contacts the PNS to request a handle, which the service uses to identify the device. These handles aren't durable and need to be managed and regularly refreshed.
2. The app sends the handle to its back-end service, where it's stored, usually with other data that enables the service to correctly target the device. As you'll see, Azure is great for this.
3. When sending a push notification, the back-end service sends a request to the notification service, which includes the handle and the message, in a platform-specific format (usually JSON or XML).
4. The PNS authenticates the request and sends the requested message to the specific device, which is identified by the provided handle.

Each native device platform has its own PNS. This means when you create a cross-platform app using Cordova and want to send messages, you must provision your app with one or more of the following services:

- Apple Push Notification Service (APNS) for iPad and iPhone devices
- Google Cloud Messaging service (GCM) for Android devices
- Windows Notification Service (WNS) for Windows devices
- Microsoft Push Notification Service (MPNS) for Windows Phone devices

The push notification plug-in for PhoneGap (aka.ms/xltv38) is used to register your Cordova app with the native device's PNS. However, even with this convenient plug-in, you still have to both manage registrations for your app on the various platforms you support and send a given message across all of those disparate platforms. On top of all of this, you still need to create and maintain a back-end service that sends out the notifications. Fortunately, Azure comes to the rescue by providing both a simplified way to register for and manage device registrations, as well as a convenient back-end service from which to send notifications to your Cordova app.

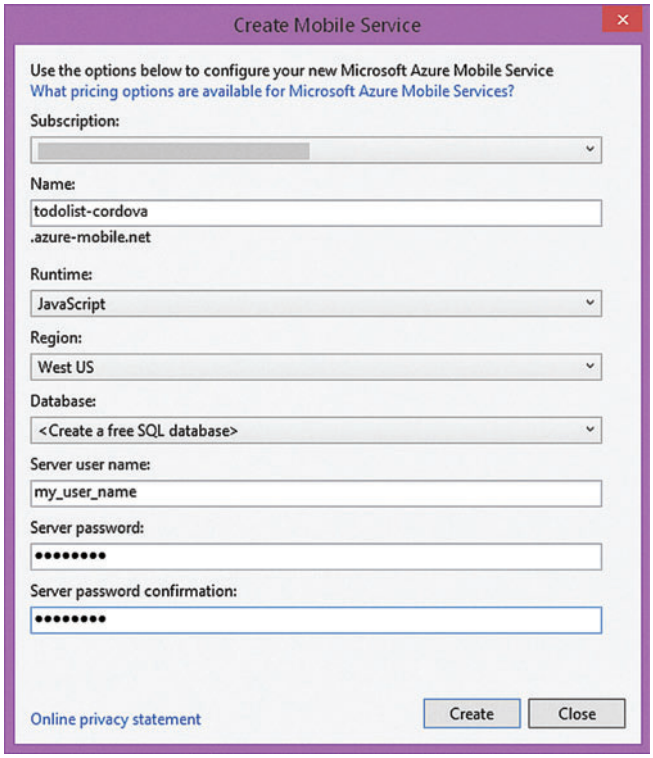


Figure 2 **The Create Mobile Service Dialog**

Why Use Azure for Push Notifications?

There are two services in Azure designed specifically to make it easy to send push notifications to your cross-platform mobile device apps: Azure Notification Hubs and Azure Mobile Services.

Notification Hubs A scalable, cross-platform solution for sending push notifications to mobile devices, Notification Hubs works well with Cordova apps. Notification Hubs manages the registrations with each PNS. More important, Notification Hubs lets you create template registrations so you can send messages to all registered devices, regardless of platform, with only a single line of code. You can also use tags to send targeted notifications only to devices with specific registrations. For more information about Notification Hubs, see the Azure Web site at aka.ms/nkn4n4.

Mobile Services Designed specifically to support mobile apps, Mobile Services integrates with Notification Hubs. When you create a new mobile service, a new notification hub is automatically provisioned for you. As of the writing of this article, even a mobile service running in the Free Tier allows 1 million pushes per month

Figure 3 **The Script Executed When Items Are Inserted into the TodoItem Table**

```
function insert(item, user, request) {
  // Execute the request and send notifications.
  request.execute({
    success: function () {
      // Create a template-based payload.
      var payload = '{ "message" : "New item added: ' + item.text + '" }';

      // Write the default response and send a template notification
      // to all registered devices on all platforms.
      push.send(null, payload, {
        success: function (pushResponse) {
          console.log("Sent push:", pushResponse);
          // Send the default response.
          request.respond();
        },
        error: function (pushResponse) {
          console.log("Error Sending push:", pushResponse);
          // Send the error response.
          request.respond(500, { error: pushResponse });
        }
      });
    }
  });
}
```

free of charge. In addition to push notifications, Mobile Services also provides simple storage for your app's data in Azure SQL Database and authentication with popular providers, such as Facebook, Google, Azure Active Directory, Microsoft Account and Twitter. For more information about Mobile Services, see the Azure Web site at aka.ms/az48v5.

Because Mobile Services integrates with Notification Hubs, you can register for push notifications using the Mobile Services client. There's no need to deal with the Notification Hubs access policies or connection strings.

As of this writing, the Mobile Services client only supports registration for iOS and Android apps. If you also need to support your Cordova app on Windows Phone using MPNS, see my blog post, "Push Notifications to PhoneGap Apps Using Notification Hubs Integration," at aka.ms/r197ab.

Get the TodoList Sample App

Mobile Services is about connecting your mobile apps—including Cordova-based apps—to Azure for storage, authentication and push notifications. As such, it doesn't really matter which app I use to demonstrate push notifications, just as long as it's a Cordova app. There are several Cordova-based Todo sample app projects published by the Visual Studio folks, each using a different JavaScript framework. You can find these Visual Studio-based Cordova samples at aka.ms/dxrr30.

Although AngularJS is arguably the more popular framework, the AngularJS version of the Todo sample demonstrates using the Mobile Services REST APIs for remote data storage. The BackboneJS sample uses the Mobile Services plug-in, so extending this sample to add push notifications is more straightforward. Thus, I chose to use the BackboneJS version of the sample (aka.ms/sffl1f). Regardless of the JavaScript framework you use for your app (if any), the same basic steps apply.

At this point, you should open the project's index.html file and delete or comment out the existing Mobile Services references, which look like this:

```
<script src="http://ajax.aspnetcdn.com/ajax/mobileservices/
  MobileServices.Web-1.1.3.min.js"></script>
<script src="services/mobile services/todolist-xplat/service.js"></script>
```

You should also probably delete the entire existing \services folder, which contains the original sample's service.js file. This will help to avoid confusion between multiple MobileServiceClient variables later on. In the next section, you'll use Visual Studio to create a new mobile service and connect it to your Cordova app. You'll also add the PushPlugin, device and console plug-ins to the project.

Create the Back-End Azure Services

Visual Studio makes it easy to provision the Azure services you need, and you can do it right from the IDE. Next, I'll show how to create a new mobile service right from Visual Studio (assuming you have an active Azure subscription).

In the Visual Studio Solution Explorer, right-click the project, then click Add | Connected Service. In the Services Manager, click Create service. Set the following fields in the Create Mobile Service dialog, shown in **Figure 2**, then click Create:

- **Subscription:** This is your Azure subscription.
- **Name:** This is the name of your new mobile service, which must be unique across Azure and which becomes part of the service's URL. A red X is displayed when the name you entered isn't available.
- **Runtime:** This app uses a JavaScript (Node.js) back end, which just makes more sense for JavaScript programmers than a .NET back end (ASP.NET Web API project).
- **Region:** The location of your mobile service—ideally in the same location as your SQL Database.
- **Database:** Either choose an existing database in the same region or create a new database, ideally a free one. When you choose an existing database, tables belong to the schema that's the mobile service name, so there's no risk to your current data.

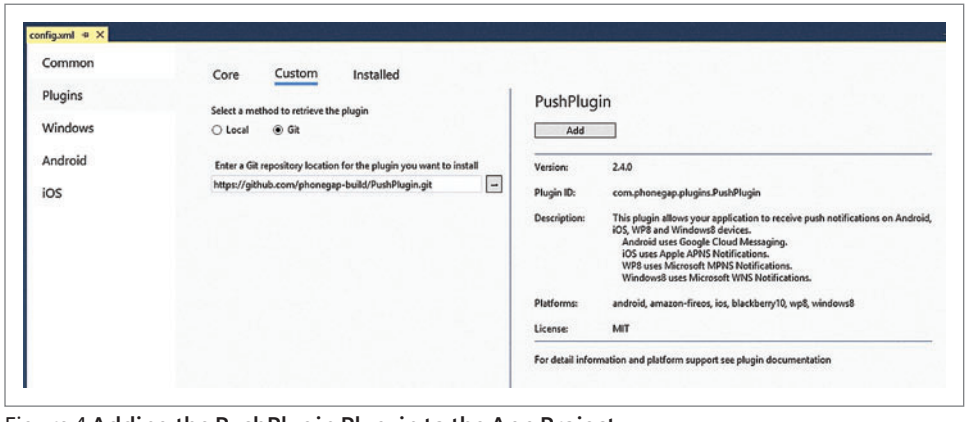
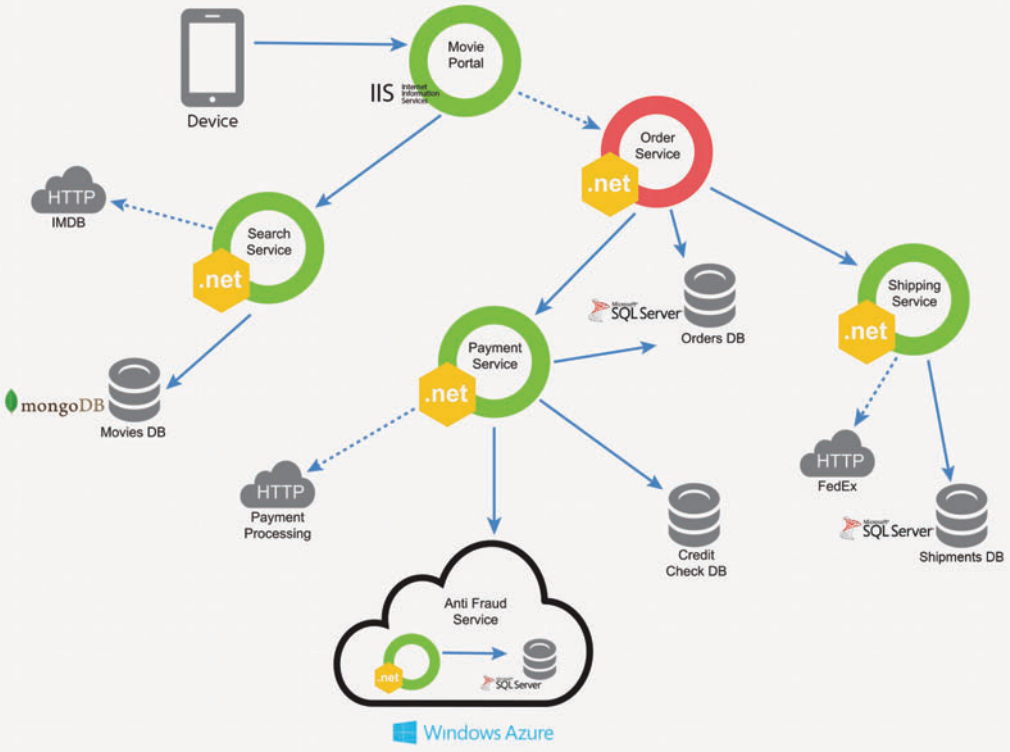


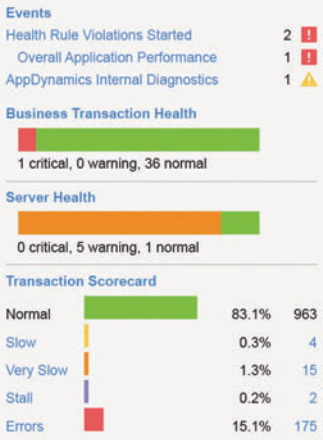
Figure 4 **Adding the PushPlugin Plug-in to the App Project**

APPDYNAMICS

There's nothing about .NET that AppDynamics doesn't see.

AppDynamics gives you the visibility to take command of your .NET application's performance, no matter how complicated or distributed your environment is.





When your business runs on .NET and SQL Server, count on AppDynamics to give you the complete visibility you need to be sure they are delivering the performance and business results you need — no matter how complex, distributed or asynchronous your environment, live 'in production' or during development.

See every line of code. Get a complete view of your environment with deep code diagnostics and auto-discovery. Understand performance trends with dynamic baselining. And drastically reduce time to root cause and remediation.

See why the world's largest .NET deployments rely on the AppDynamics Application Intelligence Platform. Sign up for a FREE trial today at **www.appdynamics.com/Microsoft**.

appdynamics.com

- **Server user name** and **Server password:** These are important credentials, so keep them safe. Mobile Services won't need them again, but you will when you want to use this database for something else, such as to support another mobile service.

When you click Create, a new mobile service is provisioned along with a new notification hub. Back in the Service Manager dialog, select the new service and click OK. This adds a version of the Mobile Service plug-in and the InAppBrowser plug-in, which Mobile Services uses for authentication, to the Cordova project. It also adds and opens a new code file, such as todolist-cordova.js, that defines the MobileServiceClient instance used to connect to the mobile service. Leave this file open; you'll make updates here later.

Figure 5 **Code that Registers with Azure for Push Notifications When the App Starts**

```
var GCM_SENDER_ID = '<your_project_number>'; // Replace with your own ID.
var mobileServiceClient;
var pushNotification;

// Create the Azure client register for notifications.
document.addEventListener('deviceready', function () {
  mobileServiceClient = new WindowsAzure.MobileServiceClient(
    'https://todolist-cordova.azure-mobile.net/',
    'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX');

  // Define the PushPlugin.
  pushNotification = window.plugins.pushNotification;

  // Platform-specific registrations.
  if (device.platform == 'android' || device.platform == 'Android') {
    // Register with GCM for Android apps.
    pushNotification.register(successHandler, errorHandler,
      {
        "senderID": GCM_SENDER_ID,
        "ecb": "onGcmNotification"
      });
  }
});

// Handle a GCM notification.
function onGcmNotification(e) {
  switch (e.event) {
    case 'registered':
      // Handle the registration.
      if (e.regid.length > 0) {
        console.log("gcm id " + e.regid);
        if (mobileServiceClient) {
          // Template registration.
          var template = "{ \"data\" : {\"message\":\"$(message)\"}}";
          // Register for notifications.
          mobileServiceClient.push.gcm.registerTemplate(e.regid,
            "myTemplate", template, null)
            .done(function () {
              alert('Registered template with Azure!');
            }).fail(function (error) {
              alert('Failed registering with Azure: ' + error);
            });
          }
        }
        break;
      case 'message':
        if (e.foreground) {
          // Handle the received notification when the app is running
          // and display the alert message.
          alert(e.payload.message);
          // Reload the items list.
          refreshTodoItems();
        }
        break;
      case 'error':
        alert('Google Cloud Messaging error: ' + e.message);
        break;
      default:
        alert('An unknown GCM event has occurred');
        break;
  }
}
```

Next, you need to create a new TodoItem table in your SQL Database that the app can use to store items. Open Server Explorer, expand the Azure node and then the Mobile Services node, right-click the mobile service and select Create Table. Type TodoItem for Table name, then click Create. A new TodoItem table is added under the mobile service node in Server Explorer. This node corresponds to a new SQL Database table, which is exposed as a new table in your mobile service.

At this point, you need to update the default insert script executed by Mobile Services when an insert is made into the new table. In Server Explorer, expand the TodoItem table you just added, open the insert.js file and replace the existing insert function with the code shown in **Figure 3**.

The code in **Figure 3** calls the send method on the push object to send a template-based push notification to all registered devices. Instead of passing a null to the first parameter (tags) as in this case, you can also pass a tag value or array of tags. When you use tags, Notification Hubs will only send notifications to a device whose registration contains one of those tags. For more on tags, see aka.ms/cwr4l4.

Finally, you need to navigate to the Azure Management Portal Web site so you can register your app's PNS credentials, which you'll create in the next section. In Server Explorer, right-click the node for your mobile service in Azure, then click Open in Management Portal and sign in to the portal. At this point, you should be in the quickstart tab (denoted by a symbol only) for your new mobile service. Click the Push tab and notice the name of the notification hub used by your mobile service at the top of the page. You'll need to know this if you have to troubleshoot your notification hub in Visual Studio (as you'll see later on). Scroll down to GCM settings at the bottom of the page. This is where you set the GCM API key once push notifications are provisioned.

Provision the Google Cloud Messaging Service

The most time-consuming part of configuring push notifications is usually provisioning the app with the PNS. Thankfully, this provisioning need only be done once for a given app and platform. For Android, you need a Google account to provision your Cordova app with GCM at the Google Developers Console Web site (aka.ms/wt80js). Once signed in, click Create Project, specify a Project Name and (optionally) your own unique Project ID, accept the terms, and then click Create. Once created, the new project is displayed. Write down the Project Number value shown at the top of the page; you'll need it later on.

Next, expand APIs & auth, click APIs, scroll down to Google Cloud Messaging for Android, enable it and accept the terms of service. Now, under APIs & auth, click Credentials, under Public API access click Create new Key, click Server key and then Create. Write down the API key value. Back in the Azure Management Portal, enter the API key under the "google cloud messaging settings" section, then click Save.

The API key is now set and Notification Hubs can now use GCM to send push notifications to your app. If you have trouble


Deploy deep into space with InstallAware 18
New InstallAware Direct Deploy Technology. Pushes any EXE without ANY client or server software.
Complete Unicode support, retaining full backwards compatibility with previous InstallAware projects.
Bullet-proof, depenency free; run the same setup on Windows XP all the way to Windows Server 2012 R2 x64!
Supports all Active Directory Domains.
Competitive upgrade discounts from InstallShield.
www.installaware.com
InstallAware

provisioning GCM, or if the Web site changes, you can always find the latest, detailed steps for provisioning GCM in the article, "How to Enable Google Cloud Messaging," on the Azure Web site at aka.ms/po7r8n. You can also find the equivalent steps for provisioning an iOS client with APNS in the article, "How to Enable Apple Push Notifications," on the Azure Web site at aka.ms/vooydc. Note that because Notification Hubs is an independent service, other back-end services can also use this registration to send notifications to your app.

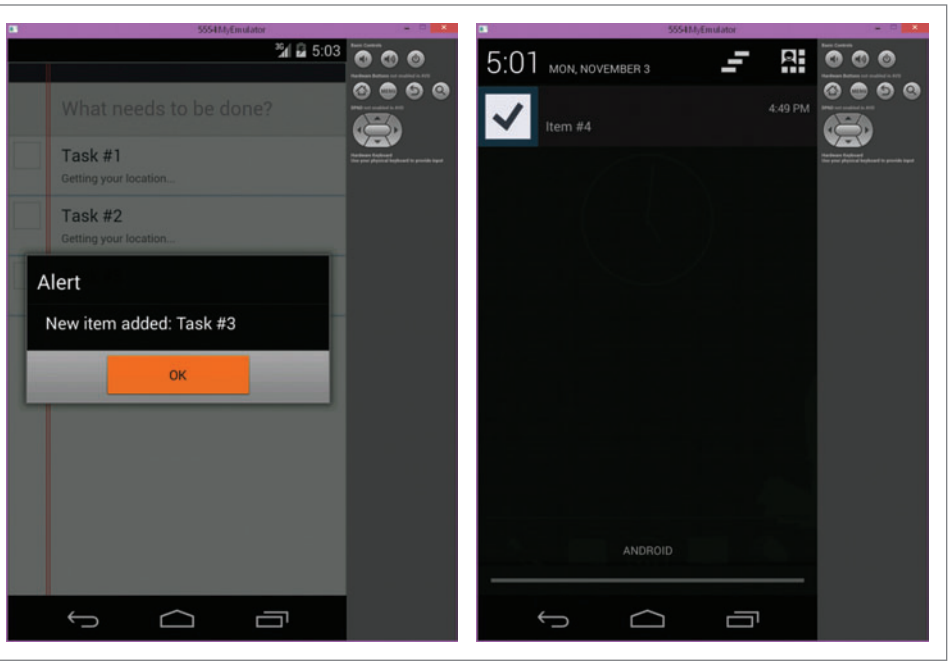


Figure 6 **Notification Messages as an Alert (Left) and Appearing in the Status Bar**

Update the Client to Register for Notifications

The last thing you need to do is to update the downloaded TodoList sample project to register for push notifications with Azure when the app starts. First, you need to add the required plug-ins. In Solution Explorer, double-click the config.xml project file to open it in the configuration designer and choose the Plugins tab. Under Core, add the Device, Console and Geolocation plug-ins. If the Azure Mobile Services plug-in has already been installed, click Remove to uninstall it—you need to install a more recent version that supports push notifications. Next, click Custom, select Git, enter the following URL to the Azure Mobile Services plug-in repository, and then click Add:

```
https://github.com/Azure/azure-mobile-services-cordova.git
```

This installs the most recent version of the plug-in that contains the Mobile Services client library. Repeat the previous step to add the PushPlugin to the project by using the following repository URL:

```
https://github.com/phonegap-build/PushPlugin.git
```

Figure 4 shows how simple Visual Studio now makes it to add any valid Cordova plug-in to your project.

Open the project file that was created when you added the Mobile Services connection to your project (the one with the name such as todolist-cordova.js, where todolist-cordova is the name of the service). Save a copy of the existing MobileServiceClient constructor, which contains the URL for your mobile service and application key (so they don't get overwritten), then replace this generated code with the code shown in **Figure 5**.

The code in **Figure 5** requests a registration ID from GCM, then uses the returned registration ID to either create or update a Notification Hubs registration. Note that **Figure 5** only contains GCM code required for Android, whereas the accompanying project download contains code for both Android and iOS. To get this code to run in your project, replace the MobileServiceClient constructor with the saved one that was added by the Visual Studio tools. Also, set GCM_SENDER_ID to the project number of your app in Google Developers Console.

Configure the Android Emulator for Push Notifications

Unlike the iOS simulator, you can receive push notifications on the Android emulator, provided you perform a few extra configuration steps. You basically need to use the Google APIs and add a Google account. When using a physical Android device, you can skip these steps and go to the next section to run and test your app. (Again, if possible, you should really try to run on a physical Android device—the emulator that ships with the Android SDK is excruciatingly slow, though it does work.)

From a command prompt, type "android avd," which displays the Android Virtual Device (AVD) Manager. You should have at least one AVD available, which is sometimes named MyEmulator. If you don't have an AVD, you need to create one. Select your AVD, click Edit, then set the Target to Google APIs and click OK. Click Start to run this emulator. On the emulator, click the apps icon, then click Settings. Scroll down to Accounts, click Add account, click Google, and add a new or existing account to the emulator. With the Google account added, you can receive notifications on the emulator.

Once you've added a Google account on the emulator, you should plan to keep the emulator running (unless you've enabled snapshots in the emulator options). Otherwise, the state of the device won't be maintained when you restart the emulator, and you'll have to re-add a Google account to be able to receive notifications again.

Run the App and Generate Push Notifications

The first time you run the app, a new registration is created in your notification hub. On subsequent startups, the existing registration is updated with the most recent registration ID returned by the GCM. With the Android emulator selected, press F5 to start and run the app. Visual Studio starts a Cordova build targeted at the Android platform, and the app is deployed to and started on the running emulator (it also starts the emulator, if needed).

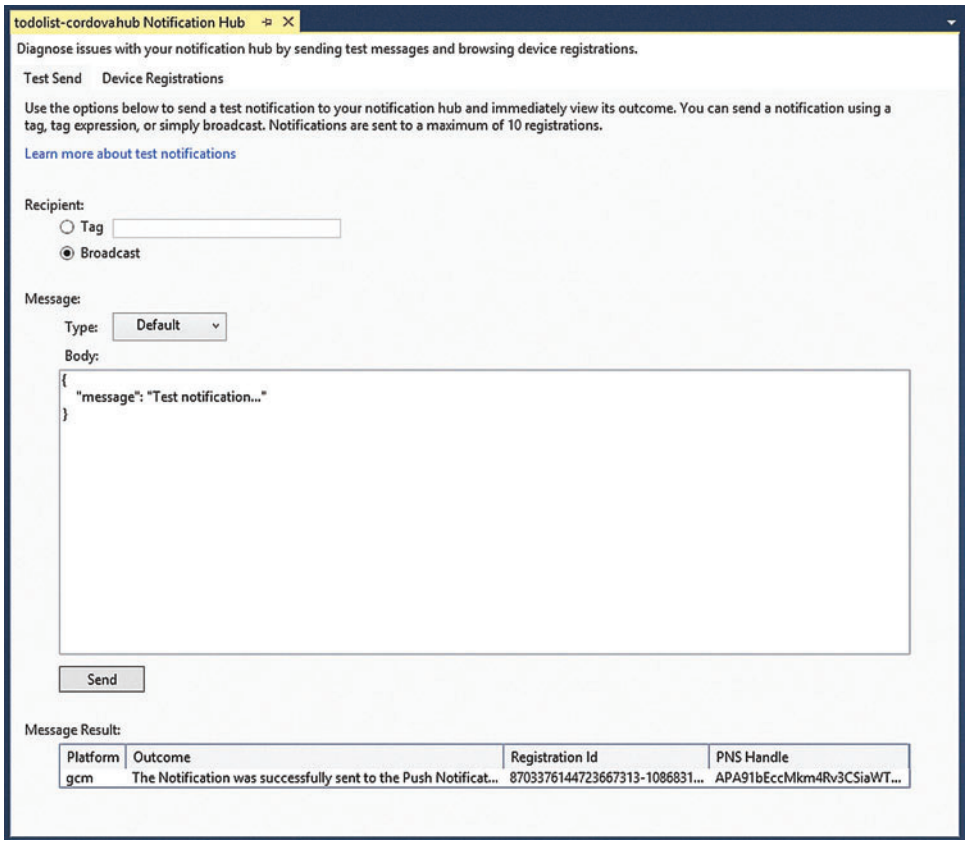


Figure 7 **Notification Hubs Diagnostic Page**

After the app loads, an alert is displayed after the app successfully registers (or updates the registration) for push notifications with Azure. When you add a new item, the onGcmNotification function (shown in **Figure 5**) handles the incoming message, when the app is running, and displays the message in an alert.

When the app isn't running, the notification ends up in the status bar. **Figure 6** shows the notification appearing as an alert (on the left) and in the status bar.

You can test this behavior by switching out of the app right after sending. Or, if there isn't enough time to do this, you can add a delay in the insert script before the send method is called. When you click the notification in the status bar, the app is started.

Another way to demonstrate this is by sending a test notification to your app.

Troubleshoot Registrations by Sending Test Notifications

Because several services and various registrations are involved, it can be tricky to troubleshoot push notifications. Fortunately, diagnostic functionality for Notification Hubs is built right into Visual Studio 2013. From the Server Explorer, expand Azure and then Notification Hubs, and you should see your notification hub listed. Double-click this item and you will see a diagnostic page open right in Visual Studio (see **Figure 7**). This page has the following two tabs:

- **Test Send:** This lets you compose and send messages using the notification hub. You can target specific native device registrations or, in this case, a template registration. When you click Send, the specific notification is sent; the result, as logged by Notification Hubs, is shown in Message Result.
- **Device Registrations:** This enumerates all registrations that currently exist in the notification hub. From this tab, you can delete registrations or update the tags collection for a registration. On this tab, you should see at least one registration, which was just added when you started your app.

Remember, when you send a test notification from Visual Studio and the app isn't running, the message shows up in the status bar. Had the app been running, an alert would've been raised instead (as was shown in **Figure 6**).

You can find much of the same push notification test functionality in the Notification Hubs tab at the Azure Management Portal (manage.windowsazure.com), but it's great to be able to test and do some basic debugging right in the Visual Studio IDE.

Simplified Notifications

To review, mobile users now more than ever expect their devices to keep them up-to-date with the latest on their interests and what's going on around them. Your mobile app needs to be able to notify your customers, in near-real time, about updates in the important data that your app provides. In a cross-platform app, such as a Cordova-based app, dealing with four or more different native push notifications platforms can be tricky to set up and manage.

With its support for template-based registrations, Notification Hubs is a great solution for sending push notifications to Cordova apps. Push functionality is enabled in your app by the PushPlugin. Combined with the data storage and authentication facilities of Mobile Services, the Azure platform is convenient for supporting Cordova apps. For more information about using Azure to support cross-platform apps, see the Azure documentation topic, "Supporting Multiple Device Platforms from a Single Mobile Service," at aka.ms/w7oq0y. Not only is Visual Studio a great platform for developing Cordova apps, it's also a great tool for creating and managing Azure for your Cordova apps. ■

Glenn Gailey *is a programmer writer and 14-year Microsoft veteran who has created documentation and samples for Azure Mobile Services, Notification Hubs, WCF Data Services, OData, ADO.NET Entity Framework, SQL Server Replication and SQL Server CE. Reach him at glenn.gailey@microsoft.com or via Twitter at ggailey777.*

Thanks to the following Microsoft technical experts for reviewing this article: *Suresh Jayabalan, Mike Jones and Piyush Joshi*

Introducing the ASP.NET 5 Preview

Daniel Roth

ASP.NET shipped as part of the Microsoft .NET Framework 1.0, released in 2002 along with Visual Studio 2002. It was an evolution of Active Server Pages (ASP) that brought object-oriented design, the .NET Base Class Libraries (BCLs), better performance and much more. ASP.NET was designed to make it easy for developers used to writing desktop applications to build Web applications with ASP.NET Web Forms. As the Web evolved, new frameworks were added to ASP.NET: MVC in 2008, Web Pages in 2010, and Web API and SignalR in 2012. Each of these new frameworks built on top of the base from ASP.NET 1.0.

With ASP.NET 5, ASP.NET is being reimagined just like ASP was reimagined to ASP.NET in 2002. This reimagining brings many new features:

- **Full side-by-side support:** ASP.NET 5 applications can now be installed on a machine without affecting any other applications on the machine.
- **Cross-platform support:** ASP.NET 5 runs and is supported on Windows, Mac and Linux.
- **Cloud-ready:** Features such as diagnostics, session state, cache and configuration are designed to work locally and in the cloud.
- **Faster development:** The build step is removed; just save source files and refresh the browser and compilation happens automatically.
- **MVC, Web Pages and Web API:** These are all merged together, simplifying the number of concepts.
- **Flexible hosting:** You can now host your entire ASP.NET 5 application on IIS or in your own process.

This article discusses prerelease versions of ASP.NET 5 and Visual Studio 2015. All related information is subject to change.

This article discusses:

- Basics of the ASP.NET 5 runtime
- Working with Entity Framework
- Command-line functionality
- Visual Studio updates for ASP.NET 5

Technologies discussed:

ASP.NET 5, Visual Studio 2015 Preview

Getting Started with ASP.NET 5 Preview

In this article, I'll give an overview of the new experiences the ASP.NET development team—of which I'm a part—has created for ASP.NET 5 and Visual Studio 2015 Preview. For general help with building and running ASP.NET 5 applications, visit asp.net/vNext, where you can find step-by-step guides and additional documentation. In addition, we also post updates regularly to blogs.msdn.com/b/webdev. To get started, download and install Visual Studio 2015 Preview.

Overview of the ASP.NET 5 Runtime

ASP.NET 5 has been rebuilt from the ground up to support building modern Web applications and services. It's open source, cross-platform and works both on-premises and in the cloud. ASP.NET 5 is currently in Preview and under active development on GitHub (github.com/aspnet). I'll provide an overview of what's new in the ASP.NET 5 Preview along with pointers to where you can learn more.

Flexible, Cross-Platform Runtime At its foundation, ASP.NET 5 is based on a new flexible runtime host. It provides the flexibility to run your application on one of three different runtimes:

1. **Microsoft .NET Framework:** You can run your ASP.NET 5 applications on the existing .NET Framework. This gives you the greatest level of compatibility for existing binaries.
2. **.NET Core:** A refactored version of the .NET Framework that ships as a set of NuGet packages that you can include with your app. With .NET Core, you get support for true side-by-side versioning and the freedom to use the latest .NET features on your existing infrastructure. Note that not all APIs are available yet on .NET Core, and existing binaries generally need to be recompiled to run on .NET Core.
3. **Mono:** The Mono CLR enables you to develop and run ASP.NET 5 apps on a Mac or Linux device. For more information, see the blog post, "Develop ASP.NET vNext Applications on a Mac," at bit.ly/1AdChNZ.

Regardless of which CLR is used, ASP.NET 5 leverages a common infrastructure for hosting the CLR and provides various services to

Figure 1 **An Example project.json File**

```
{
  "webroot": "wwwroot",
  "version": "1.0.0-*",
  "exclude": [
    "wwwroot"
  ],
  "packExclude": [
    "**.kproj",
    "**.user",
    "**.vspscc"
  ],
  "dependencies": {
    "Microsoft.AspNet.Server.IIS": "1.0.0-beta1",
    "Microsoft.AspNet.Diagnostics": "1.0.0-beta1"
  },
  "frameworks" : {
    "aspnet50" : { },
    "aspnetcore50" : { }
  }
}
```

the application. This infrastructure is called the K Runtime Environment (KRE). While it's somewhat of a mystery where the "K" in KRE comes from (a tribute to the Katana Project? K for Krazy Kool?), the KRE provides everything you need to host and run your app.

A New HTTP Pipeline ASP.NET 5 introduces a new modular HTTP request pipeline that can be hosted on the server of your choice. You can host your ASP.NET 5 applications on IIS, on any Open Web Interface for .NET (OWIN)-based server or in your own process. Because you get to pick exactly what middleware runs in the pipeline for your app, you can run with as little or as much functionality as you need and take advantage of bare-metal performance. ASP.NET 5 includes middleware for security, request routing, diagnostics and custom middleware of your own design. For example, here's a simple middleware implementation for handling of the X-HTTP-Method-Override header:

```
app.Use((context, next) =>
{
  var value = context.Request.Headers["X-HTTP-Method-Override"];
  if (!string.IsNullOrEmpty(value))
  {
    context.Request.Method = value;
  }
  return next();
});
```

ASP.NET 5 uses an HTTP pipeline model similar in many ways to the OWIN-based model introduced with Project Katana, but with several notable improvements. Like Katana, ASP.NET 5 supports OWIN, but simplifies development by including a lightweight and easy-to-use HttpContext abstraction.

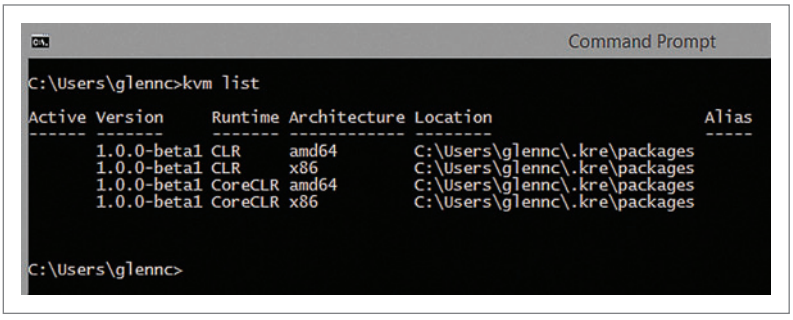


Figure 2 **Running "kvm list" at the Command Line to Get a List of KRE Versions on Your Machine**

There's a Package for That Package managers have changed the way developers think about installing, updating and managing dependencies. In ASP.NET 5, all your dependencies are represented as packages. NuGet packages are the unit of reference. ASP.NET 5 makes it easy to build, install and use packages from package feeds and also to work with community packages on the node package manager (NPM) and Bower. ASP.NET 5 introduces a simple JSON format (project.json) for managing NuGet package dependencies and for providing cross-platform build infrastructure. An example project.json file is shown in **Figure 1** (a more detailed explanation of each of the supported properties can be found on GitHub at bit.ly/1AlOhK3).

The Best of C# Design-time and run-time compilation for ASP.NET 5 applications are handled using the managed .NET Compiler Platform (code-named "Roslyn"). This means you get to take advantage of the latest C# language features while leveraging in-memory compilation to avoid unnecessary disk I/O. ASP.NET 5 projects are based on a new project system that dynamically compiles your application on-the-fly as you're coding so you can avoid the interruption of a specific build step. This gives you the power of .NET and C# with the agility and feel of an interpreted language.

Built-in Dependency Injection All ASP.NET 5 applications have access to a common dependency injection (DI) service that helps simplify composition and testing. All the ASP.NET frameworks built on ASP.NET 5 (MVC, Web API, SignalR and Identity) leverage this common DI service. While ASP.NET 5 comes with a minimalistic Inversion of Control (IoC) container to bootstrap the system, you can easily replace that built-in IoC container with your container of choice.

Familiar Web Frameworks ASP.NET 5 includes frameworks for building Web apps and services such as MVC, Web API, Web Pages (coming in a future release), SignalR and Identity. Each of these frameworks has been ported to work on the new HTTP request pipeline and has been built to support running on the .NET Framework, .NET Core or cross-platform.

Today, the existing implementations of MVC, Web API and Web Pages share many concepts and duplicate abstractions, but share very little in the way of actual implementation. As part of porting these frameworks to ASP.NET 5, Microsoft decided to take a fresh look at combining these frameworks into a single unified Web stack. ASP.NET MVC 6 takes the best of MVC, Web API and Web Pages and combines it into a single framework for building Web UI and Web APIs. This means from a single controller you can just as easily render a view as return formatted data based on content negotiation.

In addition to unification, ASP.NET MVC 6 introduces a host of new features:

- Built-in DI support
- Ability to create controllers from any class—no base class required
- Action-based request dispatching
- View Components—a simple replacement for child actions
- Routing improvements, including simplified attribute routing
- Async views with flush points

- Ability to inject servers and helpers into views using @inject
- ViewStart inheritance
- Tag helpers

You can find more information and samples at github.com/aspnet/mvc.

Web Forms isn't available on ASP.NET 5, but is still fully supported on the .NET Framework. There are a number of important new features coming to Web Forms in the upcoming version of the .NET Framework, including support for HTTP 2.0, async model binding and a Roslyn-based CodeDom provider. We're also working on various features reminiscent of Web Forms in MVC 6, such as tag helpers and other Razor improvements.

Entity Framework

Data is a key part of many applications and Entity Framework (EF) is a popular data access choice for ASP.NET developers. While EF7 isn't specific to ASP.NET, this new version of EF plays an integral role in ASP.NET 5.

EF7 Enables New Platforms EF is widely used in client and server applications that target the full .NET Framework. A key focus of EF7 is to enable EF to be used on the remaining platforms where .NET development is common. These include ASP.NET 5, Windows Store and Windows Phone applications, as well as .NET-based Mac and Linux applications.

For Windows Phone and Windows Store applications, the initial goal is to provide local data access using EF. SQLite is the most common database of choice on devices and will be the primary store for local data on devices with EF7. The full provider model will be available, though, so other data stores can be supported also.

EF7 Enables New Data Stores While parts of EF are clearly tied to relational data stores, much of the functionality that EF provides is also applicable to many non-relational data stores. Examples of such functionality include change tracking, LINQ and unit of work. EF7 will enable providers that target non-relational data stores, such as Microsoft Azure Table Storage.

We're explicitly not trying to build an abstraction layer that hides the type of data store you're targeting. The common patterns/components that apply to most data stores will be handled by the core framework. Things specific to particular types of data stores will be available as provider-specific extensions. For example, the concept of a model builder that allows you to configure your model will be part of the core framework. However, the ability to configure things such as cascade delete on a foreign key constraint will be included as extensions in the relational database provider.

EF7 Is Lightweight and Extensible EF7 will be a lightweight and extensible version that pulls forward some commonly used features. In addition, we'll be able to include some commonly requested features that would've been difficult to implement in the EF6 code base, but which can be included from the start in EF7.

The team will be keeping the same patterns and concepts you're used to in EF, except where there's a compelling reason to change something. You'll see the same DbContext/DbSet-based API, but it will be built over building block components that are easy to replace or extend as needed—the same pattern used for some of the isolated components added in recent EF releases.

More Information on EF7 For more information on EF7, visit the "What Is EF7 All About" GitHub page at aka.ms/AboutEF7. The page includes design information, links to blog posts and instructions for trying out EF7.

ASP.NET Command-Line Tools

One of the core ASP.NET 5 tenets was to provide a command-line experience before we built the tooling experience. This means that almost all tasks you need to do with an ASP.NET 5 application can be done from the command line. The main reason for this is to ensure a viable option for using ASP.NET 5 without Visual Studio for those using Mac or Linux machines.

KVM The first tool you need to get the full command-line experience for ASP.NET 5 is the K Version Manager (KVM). The KVM command-line tool can download new versions of the KRE and let you switch between them. KRE contains other command-line tools that you might use. How KVM is implemented, and how to get it, depends on the OS. You can download and install KVM for your platform by running the appropriate command from github.com/aspnet/Home.

Once you have KVM, you should be able to open a command prompt and run the kvm command. If you run "kvm list," you'll see the list of all KRE versions on your machine, as shown in **Figure 2**.

If there are no entries in your list, there are no versions of KRE in your user profile. To fix this, you can run the command "kvm upgrade." This command will determine the latest version of the KRE available, download it and modify your PATH environment variable so you can use the other command-line tools in the KRE itself.

You can use "kvm install <version/latest>" to install a particular version without making it the default. Use the –r switch to indicate whether you want the .NET Core or .NET Framework version of the KRE and the –x86 and –amd64 switches to download the 32- or 64-bit flavors of the KRE. The runtime and bitness switches can be supplied to either install or upgrade.

Once you've called "kvm upgrade," you'll be able to use the K and KPM commands. K can be used to run applications, while KPM is used to manage packages.

How does KVM work? At its heart, KVM is just a convenient way to manipulate your PATH. When you use "KVM use <version>," all it does is change your PATH to the bin folder of the KRE version you specified is on your PATH. By default the KRE is installed by copying and extracting the KRE .zip file into %USERPROFILE%\.kre\packages, so when you

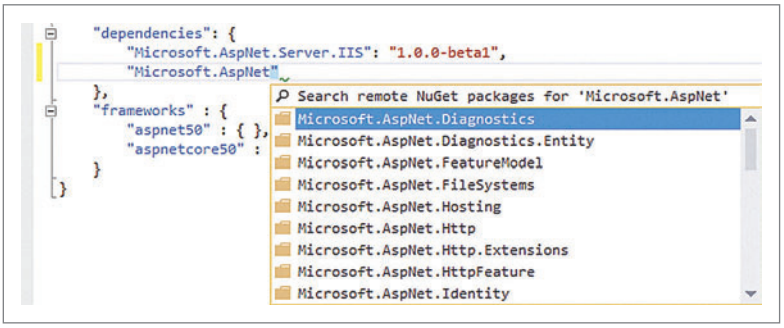


Figure 3 **IntelliSense for NuGet Package Dependencies in project.json**

type "KVM use 1.0.0-beta1," KVM will make sure that %USERPROFILE%\.kre\packages\KRE-CLR-x86.1.0.0-beta1\bin is on your PATH.

KPM The next tool you'll want to use is the KRE Package Manager (KPM). The KPM performs two main functions, with a few lesser features:

1. You can run "kpm restore" in a folder with a project.json file to download all the packages your application needs.
2. It provides the pack command, "kpm pack," which will take your application and generate a self-contained, runnable image of your application. Here, image means a folder structure that's designed to be copied to the server and run. It will include all the packages your application requires, as well as, optionally, the KRE on which you want to run the application.

The restore command can be run in a folder that contains a project.json file. It will examine the file and, using NuGet.config, connect to a NuGet feed and attempt to download all the packages your application needs. By default, it will install these packages in %USERPROFILE%\.kpm\packages so only one copy of any given package needs to be on your dev machine, even if used in multiple projects.

Packing your application—by running "kpm pack"—will generate a folder containing everything your app needs to run, including packages, source files and your Web root. You can even optionally include the KRE, although by default it's assumed the KRE is already on the server.

K Command The K command actually runs an ASP.NET 5 application from the command line. The K command is included in the KRE, the same as KPM, and is the entry point to running an application on top of the KRE.

The main way to use the K command is to run one of the commands inside your project.json file. Commands are specified by name in the project.json file under the commands property. By default, the ASP.NET 5 Starter Web template includes a "web" command in project.json that hosts your app and listens on port 5000. To run this command, simply run "k web."

Visual Studio Updates for ASP.NET 5

One of the original goals of ASP.NET 5 was to have a great experience for teams in which members use different tools. For example, you can have team members using Windows and Visual Studio working with others who are using Sublime Text on a Mac (see options for cross-platform .NET development tools at omnisharp.net). To achieve this, we had to take a step back and rethink Visual Studio support. In previous versions of Visual Studio, the project system assumed that most development was performed in Visual Studio. Visual Studio didn't work well when other tools were involved to create files or modify the project. For example, in the .csproj file, Visual Studio maintained a list of files that made up the project. If you used a tool to create a new file for your project, you'd then have to edit the .csproj file for it to be included.

In Visual Studio 2015, when you create a new ASP.NET 5 project, you get a new experience. You can still develop, debug and run your project as usual, but in addition to the standard features that you've come to know in ASP.NET projects, some new features are unique to ASP.NET 5. You now have the freedom to develop using the platform and tooling of your choice. I'll discuss some of those features.

Support for All Files in the Folder In ASP.NET 5, all files under the project directory are automatically included in the project. You can exclude files from compile or publish in the project.json file. For more info on how to exclude files in project.json, see the GitHub page at bit.ly/1AlOhK3. After the project is loaded, Visual Studio starts a file watcher and updates Solution Explorer to reflect the changes. Because Solution Explorer is always watching the files under the project directory, we've changed the location where generated files will be stored. Instead of storing generated files under the project (bin\ and obj\), we now place generated files by default in a folder named artifacts next to the solution file.

Just Edit, Save and Refresh the Browser In existing ASP.NET applications, when you change server-side logic (such as your MVC controller code, or a filter) then you need to rebuild and redeploy the application to see the changes reflected in the browser. Microsoft wanted the Web developer workflow to feel as lightweight and agile as when working with interpreted platforms (such as Node.js or Ruby), while still letting you leverage the power of .NET. In ASP.NET 5 projects, when you edit and save your C# code, a file watcher detects the change and restarts the application. The application is rebuilt in memory, so you can see the results of the change in the browser in near-real time. Note that this workflow is only supported when you aren't debugging so as to avoid interrupting your debugging session.

Updated Support for NuGet Packages In ASP.NET 5 all your dependencies are NuGet packages. Your package dependencies are listed in project.json and only direct references are listed. The runtime resolves package dependencies for you and you can see and search through the entire graph of your package dependencies in the Solution Explorer.

To install a NuGet package, you simply add the package to your project.json file. As soon as you save the project.json file, a package restore command is initiated and any changes to your dependencies are reflected in the References node in the Solution Explorer. You can see the result of the package restore and any errors that might have occurred in the package manager log in the Output window. You even get IntelliSense for packages installed locally on the machine with Visual Studio and packages that are available on the public NuGet feed, as shown in **Figure 3**.

In existing ASP.NET projects, when you install a NuGet package into a project, a copy of that NuGet package (and all of its dependencies) gets placed in a packages folder near the solution. In ASP.NET 5, a cache is used so NuGet packages are shared across projects for each user. When a package restore occurs in ASP.NET 5, any packages already installed in the cache are simply reused. You can see and modify the package cache in your user profile under the .kpm folder.

By default, ASP.NET 5 projects are built in-memory by the runtime and no artifacts are persisted to disk. However, you can easily enable building NuGet packages for your ASP.NET 5 class libraries by selecting the "Produce all outputs on build" checkbox on the Build tab of the Properties page for the project. Once this option is selected, you can find the built NuGet package under the artifacts folder for the solution.

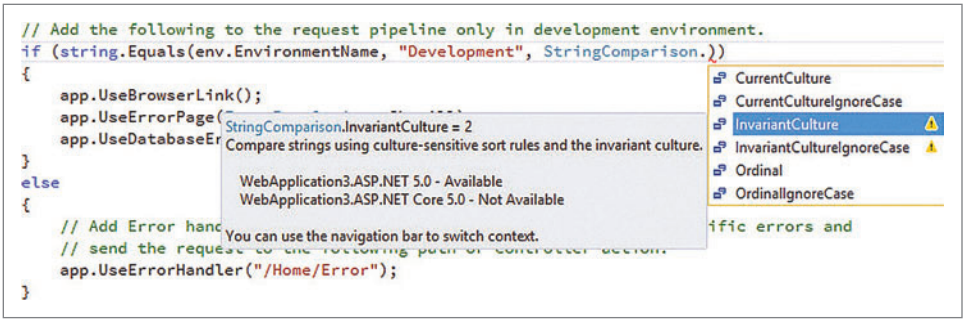


Figure 4 **Tooltip Showing Combined IntelliSense for Multiple Target Frameworks**

Build and Get Combined IntelliSense for Multiple Target Frameworks In ASP.NET 5, you list your target frameworks in project.json. Your project is built against every target framework listed. This makes discovering issues much quicker because you don't have to explicitly switch to that target framework and you can cross-compile for multiple target frameworks from a single project.

We've also updated the IntelliSense experience for ASP.NET 5 project with Combined IntelliSense. With Combined Intellisense, when the IntelliSense for a given completion differs for one or more of the target frameworks, you'll see an expanded tooltip to show you the differences. You can see this in **Figure 4** for the StringComparison class.

In **Figure 4**, you can see that the InvariantCulture value isn't available for the StringComparison enum when using ASP.NET Core 5.0.

Web Publishing In Visual Studio 2015, Microsoft developers are working on a new publish process for ASP.NET 5 projects. In the Preview release, ASP.NET 5 publishing supports publishing to Azure Websites and to the file system (for example, local/network folder). When publishing to Azure Websites, you can select the desired build configuration and KRE version. Later releases will expand this to support a broader set of targets.

Migrating to ASP.NET 5

Moving existing Web applications to ASP.NET 5 involves both creating a new ASP.NET 5 project for your existing application and then migrating your code and dependencies to run on the new framework. Creating a new ASP.NET 5 project for your application is relatively simple. First, add a project.json file in your project folder. Initially, the project.json file only needs to include an empty JSON object (for example, {}). Next, use File | Open Project to open the project.json file in Visual Studio 2015 Preview. After opening the project.json file, Visual Studio will create an ASP.NET 5 project with a .kproj file and automatically include in the project all the files and directories it finds next to the project.json file. You should see your project files in your new ASP.NET 5 project in the Solution Explorer. You have now created an ASP.NET 5 project for your existing Web application!

Migrating your code and dependencies so your new ASP.NET 5 project builds and runs correctly is a more involved process. You need to update your project.json with your top-level package dependencies, framework assembly references and project references. You need to migrate your code to use the new HTTP abstractions, the new middleware model and the new versions of the ASP.NET Web frameworks. You need to move to new infrastructure for handling concerns such as configuration, logging and DI. Porting your application to run on .NET Core requires dealing with additional platform changes and limitations. This is more than can be covered in this article, but we're working hard to provide complete migration guidance in a future article. While the investment to move to ASP.NET 5 might be significant, Microsoft believes the benefits of providing an open source, community-driven, cross-platform and cloud-ready framework are worth the additional effort.

Where to Go from Here?

In this article, I've explored ASP.NET 5 from high-level concepts to low-level components. There's a lot to learn! Fortunately, there are plenty of resources to help.

The ASP.NET site has a dedicated content area at asp.net/vnext. This page is constantly updated to feature release news, documentation, tutorials and community resources.

The ASP.NET vNext Community Standup is an open weekly meeting with the ASP.NET team. It's held every Tuesday, alternating between morning and afternoon Pacific Time, so developers around the world can attend. Come join us to hear what's coming next, ask questions and give your feedback.

You can get help from fellow developers and ASP.NET team members in the dedicated ASP.NET vNext forum at bit.ly/1xOuQx9. There's also a chat room in JabbR.net focused on ASP.NET 5/vNext development at bit.ly/1HDAGFX.

Because ASP.NET 5 is being developed as an open source project, you can get involved in the development process. The source repositories under github.com/aspnet are the actual, live repositories the team develops against, not a mirror of an internal repository. This means you can see the commits as they happen and grab a snapshot of the code at any time. The repositories include source code for the frameworks (for example, KRuntime, Razor and MVC), as well as functional tests and sample projects. We encourage you to leave feedback—both bugs and feature suggestions—by opening or commenting on issues in the GitHub repositories.

It's an exciting time for Web development on the Microsoft Web platform! We encourage you to get involved early, both so you'll be ready to take advantage of the new features and so you can provide feedback during the development process.

Finally, I want to thank Glenn Condron, Jon Galloway, Sayed Ibrahim Hashimi, Scott Hunter and Rowan Miller for going beyond the call of regular technical review duties and actually contributing significant parts of this article. ■

Daniel Roth *is a senior program manager on the ASP.NET team currently working on ASP.NET 5. His passions include .NET development and delighting customers by making frameworks simple and easy-to-use.*

Thanks to the following Microsoft technical experts for reviewing this article: *Glenn Condron, David Fowler, Jon Galloway, Sayed Ibrahim Hashimi, Scott Hunter, Rowan Miller and Praburaj Thiagaraj*

How C# 6.0 Simplifies, Clarifies and Condenses Your Code

Mark Michaelis

C# 6.0 isn't a radical revolution in C# programming. Unlike the introduction of generics in C# 2.0, C# 3.0 and its groundbreaking way to program collections with LINQ, or the simplification of asynchronous programming patterns in C# 5.0, C# 6.0 isn't going to transform development. That said, C# 6.0 will change the way you write C# code in specific scenarios, due to features that are so much more efficient you'll likely forget there was another way to code them. It introduces new syntax shortcuts, reduces the amount of ceremony on occasion, and ultimately makes writing C# code leaner. In this article I'm going to delve into the details of the new C# 6.0 feature set that make all this possible. Specifically, I'll focus on the items outlined in the Mind Map shown in **Figure 1**.

Note that many of the examples herein come from the next edition of my book, "Essential C# 6.0" (Addison-Wesley Professional).

This article discusses Visual Studio 2015 Preview; all information is subject to change.

This article discusses:

- Using static directives
- The nameof operator
- String interpolation
- Null-conditional operator
- Parameterless constructors in structs
- Auto-Property improvements
- Expression-bodied methods
- Dictionary initializer

Technologies discussed:

C# 6.0, Visual Studio 2015 Preview

Using Static

Many of the C# 6.0 features can be leveraged in the most basic of Console programs. For example, using is now supported on specific classes in a feature called the using static directive, which renders static methods available in global scope, without a type prefix of any kind, as shown in **Figure 2**. Because System.Console is a static class, it provides a great example of how this feature can be leveraged.

In this example, there are using static directives for System.ConsoleColor, System.IO.Directory, System.Threading.Interlocked and System.Threading.Tasks.Parallel. These enable the invocation of numerous methods, properties and enums directly: ForegroundColor, WriteLine, GetFiles, Increment, Yellow, White and ForEach. In each case, this eliminates the need to qualify the static member with its type. (For those of you using Visual Studio 2015 Preview or earlier, the syntax doesn't include adding the "static" keyword after using, so it's only "using System.Console," for example. In addition, not until after Visual Studio 2015 Preview does the using static directive work for enums and structs in addition to static classes.)

For the most part, eliminating the type qualifier doesn't significantly reduce the clarity of the code, even though there is less code. WriteLine in

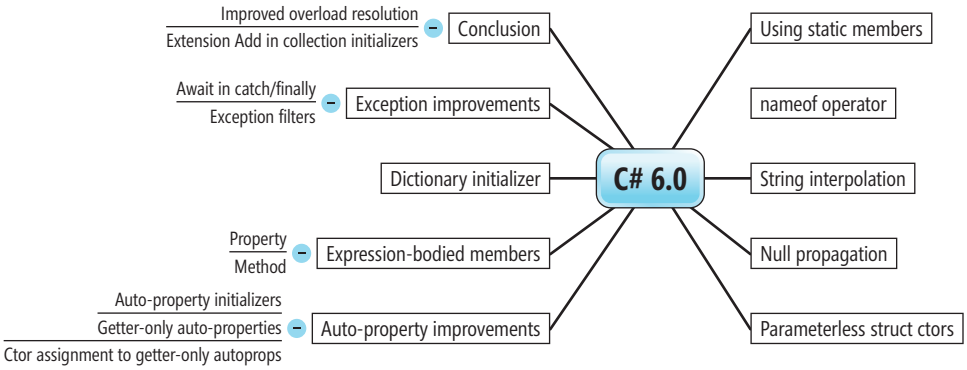


Figure 1 **C# 6.0 Mind Map**

a console program is fairly obvious as is the call to GetFiles. And, because the addition of the using static directive on System.Threading.Tasks.Parallel was obviously intentional, ForEach is a way of defining a parallel foreach loop that, with each version of C#, looks (if you squint just right) more and more like a C# foreach statement.

The obvious caution with the using static directive is to take care that clarity isn't sacrificed. For example, consider the Encrypt function defined in **Figure 3**.

It would seem that the call to Exists is exactly what's needed. However, more explicitly, the call is *Directory.Exists* when, in fact, *File.Exists* is what's needed here. In other words, although the code is certainly readable, it's incorrect and, at least in this case, avoiding the *using static* syntax is probably better.

Note that if using static directives are specified for both System.IO.Directory and System.IO.File, the compiler issues an error when calling Exists, forcing the code to be modified with a type disambiguation prefix in order to resolve the ambiguity.

Figure 2 **The Using Static Directive Reduces Noise Within Your Code**

```
using System;
using static System.ConsoleColor;
using static System.IO.Directory;
using static System.Threading.Interlocked;
using static System.Threading.Tasks.Parallel;

public static class Program
{
  // ...
  public static void Main(string[] args)
  {
    // Parameter checking eliminated for elucidation.
    EncryptFiles(args[0], args[1]);
  }

  public static int EncryptFiles(
    string directoryPath, string searchPattern = "*")
  {

    ConsoleColor color = ForegroundColor;
    int fileCount = 0;

    try
    {
      ForegroundColor = Yellow
      WriteLine("Start encryption...");

      string[] files = GetFiles(directoryPath, searchPattern,
        System.IO.SearchOption.AllDirectories);

      ForegroundColor = White

      ForEach(files, (filename) =>
      {
        Encrypt(filename);
        WriteLine("\t'{0}' encrypted", filename);
        Increment(ref fileCount);
      });

      ForegroundColor = Yellow
      WriteLine("Encryption completed");
    }
    finally
    {
      ForegroundColor = color;
    }
    return fileCount;
  }
}
```

> C# will change the way you code due to features that are so much more efficient you'll likely forget there was another way.

An additional feature of the using static directive is its behavior with extension methods. Extension methods aren't moved into global scope, as would happen with static methods normally. For example, a using static ParallelEnumerable directive would not put the Select method into global scope and you couldn't call Select(files, (filename) => { ... }). This restriction is by design. First, extension methods are intended to be rendered as instance methods on an object (files.Select((filename)=>{ ... }) for example) and it's not a normal pattern to call them as static methods directly off of the type. Second, there are libraries such as System.Linq, with types such as Enumerable and ParallelEnumerable that have overlapping method names like Select. To have all these types added to the global scope forces unnecessary clutter into IntelliSense and possibly introduces an ambiguous invocation (although not in the case of System.Linq-based classes).

Although extension methods won't get placed into global scope, C# 6.0 still allows classes with extension methods in using static directives. The using static directive achieves the same as a using (namespace) directive does except only for the specific class targeted by the using static directive. In other words, using static allows the developer to narrow what extensions methods are available down to the particular class identified, rather than the entire namespace. For example, consider the snippet in **Figure 4**.

Notice there's no using System.Linq statement. Instead, there is the using static System.Linq.ParallelEnumerable directive, causing only extension methods from ParallelEnumerable to be in scope as extension methods. All the extension methods on classes like System.Linq.Enumerable will not be available as extension methods. For example, files.Select(...) will fail compilation because Select isn't in scope on a string array (or even IEnumerable<string>). In contrast, AsParallel is in scope via System.Linq.ParallelEnumerable. In summary, the using static directive on a class with extension methods will bring that class' extension methods into scope as extension methods. (Non-extension methods on the same class will be brought into global scope normally.)

In general, the best practice is to limit usage of the using static directive to a few classes that are used repeatedly throughout the scope (unlike Parallel) such as System.Console or System.Math. Similarly, when using static for enums, be sure the enum items are self-explanatory without their type identifier. For example, and perhaps my favorite, specify using Microsoft.VisualStudio.TestTools.UnitTesting.Assert in unit tests files to enable test assertion invocations such as IsTrue, AreEqual<T>, Fail and IsNotNull.

The nameof Operator

Figure 3 includes another feature new in C# 6.0, the nameof operator. This is a new contextual keyword to identify a string literal that extracts a constant for (at compile time) the unqualified name of whatever identifier is specified as an argument. In **Figure 3**, nameof(filename) returns "filename," the name of the Encrypt method's parameter. However, nameof works with any programmatic identifier. For example, **Figure 5** leverages nameof to pass the property name to INotifyPropertyChanged.PropertyChanged. (By the way, using the CallerMemberName attribute to retrieve a property name for the PropertyChanged invocation is still a valid approach for retrieving the property name; see itl.tc/?p=11661.)

Figure 3 **Ambiguous Exists Invocation (with the nameof Operator)**

```
private static void Encrypt(string filename)
  {
    if (!Exists(filename)) // LOGIC ERROR: Using Directory rather than File
    {
      throw new ArgumentException("The file does not exist.", nameof(filename));
    }

    // ...
  }
```

Figure 4 **Only Extension Methods from ParallelEnumerable Are in Scope**

```
using static System.Linq.ParallelEnumerable;
using static System.IO.Console;
using static System.Threading.Interlocked;

// ...
    string[] files = GetFiles(directoryPath, searchPattern,
      System.IO.SearchOption.AllDirectories);
    files.AsParallel().ForAll( (filename) =>
    {
      Encrypt(filename);
      WriteLine($"\t'{filename}' encrypted");
      Increment(ref fileCount);
    });
// ...
```

Notice that whether only the unqualified "Name" is provided (because it's in scope) or the fully qualified CSharp6.Person.Name identifier is used as in the test, the result is only the final identifier (the last element in a dotted name).

By leveraging the nameof operator, it's possible to eliminate the vast majority of "magic" strings that refer to code identifiers as long as they're in scope. This not only eliminates runtime errors due to misspellings within the magic strings, which are never verified by the complier, but also enables refactoring tools like Rename to update all references to the name change identifier. And, if a name changes without a refactoring tool, the compiler will issue an error indicating that the identifier no longer exists.

String Interpolation

Figure 3 could be improved by not only specifying the exception message indicating the file wasn't found, but also by displaying the file name itself. Prior to C# 6.0, you'd do this using string.Format in order to embed the filename into the string literal. However, com-

Figure 5 **Using the nameof Operator for INotifyPropertyChanged.PropertyChanged**

```
public class Person : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  public Person(string name)
  {
    Name = name;
  }

  private string _Name;
  public string Name
  {
    get{ return _Name; }
    set
    {
      if (_Name != value)
      {
        _Name = value;
        PropertyChangedEventHandler propertyChanged = PropertyChanged;
        if (propertyChanged != null)
        {
          propertyChanged(this,
            new PropertyChangedEventArgs(nameof(Name)));
        }
      }
    }
  }


  // ...
}

[TestClass]
public class PersonTests
{

  [TestMethod]
  public void PropertyName()
  {
    bool called = false;
    Person person = new Person("Inigo Montoya");
    person.PropertyChanged += (sender, eventArgs) =>
    {
      AreEqual(nameof(CSharp6.Person.Name), eventArgs.PropertyName);
      called = true;
    };
    person.Name = "Princess Buttercup";
    IsTrue(called);
  }
}
```

Figure 6 **Composite String Formatting Versus String Interpolation**

```
[TestMethod]
public void InterpolateString()
{
  Person person = new Person("Inigo", "Montoya") { Age = 42 };
  string message =
    string.Format("Hello!  My name is {0} {1} and I am {2} years old.",
    person.FirstName, person.LastName, person.Age);
  AreEqual<string>
    ("Hello!  My name is Inigo Montoya and I am 42 years old.", message);

  string messageInterpolated =
    $"Hello!  My name is {person.FirstName} {person.LastName} and I am
    {person.Age} years old.";
  AreEqual<string>(message, messageInterpolated);
}
```

posite formatting wasn't the most readable. Formatting a person's name, for example, required substituting placeholders based on the order of parameters as shown in the **Figure 6** assignment of message.

Notice the alternate approach to composite formatting with the assignment to messageInterpolated. In this example, the expression assigned to messageInterpolated is a string literal prefixed with a "$" and curly brackets identify code that is embedded inline within the string. In this case, the properties of person are used to make this string significantly easier to read than a composite string. Furthermore, the string interpolation syntax reduces errors caused by arguments following the format string that are in improper order, or missing altogether and causing an exception. (In Visual Studio 2015 Preview there's no $ character and, instead, each left curly bracket requires a slash before it. Releases following Visual Studio 2015 Preview are updated to use the $ in front of the string literal syntax instead.)

String interpolation is transformed at compile time to invoke an equivalent string.Format call. This leaves in place support for localization as before (though still with traditional format strings) and doesn't introduce any post compile injection of code via strings.

Figure 7 shows two more examples of string interpolation.

Notice that in the second case, the throw statement, both string interpolation and the nameof operator are leveraged. String interpolation is what causes the ArgumentException message to include the file name (that is, "The file, 'c:\data\missingfile.txt' does not exist"). The use of the nameof operator is to identify the name of the Encrypt parameter ("filename"), the second argument of the ArgumentException constructor. Visual Studio 2015 is fully aware of the string interpolation syntax, providing both color coding and IntelliSense for the code blocks embedded within the interpolated string.

Figure 7 **Using String Interpolation in Place of string.Format**

```
public Person(string firstName, string lastName, int? age=null)
{
  Name = $"{firstName} {lastName}";
  Age = age;
}
private static void Encrypt(string filename)
{
  if (!System.IO.File.Exists(filename))
  {
    throw new ArgumentException(
      $"The file, '{filename}', does not exist.", nameof(filename));
  }
  // ...
}
```

Null-Conditional Operator

Although eliminated in **Figure 2** for clarity, virtually every Main method that accepts arguments requires checking the parameter for null prior to invoking the Length member to determine how many parameters were passed in. More generally, it's a very common pattern to check for null before invoking a member in order to avoid a System.NullReferenceException (which almost always indicates an error in the programming logic). Because of the frequency of this pattern, C# 6.0 introduces the "?." operator known as the null-conditional operator:

```
public static void Main(string[] args)
{
  switch (args?.Length)
  {
  // ...
  }
}
```

The null-conditional operator translates to checking whether the operand is null prior to invoking the method or property (Length in this case). The logically equivalent explicit code would be (although in the C# 6.0 syntax the value of args is only evaluated once):

```
(args != null) ? (int?)args.Length : null
```

What makes the null-conditional operator especially convenient is that it can be chained. If, for example, you invoke string[] names = person?.Name?.Split(' '), Split will only be invoked if both person and person.Name are not null. When chained, if the first operand is null, the expression evaluation is short-circuited, and no further invocation within the expression call chain will occur. Beware, however, you don't unintentionally neglect additional null-conditional operators. Consider, for example, names = person?.Name.Split(' '). If there's a person instance but Name is null, a NullReferenceException will occur upon invocation of Split. This doesn't mean you must use a chain of null-conditional operators, but rather that you should be intentional about the logic. In the Person case, for example, if Name is validated and can never be null, no additional null-conditional operator is necessary.

An important thing to note about the null-conditional operator is that, when utilized with a member that returns a value type, it always returns a nullable version of that type. For example, args?.Length returns an int?, not simply an int. Although perhaps a little peculiar (in comparison to other operator behavior), the return of a nullable value type occurs only at the end of the call chain. The result is that calling the dot (".") operator on Length only allows

Figure 8 **A Console Color Configuration Example**

```
string jsonText =
    @"{
      'ForegroundColor':  {
        'Error':  'Red',
        'Warning':  'Red',
        'Normal':  'Yellow',
        'Verbose':  'White'
      }
    }";

  JObject consoleColorConfiguration = JObject.Parse(jsonText);
  string colorText = consoleColorConfiguration[
    "ForegroundColor"]?["Normal"]?.Value<string>();
  ConsoleColor color;
  if (Enum.TryParse<ConsoleColor>(colorText, out color))
  {
    Console.ForegroundColor = colorText;
  }
```

invocation of int (not int?) members. However, encapsulating args?.Length in parenthesis—forcing the int? result via parentheses operator precedence—will invoke the int? return and make the Nullable<T> specific members (HasValue and Value) available.

The null-conditional operator is a great feature on its own. However, using it in combination with a delegate invocation resolves a C# pain point that has existed since C# 1.0. Note how in **Figure 5** I assign the PropertyChange event handler to a local copy (propertyChanged) before checking the value for null and then finally firing the event. This is the easiest thread safe way to invoke events without risking that an event unsubscribe will occur between the time the check for null occurs and when the event is fired. Unfortunately, this is non-intuitive and I frequently encounter code that doesn't follow this pattern—with the result of inconsistent NullReferenceExceptions. Fortunately, with the introduction of the null-conditional operator in C# 6.0, this issue is resolved.

Figure 9 **Declaring a Default Constructor on a Value Type**

```
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.IO;

public struct ConsoleConfiguration
{
  public ConsoleConfiguration() :
    this(ConsoleColor.Red, ConsoleColor.Yellow, ConsoleColor.White)
  {
    Initialize(this);
  }

  public ConsoleConfiguration(ConsoleColor foregroundColorError,
    ConsoleColor foregroundColorInformation,
    ConsoleColor foregroundColorVerbose)
  {
    // All auto-properties and fields must be set before
    // accessing expression bodied members
    ForegroundColorError = foregroundColorError;
    ForegroundColorInformation = foregroundColorInformation;
    ForegroundColorVerbose = foregroundColorVerbose;
  }

   private static void Initialize(ConsoleConfiguration configuration)
  {
    // Load configuration from App.json.config file ...
  }

  public ConsoleColor ForegroundColorVerbose { get; }
  public ConsoleColor ForegroundColorInformation { get; }
  public ConsoleColor ForegroundColorError { get; }

  // ...

  // Equality implementation excluded for elucidation
}

[TestClass]
public class ConsoleConfigurationTests
{
  [TestMethod]
  public void DefaultObjectIsNotInitialized()
  {
    ConsoleConfiguration configuration = default(ConsoleConfiguration);
    AreEqual<ConsoleColor>(0, configuration.ForegroundColorError);
    ConsoleConfiguration[] configurations = new ConsoleConfiguration[42];
    foreach(ConsoleConfiguration item in configurations)
    {
      AreEqual<ConsoleColor>(default(ConsoleColor),
        configuration.ForegroundColorError);
      AreEqual<ConsoleColor>(default(ConsoleColor),
        configuration.ForegroundColorInformation);
      AreEqual<ConsoleColor>(default(ConsoleColor),
        configuration.ForegroundColorVerbose);
    }
  }

  [TestMethod]
  public void CreateUsingNewIsInitialized()
  {
    ConsoleConfiguration configuration = new ConsoleConfiguration();
    AreEqual<ConsoleColor>(ConsoleColor.Red,
      configuration.ForegroundColorError);
    AreEqual<ConsoleColor>(ConsoleColor.Yellow,
      configuration.ForegroundColorInformation);
    AreEqual<ConsoleColor>(ConsoleColor.White,
      configuration.ForegroundColorVerbose);
  }
}
```

With C# 6.0, the code snippet changes from:

```
PropertyChangedEventHandler propertyChanged = PropertyChanged;
if (propertyChanged != null)
{
  propertyChanged(this, new PropertyChangedEventArgs(nameof(Name)));
}
```

to simply:

```
PropertyChanged?.Invoke(propertyChanged(
  this, new PropertyChangedEventArgs(nameof(Name)));
```

And, because an event is just a delegate, the same pattern of invoking a delegate via the null-conditional operator and an Invoke is always possible. This feature, perhaps more than any other in C# 6.0, is sure to change the way you write C# code in the future. Once you leverage null-conditional operators on delegates, chances are you'll never go back and code the old way (unless, of course, you're stuck in a pre-C# 6.0 world.)

> Once you leverage null-conditional operators on delegates, chances are you'll never go back and code the old way.

Null-conditional operators can also be used in combination with an index operator. For example, when you use them in combination with a Newtonsoft.JObject, you can traverse a JSON object to retrieve specific elements, as shown in **Figure 8**.

It's important to note that, unlike most collections within MSCORLIB, JObject doesn't throw an exception if an index is invalid. If, for example, ForegroundColordoesn't exist, JObject returns null rather than throwing an exception. This is significant because using the null-conditional operator on collections that throw an IndexOutOfRangeException is almost always unnecessary and may imply safety when no such safety exists. Returning to the snippet showing the Main and args example, consider the following:

```
public static void Main(string[] args)
{
  string directoryPath = args?[0];
  string searchPattern = args?[1];

  // ...
}
```

Figure 10 **Expression Bodied Auto-Properties**

```
public class Person
{
  public Person(string name)
  {
    Name = name;
  }

  public Person(string firstName, string lastName)
  {
    Name = $"{firstName} {lastName}";
    Age = age;
  }

  // Validation ommitted for elucidation
  public string Name {get; set; }
  public string FirstName => Name.Split(' ')[0];
  public string LastName => Name.Split(' ')[1];

  public override string ToString() => "\{Name}(\{Age}";

}
```

What makes this example dangerous is that the null-conditional operator gives a false sense of security, implying that if args isn't null then the element must exist. Of course, this isn't the case because the element may not exist even if args isn't null. Because checking for the element count with args?.Length already verifies that args isn't null, you never really need to also use the null-conditional operator when indexing the collection after checking length. In conclusion, avoid using the null-conditional operator in combination with the index operator if the index operator throws an IndexOutOfRangeException for non-existent indexes. Doing so leads to a false sense of code validity.

Default Constructors in Structs

Another C# 6.0 feature to be cognizant of is support for a default (parameterless) constructor on a value type. This was previously disallowed because the constructor wouldn't be called when initializing arrays, defaulting a field of type struct, or initializing an instance with the default operator. In C# 6.0, however, default constructors are now allowed with the caveat that *they're only invoked when the value type is instantiated with the new operator*. Both array initialization and explicit assignment of the default value (or the implicit initialization of a struct field type) will circumvent the default constructor.

Figure 11 **Using Await Within Catch and Finally Blocks**

```
public static async Task<int> EncryptFilesAsync(string directoryPath,
string searchPattern = "*")
{
  ConsoleColor color = Console.ForegroundColor;

  try
  {

  // ...

  }
  catch (System.ComponentModel.Win32Exception exception)
    if (exception.NativeErrorCode == 0x00042)
  {
    // ...
  }
  catch (AggregateException exception)
  {
    await LogExceptionsAsync(exception.InnerExceptions);
  }
  finally
  {
    Console.ForegroundColor = color;
    await RemoveTemporaryFilesAsync();
  }
}
```

To understand how the default constructor is leveraged, consider the example of the ConsoleConfiguration class shown in **Figure 9**. Given a constructor, and its invocation via the new operator as shown in the CreateUsingNewIsInitialized method, structs will be fully initialized. As you'd expect, and as is demonstrated in **Figure 9**, constructor chaining is fully supported, whereby one constructor can call another using the "this" keyword following the constructor declaration.

There's one key point to remember about structs: all instance fields and auto-properties (because they have backing fields) must be fully initialized prior to invoking any other instance members. As a result, in the example in **Figure 9**, the constructor can't call the Initialize method until all fields and auto-properties have been assigned. Fortunately, if a chained constructor handles all requisite initialization and is invoked via a "this" invocation, the compiler is clever enough to detect that it isn't necessary to initialize data again from the body of the non-this invoked constructor, as shown in **Figure 9**.

Auto-Property Improvements

Notice also in **Figure 9** that the three properties (for which there are no explicit fields) are all declared as auto-properties (with no body) and only a getter. These getter-only auto-properties are a C# 6.0 feature for declaring read-only properties that are backed (internally) by a read-only field. As such, these properties can only be modified from within the constructor.

Getter-only auto-properties are available in both structs and class declarations, but they're especially important to structs because of the best practice guideline that structs be immutable. Rather than the six or so lines needed to declare a read-only property and initialize it prior to C# 6.0, now a single-line declaration and the assignment from within the constructor are all that's needed. Thus, declaration of immutable structs is now not only the correct programming pattern for structs, but also the simpler pattern—a much appreciated change from prior syntax where coding correctly required more effort.

A second auto-property feature introduced in C# 6.0 is support for initializers. For example, I can add a static DefaultConfig auto-property with initializer to ConsoleConfiguration:

```
// Instance property initialization not allowed on structs.
static private Lazy<ConsoleConfiguration> DefaultConfig{ get; } =
  new Lazy<ConsoleConfiguration>(() => new ConsoleConfiguration());
```

Such a property would provide a single instance factory pattern for accessing a default ConsoleConfigurtion instance. Notice that rather than assigning the getter only auto-property from within the constructor, this example leverages System.Lazy<T> and instantiates it as an initializer during declaration. The result is that once the constructor completes, the instance of Lazy<ConsoleConfiguration> will be immutable and an invocation of DefaultConfig will always return the same instance of ConsoleConfiguration.

Note that auto-property initializers aren't allowed on instance members of structs (although they're certainly allowed on classes).

Expression Bodied Methods and Auto-Properties

Another feature introduced in C# 6.0 is expression bodied members. This feature exists for both properties and methods and allows the use of the arrow operator (=>) to assign an expression to either a property or method in place of a statement body. For example, because the DefaultConfig property in the previous example is both private and of type Lazy<T>, retrieving the actual default instance of ConsoleConfiguration requires a GetDefault method:

```
static public ConsoleConfiguration GetDefault() => DefaultConfig.Value;
```

However, in this snippet, notice there's no statement block type method body. Rather, the method is implemented with only an expression (not a statement) prefixed with the lambda arrow operator. The intent is to provide a simple one-line implementation without all the ceremony, and one that's functional with or without parameters in the method signature:

```
private static void LogExceptions(ReadOnlyCollection<Exception> innerExceptions) =>
  LogExceptionsAsync(innerExceptions).Wait();
```

In regard to properties, note that the expression bodies work only for read-only (getter-only) properties. In fact, the syntax is virtually identical to that of the expression bodied methods except that there are no parentheses following the identifier. Returning to the earlier Person example, I could implement read-only FirstName and LastName properties using expression bodies, as shown in **Figure 10**.

Furthermore, expression-bodied properties can also be used on index members, returning an item from an internal collection, for example.

Dictionary Initializer

Dictionary type collections are a great means for defining a name value pair. Unfortunately, the syntax for initialization is somewhat suboptimal:

```
{ {"First", "Value1"}, {"Second", "Value2"}, {"Third", "Value3"} }
```

To improve this, C# 6.0 includes a new dictionary assignment type syntax:

```
Dictionary<string, Action<ConsoleColor>> colorMap =
  new Dictionary<string, Action<ConsoleColor>>
{
  ["Error"] =       ConsoleColor.Red,
  ["Information"] = ConsoleColor.Yellow,
  ["Verbose"] =     ConsoleColor.White
};
```

To improve the syntax, the language team introduced the assignment operator as a means of associating a pair of items that make a lookup (name) value pair or map. The lookup is whatever the index value (and data type) the dictionary is declared to be.

Exception Improvements

Not to be outdone, exceptions also had a couple of minor language tweaks in C# 6.0. First, it's now possible to use await clauses within both catch and finally blocks, as shown in **Figure 11**.

Since the introduction of await in C# 5.0, support for await in catch and finally blocks turned out to be far more sought after than originally expected. For example, the pattern of invoking asynchronous methods from within a catch or finally block is fairly common, especially when it comes to cleaning up or logging during those times. Now, in C# 6.0, it's finally possible.

The second exception-related feature (which has been available in Visual Basic since 1.0) is support for exception filters such that on top of filtering by a particular exception type, it's now possible to specify an if clause to further restrict whether the exception will be caught by the catch block or not. (On occasion this feature has also been leveraged for side effects such as logging exceptions as an exception "flies by" without actually performing any exception processing.) One caution to note about this feature is that, if there's any chance your application might be localized, avoid catch conditional expressions that operate via exception messages because they'll no longer work without changes following localization.

Wrapping Up

One final point to mention about all the C# 6.0 features is that although they obviously require the C# 6.0 compiler, included with Visual Studio 2015 or later, they do not require an updated version of the Microsoft .NET Framework. Therefore, you can use C# 6.0 features even if you're compiling against the .NET Framework 4, for example. The reason this is possible is that all features are implemented in the compiler, and don't have any .NET Framework dependencies.

With that whirlwind tour, I wrap up my look at C# 6.0. The only two remaining features not discussed are support for defining a custom Add extension method to help with collection initializers, and some minor but improved overload resolution. In summary, C# 6.0 doesn't radically change your code, at least not in the way that generics or LINQ did. What it does do, however, is make the right coding patterns simpler. The null-conditional operator of a delegate is probably the best example of this, but so, too, are many of the other features including string interpolation, the nameof operator and auto-property improvements (especially for read-only properties).

For additional information here are a few additional references:

- What's new in C# 6.0 by Mads Torgersen (video): bit.ly/CSharp6Mads
- Mark Michaelis' C# Blog with 6.0 updates since writing this article: itl.tc/csharp
- C# 6.0 language discussions: roslyn.codeplex.com/discussions

In addition, although not available until the second quarter of 2015, look for the next release of my book, "Essential C# 6.0" (intellitect.com/EssentialCSharp).

By the time you read this, C# 6.0 feature discussions will likely be closed. However, there's little doubt that a new Microsoft is emerging, one that's committed to investing in cross-platform development using open source best practices that allow the development community to share in creating great software. For this reason, you'll soon be able to read about early design discussions on C# 7.0, because this time the discussion will take place in an open source forum. ■

Mark Michaelis *(itl.tc/Mark) is the founder of IntelliTect. He also serves as the chief technical architect and trainer. Since 1996, he has been a Microsoft MVP for C#, Visual Studio Team System (VSTS), and the Windows SDK, and he was recognized as a Microsoft Regional Director in 2007. He also serves on several Microsoft software designreview teams, including C#, the Connected Systems Division and VSTS. Michaelis speaks at developer conferences, has written numerous articles and books, and is currently working on the next edition of "Essential C#" (Addison-Wesley Professional).*

Thanks to the following Microsoft technical expert for reviewing this article: *Mads Torgersen*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

vslive.com/lasvegas

Las Vegas MARCH 16 - 20

BALLY'S HOTEL & CASINO LAS VEGAS, NV

Code on the Strip

Visual Studio Live!'s first stop on its 2015 Code Trip is Las Vegas, situated fittingly near Historic Route 66. Developers, software architects, engineers, and designers will cruise onto the Strip for five days of unbiased and cutting-edge education on the Microsoft Platform. Navigate the .NET Highway with industry experts and Microsoft insiders in 60+ sessions and fun networking events – all designed to make you better at your job.

SUPPORTED BY

Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA

NAVIGATE THE .NET HIGHWAY

TRACKS INCLUDE:

- Visual Studio / .NET
- JavaScript/HTML5
- ASP.NET
- Cross-Platform Mobile Development
- Windows 8.1/WinRT
- Database and Analytics
- Cloud Computing
- Windows Phone

Register NOW and Save $400!

Use promo code VSLJAN2

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

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

vslive.com/lasvegas

Use Roslyn to Write a Live Code Analyzer for Your API

Alex Turner

For 10 years now, Visual Studio Code Analysis has provided build-time analysis of your C# and Visual Basic assemblies, running a specific set of FxCop rules written for the Microsoft .NET Framework 2.0. These rules are great at helping you avoid general pitfalls in your code, but they're targeted at the problems developers hit back in 2005. What about today's new language features and APIs?

With live, project-based code analyzers in Visual Studio 2015 Preview, API authors can ship domain-specific code analysis as part of their NuGet packages. Because these analyzers are powered by the .NET Compiler Platform (code-named "Roslyn"), they can produce warnings in your code as you type even before you've finished the line—no more waiting to build your code to find out you made a mistake. Analyzers can also surface an automatic code fix through the new Visual Studio light bulb prompt to let you clean up your code immediately.

This article discusses prerelease versions of Visual Studio 2015 and the .NET Compiler Platform ("Roslyn") SDK. All related information is subject to change.

This article discusses:

- Setting up to write an analyzer
- Using the analyzer template
- Inspecting code with the Syntax Visualizer
- Building a diagnostic
- Use cases for building your own diagnostic analyzer

Technologies discussed:

.NET Compiler Platform, Visual Studio 2015 Preview

With live, project-based code analyzers in Visual Studio 2015 Preview, API authors can ship domain-specific code analysis as part of their NuGet packages.

Many of these live code analyzers weigh in at just 50 to 100 lines of code, and you don't have to be a compiler jock to write them. In this article I'll show how to write an analyzer that targets a common coding problem hit by those using .NET regular expressions (regex): How to make sure the regex pattern you wrote is syntactically valid before you run your app. I'll do this by showing how to write an analyzer that contains a diagnostic to point out invalid regex patterns. I'll follow up this article with a second installment later, where I'll show how to add in a code fix to clean up the errors your analyzer finds.

Getting Ready

To get started, make sure you have the necessary Visual Studio 2015 Preview bits:

- Set up a box with Visual Studio 2015 Preview in one of these two ways:
  1. Install Visual Studio 2015 Preview from aka.ms/vs2015preview.
  2. Grab a prebuilt Azure VM image from aka.ms/vs2015azuregallery.
- Install the Visual Studio 2015 Preview SDK from aka.ms/vs2015preview. You'll need to do this even if you're using the Azure VM image.
- Install the SDK Templates VSIX package from aka.ms/roslynsdktemplates to get the .NET Compiler Platform project templates.
- Install the Syntax Visualizer VSIX package from aka.ms/roslynsyntaxvisualizer to get a Syntax Visualizer tool window to help explore the syntax trees you'll be analyzing.
  - You also need to install the Syntax Visualizer into the experimental Visual Studio sandbox you'll use to debug your analyzer. I'll walk through how to do that in the Syntax Visualizer section later.

Exploring the Analyzer Template

Once you're up and running with Visual Studio 2015, the Visual Studio SDK and the necessary VSIX packages, check out the project template for building an analyzer.

Inside Visual Studio 2015, go to File | New Project | C# | Extensibility and choose the Diagnostic with Code Fix (NuGet + VSIX) template, as shown in **Figure 1**. You can create analyzers in Visual Basic, as well, but for this article I'll use C#. Be sure the target framework is set to .NET Framework 4.5 at the top. Give your project the name RegexAnalyzer and select OK to create the project.

You'll see a set of three projects the template generates:

- **RegexAnalyzer:** This is the main project that builds the analyzer DLL containing the diagnostics and code fix. Building this project also produces a project-local NuGet package (.nupkg file) containing the analyzer.
- **RegexAnalyzer.VSIX:** This is the project that bundles your analyzer DLL into a Visual Studio-wide extension package (.vsix file). If your analyzer doesn't need to add warnings that affect builds, you can choose to distribute this .vsix file rather than the NuGet package. Either way, the .vsix project lets you press F5 and test your analyzer in a separate (debugee) instance of Visual Studio.
- **RegexAnalyzer.Test:** This is a unit test project that lets you make sure your analyzer is producing the right diagnostics and fixes, without running that debugee instance of Visual Studio each time.

If you open the DiagnosticAnalyzer.cs file in the main project, you can see the default code in the template that produces diagnostics. This diagnostic does something a bit silly—it "squiggles" any type names that have lowercase letters. However, because most programs will have such type names, this lets you easily see the analyzer in action.

Make sure that the RegexAnalyzer.VSIX project is the startup project and press F5. Running the VSIX project loads an experimental sandbox copy of Visual Studio, which lets Visual Studio keep track of a separate set of Visual Studio extensions. This is useful as you develop your own extensions and need to debug Visual Studio with Visual Studio. Because the debuggee Visual Studio instance is a new experimental sandbox, you'll see the same dialogs you got when you first ran Visual Studio 2015. Click through them as normal. You might also see some delays as you download debugging symbols for Visual Studio itself. After the first time, Visual Studio should cache those symbols for you.

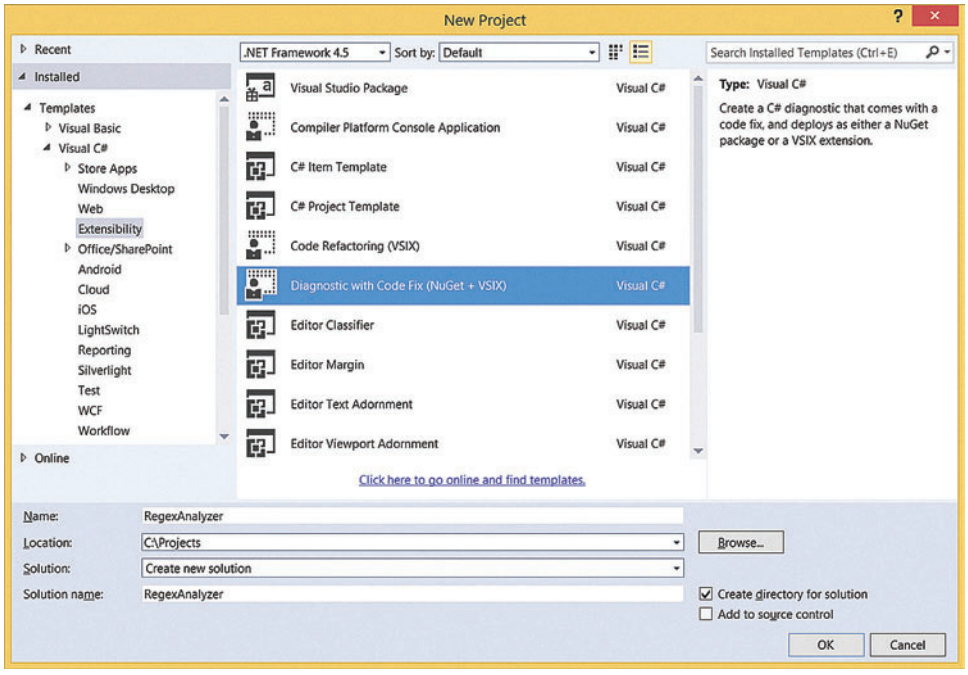


Figure 1 **Creating the Analyzer Project**

Once your debuggee Visual Studio instance is running, use it to create a C# console application. Because the analyzer you're debugging is the Visual Studio-wide .vsix extension, you should see a green squiggle appear on the Program class definition within a few seconds. If you hover over this squiggle or look at the Error List, you'll see the message, "Type name 'Program' contains lowercase letters," as shown in **Figure 2**. When you click on the squiggle, you'll see a light bulb icon appear on the left. Clicking on the light bulb icon will show a "Make uppercase" fix that cleans up the diagnostic by changing the type name to be uppercase.

You're able to debug the analyzer from here, as well. In your main instance of Visual Studio,

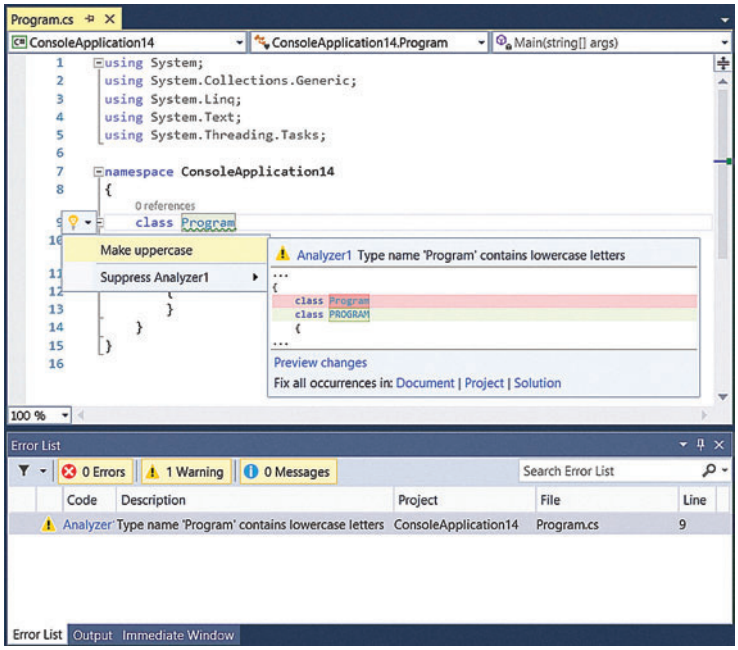


Figure 2 **The Code Fix from the Analyzer Template**

set a breakpoint inside the Initialize method within DiagnosticAnalyzer.cs. As you type in the editor, analyzers continually recalculate diagnostics. The next time you type within Program.cs in your debuggee Visual Studio instance, you'll see the debugger stop at that breakpoint.

Keep the console application project open for now, as you'll use it further in the next section.

Inspecting the Relevant Code Using the Syntax Visualizer

Now that you're oriented in the analyzer template, it's time to start planning what code patterns you'll look for in the analyzed code to decide when to squiggle.

Your goal is to introduce an error that shows up when you write an invalid regex pattern. First, within the console application's Main method, add the following line that calls Regex.Match with an invalid regex pattern:

```
Regex.Match("my text", @"\pXXX");
```

Looking at this code and hovering on Regex and Match, you can work out the conditions for when you want to generate a squiggle:

- There's a call to the Regex.Match method.
- The Regex type involved is the one from the System.Text.RegularExpressions namespace.
- The second parameter of the method is a string literal that represents an invalid regular expression pattern. In practice, the expression might also be a variable or constant reference—or a computed string—but for this initial version of the analyzer, I'll focus first on string literals. It's often best to get an analyzer working end to end for a simple case before you move on to support more code patterns.

So, how do you translate these simple constraints into .NET Compiler Platform code? The Syntax Visualizer is a great tool to help figure that out.

You'll want to install the visualizer within the experimental sandbox you've been using to debug analyzers. You might have installed the visualizer earlier, but the installer just installs the package into your main Visual Studio.

While you're still in your debuggee Visual Studio instance, open Tools | Extensions & Updates | Online, and search the Visual Studio Gallery for "syntax visualizer." Download and install the .NET Compiler Platform Syntax Visualizer package. Then choose Restart Now to restart Visual Studio.

Once Visual Studio restarts, open up the same console application project and open the Syntax Visualizer by choosing View | Other Windows | Roslyn Syntax Visualizer. You can now move the caret around the editor and watch as the Syntax Visualizer shows you the relevant part of the syntax tree. **Figure 3** shows the view for the Regex.Match invocation expression you're interested in here.

The Parts of the Syntax Tree As you browse around in the syntax tree, you'll see various elements.

The blue nodes in the tree are the syntax nodes, representing the logical tree structure of your code after the compiler has parsed the file.

The green nodes in the tree are the syntax tokens, the individual words, numbers and symbols the compiler found when it read the source file. Tokens are shown in the tree under the syntax nodes to which they belong.

The red nodes in the tree are the trivia, representing everything else that's not a token: whitespace, comments and so on. Some compilers throw this information away, but the .NET Compiler Platform holds onto it, so your code fix can maintain the trivia as needed when your fix changes the user's code.

By selecting code in the editor, you can see the relevant nodes in the tree, and vice versa. To help visualize the nodes you care about, you can right-click on the InvocationExpression in the Syntax Visualizer tree and choose View Directed Syntax Graph. This will generate a .dgml diagram that visualizes the tree structure below the selected node, as shown in **Figure 4**.

> Your goal is to introduce an error that shows up when you write an invalid regex pattern.

In this case, you can see you're looking for an InvocationExpression that's a call to Regex.Match, where the ArgumentList has a second Argument node that contains a StringLiteralExpression. If the string value represents an invalid regex pattern, such as "\pXXX," you've found the span to squiggle. You've now gathered most of the information needed to write your diagnostic analyzer.

Symbols and the Semantic Model: Going Beyond the Syntax Tree While the syntax nodes, tokens and trivia shown in the Syntax Visualizer represent the full text of the file, they don't tell you everything. You also need to know what each identifier in the code is actually referencing. For example, you know this invocation is a call

to a Match method on a Regex type with two parameters, but you don't know what namespace that Regex type is in or which overload of Match is called. Discovering exactly what definitions are referenced requires the compilers to analyze the identifiers in the context of their nearby using directives.

Answering those kinds of questions requires you to ask the semantic model to give you the symbol associated with a given expression node. Symbols represent the logical entities that your code defines, such as your types and methods. The process of figuring out the symbol referenced by a given expression is known as binding. Symbols can also represent the entities you consume from referenced libraries, such as the Regex type from the Base Class Library (BCL).

If you right-click on the InvocationExpression and choose View Symbol, the property grid below fills with information from the method symbol of the invoked method, as shown in **Figure 5**.

In this case you can look at the OriginalDefinition property and see that the invocation refers to the System.Text.RegularExpressions.Regex.Match method, and not a method on some other Regex type. The last piece of the puzzle in writing your diagnostic is to bind that invocation and check that the symbol you get back matches the string System.Text.RegularExpressions.Regex.Match.

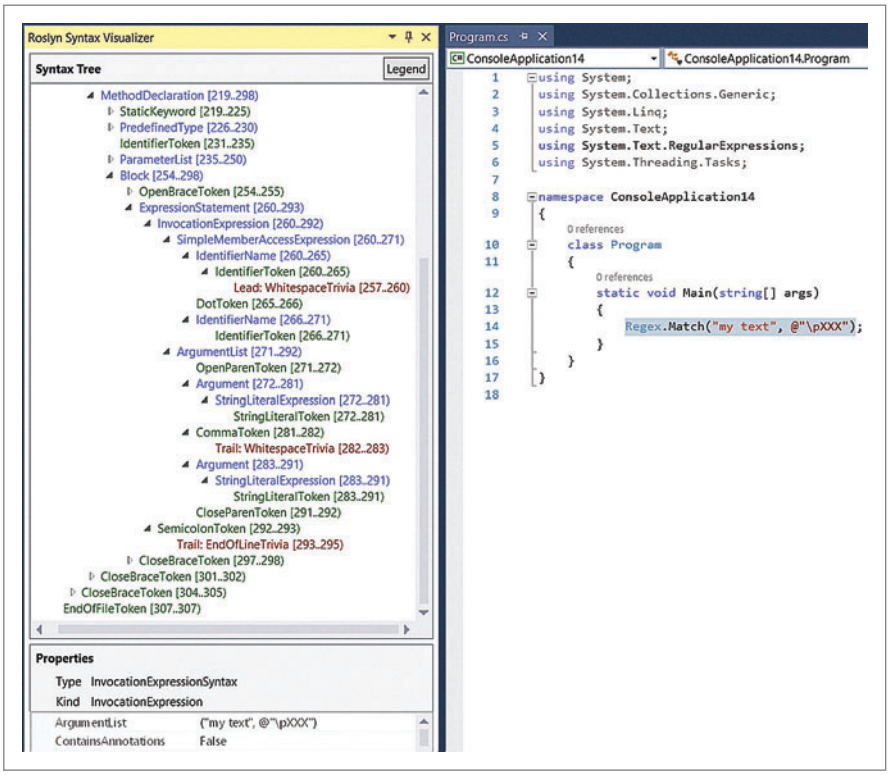


Figure 3 **The Syntax Visualizer in Action for the Target Invocation Expression**

Building the Diagnostic

Now that you've got your strategy, close the debuggee Visual Studio instance and return to your analyzer project to start building your diagnostic.

The SupportedDiagnostics Property Open up the DiagnosticAnalyzer.cs file and look at the four string constants near the top. This is where you define the metadata for your diagnostic rule. Even before your analyzer produces any squiggles, the Ruleset Editor and other Visual Studio features will use this metadata to know the details of the diagnostics your analyzer might produce.

Update these strings to match the Regex diagnostic you plan to produce:

```
public const string DiagnosticId = "Regex";
internal const string Title = "Regex error parsing string argument";
internal const string MessageFormat = "Regex error {0}";
internal const string Category = "Syntax";
```

The diagnostic ID and filled-in message string are shown to users in the Error List when this diagnostic is produced. A user can also use the ID in his source code in a #pragma directive to suppress an instance of the diagnostic. In the follow-up article, I'll show how to use this ID to associate your code fix with this rule.

In the line declaring the Rule field, you can also update the severity of the diagnostics you'll be producing to be errors rather than warnings. If the regex string doesn't parse, the Match method will definitely throw an exception at run time, and you should block the build as you would for a C# compiler error. Change the rule's severity to DiagnosticSeverity.Error:

```
internal static DiagnosticDescriptor Rule =
  new DiagnosticDescriptor(DiagnosticId, Title, MessageFormat,
  Category, DiagnosticSeverity.Error, isEnabledByDefault: true);
```

This is also the line where you decide whether the rule should be enabled by default. Your analyzer can define a larger set of rules that are off by default, and users can choose to opt in to some or all of the rules. Leave this rule enabled by default.

> The process of figuring out the symbol referenced by a given expression is known as binding.

The SupportedDiagnostics property returns this DiagnosticDescriptor as the single element of an immutable array. In this case, your analyzer will only produce one kind of diagnostic, so there's nothing to change here. If your analyzer can produce multiple kinds of diagnostics, you could make multiple descriptors and return them all from SupportedDiagnostics.

Initialize Method The main entry point for your diagnostic analyzer is the Initialize method. In this method, you register a set of actions to handle various events the compiler fires as it walks through your code, such as finding various syntax nodes, or encountering a declaration of a new symbol. The silly default analyzer you get from the template calls RegisterSymbolAction to find out when type symbols change or are introduced. In that case, the

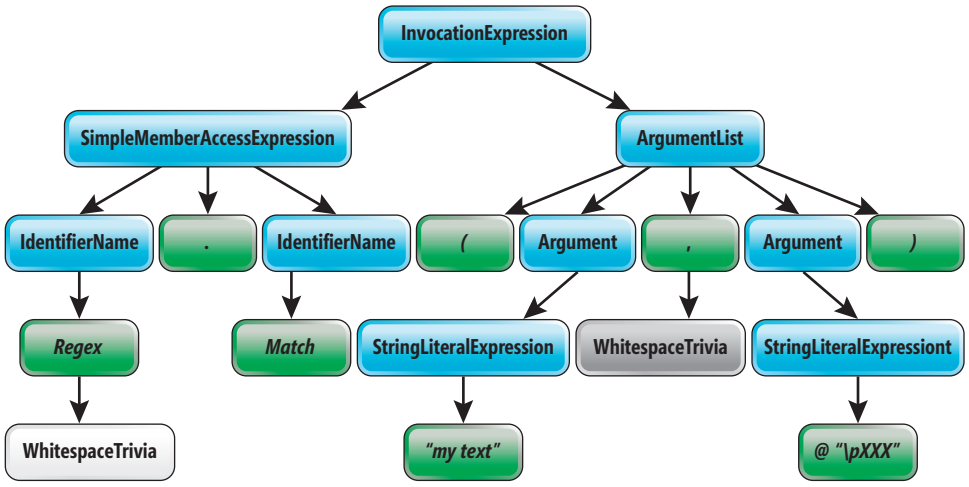


Figure 4 **Syntax Graph for the Target Invocation Expression**

symbol action lets the analyzer look at each type symbol to see if it indicates a type with a bad name that needs a squiggle.

In this case, you need to register a SyntaxNode action to find out when there's a new call to Regex.Match. Recall from exploring in the Syntax Visualizer that the specific node kind you're looking for is InvocationExpression, so replace the Register call in the Initialize method with the following call:

```
context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.InvocationExpression);
```

The regex analyzer only needed to register a syntax node action that produces diagnostics locally. What about more complex analysis that gathers data across multiple methods? See "Handling Other Register Methods" in this article for more on this.

AnalyzeNode Method Delete the template's AnalyzeSymbol method, which you no longer need. In its place, you'll create an AnalyzeNode method. Click on AnalyzeNode within the RegisterSyntaxNodeAction call and press Ctrl+Dot to pull up the new light bulb menu. From there, choose Generate method to create an AnalyzeNode method with the right signature. In the generated AnalyzeNode method, change the parameter's name from "obj" to "context."

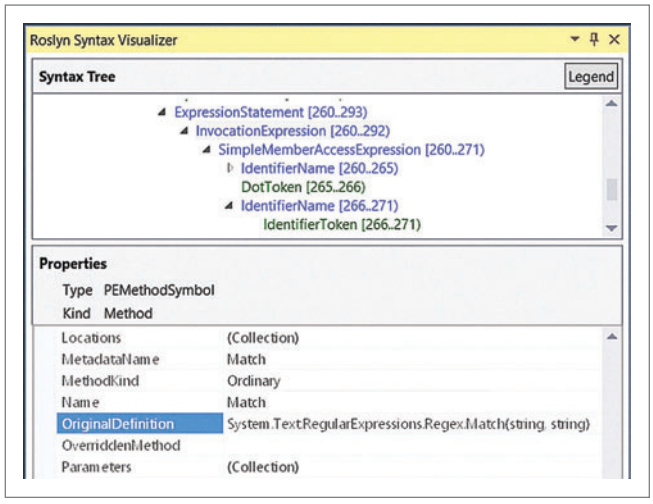


Figure 5 **Viewing the Regex.Match Method Symbol in the Syntax Visualizer**

Now that you've gotten to the core of your analyzer, it's time to examine the syntax node in question to see if you should surface a diagnostic.

First, you should press Ctrl+F5 to launch the debuggee instance of Visual Studio again (this time without debugging). Open up the console application you were just looking at and make sure the Syntax Visualizer is available. You'll look at the visualizer a few times to find relevant details as you build the AnalyzeNode method.

Getting the Target Node Your first step in the AnalyzeNode method is to take the node object you're analyzing and cast it to the relevant type. To find that type, use the Syntax Visualizer you just opened. Select the Regex.Match call and navigate in the syntax tree to the InvocationExpression node. You can see just above the property grid that InvocationExpression is the kind of node, while the type is InvocationExpressionSyntax.

You can test a node's type with a type check or test its specific kind with the IsKind method. However, here you can guarantee that a cast will succeed without either test, because you asked for the particular kind in the Initialize method. The node your action analyzes is available from the Node property on the context parameter:

```
var invocationExpr = (InvocationExpressionSyntax)context.Node;
```

Now that you have the invocation node, you need to check whether it's a Regex.Match call that needs a squiggle.

Check No. 1: Is This a Call to a Match Method? The first check you need is to make sure this invocation is a call to the correct Regex.Match. Because this analyzer will run on every keystroke in the editor, it's a good idea to perform the quickest tests first and ask more expensive questions of the API only if those initial tests pass.

The cheapest test is to see whether the invocation syntax is a call to a method named Match. That can be determined before the compiler has done any special work to figure out which particular Match method this is.

Looking back at the Syntax Visualizer, you see that the InvocationExpression has two main child nodes, the SimpleMemberAccessExpression and the ArgumentList. By selecting the Match identifier in the editor as shown in **Figure 6**, you can see that the node you're looking for is the second IdentifierName within the SimpleMemberAccessExpression.

As you build an analyzer, you'll quite often be digging into syntax and symbols like this to find the relevant types and property values you need to reference in your code. When building analyzers, it's convenient to keep a target project with the Syntax Visualizer handy.

Back in your analyzer code, you can browse the members of invocationExpr in IntelliSense and find a property that corresponds to each of the InvocationExpression's child nodes, one named Expression and one named ArgumentList. In this case, you want the property named Expression. Because the part of an invocation

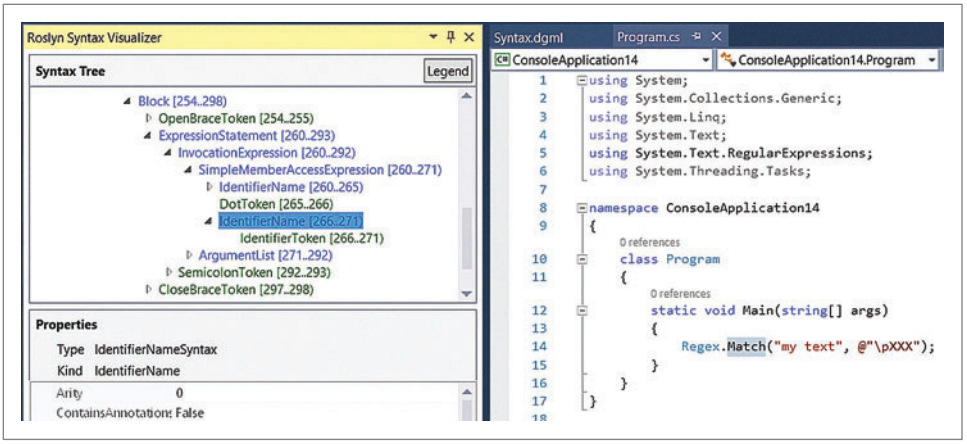


Figure 6 **Finding the Match Identifier in the Syntax Tree**

that's before the argument list can have many forms (for example, it might be a delegate invocation of a local variable), this property returns a general base type, ExpressionSyntax. From the Syntax Visualizer, you can see that the concrete type that you expect is a MemberAccessExpressionSyntax, so cast it to that type:

```
var memberAccessExpr =
  invocationExpr.Expression as MemberAccessExpressionSyntax;
```

You see a similar breakdown when you dig into the properties on memberAccessExpr. There's an Expression property that represents the arbitrary expression before the dot, and a Name property that represents the identifier to the right of the dot. Because you want to check to see if you're calling a Match method, check the string value of the Name property. For a syntax node, getting the string value is a quick way to get the source text for that node. You can use the new C# "?." operator to handle the case where the expression you're analyzing wasn't actually a member access, causing the previous line's "as" clause to return a null value:

```
if (memberAccessExpr?.Name.
  ToString() != "Match") return;
```

If the method being called isn't named Match, you simply bail out. Your analysis is complete at minimal cost and there's no diagnostic to generate.

Check No. 2: Is This a Call to the Real Regex.Match Method?

If the method *is* named Match, do a more involved check that asks the compiler to determine precisely which Match method the code is calling. Determining the exact Match requires asking the context's semantic model to bind this expression to get the referenced symbol.

Call the GetSymbolInfo method on the semantic model and pass it the expression for which you want the symbol:

```
var memberSymbol =
  context.SemanticModel.GetSymbolInfo(memberAccessExpr).Symbol as IMethodSymbol;
```

The symbol object you get back is the same one you can preview in the Syntax Visualizer by right-clicking the SimpleMemberAccessExpression and choosing View Symbol. In this case, you're choosing to cast the symbol to the common IMethodSymbol interface. This interface is implemented by the internal PEMethodSymbol type mentioned for that symbol in the Syntax Visualizer.

Figure 7 **The Complete Code for DiagnosticAnalyzer.cs**

```
using System;
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;

namespace RegexAnalyzer
{
  [DiagnosticAnalyzer(LanguageNames.CSharp)]
  public class RegexAnalyzerAnalyzer : DiagnosticAnalyzer
  {
    public const string DiagnosticId = "Regex";
    internal const string Title = "Regex error parsing string argument";
    internal const string MessageFormat = "Regex error {0}";
    internal const string Category = "Syntax";

    internal static DiagnosticDescriptor Rule =
      new DiagnosticDescriptor(DiagnosticId, Title, MessageFormat,
      Category, DiagnosticSeverity.Error, isEnabledByDefault: true);

    public override ImmutableArray<DiagnosticDescriptor>
      SupportedDiagnostics { get { return ImmutableArray.Create(Rule); } }

    public override void Initialize(AnalysisContext context)
    {
      context.RegisterSyntaxNodeAction(
        AnalyzeNode, SyntaxKind.InvocationExpression);
    }

    private void AnalyzeNode(SyntaxNodeAnalysisContext context)
    {
      var invocationExpr = (InvocationExpressionSyntax)context.Node;

      var memberAccessExpr =
        invocationExpr.Expression as MemberAccessExpressionSyntax;
      if (memberAccessExpr?.Name.ToString() != "Match") return;

      var memberSymbol = context.SemanticModel.
        GetSymbolInfo(memberAccessExpr).Symbol as IMethodSymbol;
      if (!memberSymbol?.ToString().StartsWith(
        "System.Text.RegularExpressions.Regex.Match") ?? true) return;

      var argumentList = invocationExpr.ArgumentList as ArgumentListSyntax;
      if ((argumentList?.Arguments.Count ?? 0) < 2) return;

      var regexLiteral =
        argumentList.Arguments[1].Expression as LiteralExpressionSyntax;
      if (regexLiteral == null) return;

      var regexOpt = context.SemanticModel.GetConstantValue(regexLiteral);
      if (!regexOpt.HasValue) return;

      var regex = regexOpt.Value as string;
      if (regex == null) return;

      try
      {
        System.Text.RegularExpressions.Regex.Match("", regex);
      }
      catch (ArgumentException e)
      {
        var diagnostic =
          Diagnostic.Create(Rule, regexLiteral.GetLocation(), e.Message);
        context.ReportDiagnostic(diagnostic);
      }
    }
  }
}
```

Now that you have the symbol, you can compare it against the fully qualified name you expect from the real Regex.Match method. For a symbol, getting the string value will give you its fully qualified name. You don't really care which overload you're calling yet, so you can just check as far as the word Match:

```
if (!memberSymbol?.ToString().
  StartsWith("System.Text.RegularExpressions.Regex.Match") ?? true) return;
```

As with your previous test, check to see if the symbol matches the name you expect, and if it doesn't match, or if it wasn't actually a method symbol, bail out. Though it might feel a bit odd to operate on strings here, string comparisons are a common operation within compilers.

> There are many domains in your everyday coding where writing a diagnostic analyzer can be quick and useful.

The Remaining Checks Now your tests are starting to fall into a rhythm. At each step, you dig a bit further into the tree and check either the syntax nodes or the semantic model to test if you're still in an error situation. Each time, you can use the Syntax Visualizer to see what types and property values you expect, so you know in which case to return and in which case to continue. You'll follow this pattern to check the next few conditions.

Make sure the ArgumentList has at least two arguments:

```
var argumentList = invocationExpr.ArgumentList as ArgumentListSyntax;
if ((argumentList?.Arguments.Count ?? 0) < 2) return;
```

Then make sure the second argument is a LiteralExpression, because you're expecting a string literal:

```
var regexLiteral =
  argumentList.Arguments[1].Expression as LiteralExpressionSyntax;
if (regexLiteral == null) return;
```

Finally, once you know it's a literal, you can ask the semantic model to give you its constant compile-time value, and make sure it's specifically a string literal:

```
var regexOpt = context.SemanticModel.GetConstantValue(regexLiteral);
if (!regexOpt.HasValue) return;

var regex = regexOpt.Value as string;
if (regex == null) return;
```

Validating the Regex Pattern At this point, you've got all the data you need. You know you're calling Regex.Match, and you've got the string value of the pattern expression. So how do you validate it?

Simply call the same Regex.Match method and pass in that pattern string. Because you're only looking for parse errors in the pattern string, you can pass an empty input string as the first argument. Make the call within a try-catch block so you can catch the ArgumentException that Regex.Match throws when it sees an invalid pattern string:

```
try
{
  System.Text.RegularExpressions.Regex.Match("", regex);
}
catch (ArgumentException e)
{

}
```

If the pattern string parses without error, your AnalyzeNode method will exit normally and there's nothing to report. If there's a parse error, you'll catch the argument exception—you're ready to report a diagnostic!

Reporting a Diagnostic Inside the catch block, you use the Rule object you filled in earlier to create a Diagnostic object, which represents one particular squiggle you want to produce. Each diagnostic needs two main things specific to that instance: the span of code that should be squiggled and the fill-in strings to plug into the message format you defined earlier:

```
var diagnostic =
  Diagnostic.Create(Rule,
  regexLiteral.GetLocation(), e.Message);
```

In this case, you want to squiggle the string literal so you pass in its location as the span for the diagnostic. You also pull out the exception message that describes what was wrong with the pattern string and include that in the diagnostic message.

The last step is to take this diagnostic and report it back to the context passed to AnalyzeNode, so Visual Studio knows to add a row to the Error List and add a squiggle in the editor:

```
context.ReportDiagnostic(diagnostic);
```

Your code in DiagnosticAnalyzer.cs should now look like **Figure 7**.

Trying It Out That's it—your diagnostic is now complete! To try it, just press F5 (make sure RegexAnalyzer.VSIX is the startup project) and reopen the console application in the debuggee instance of

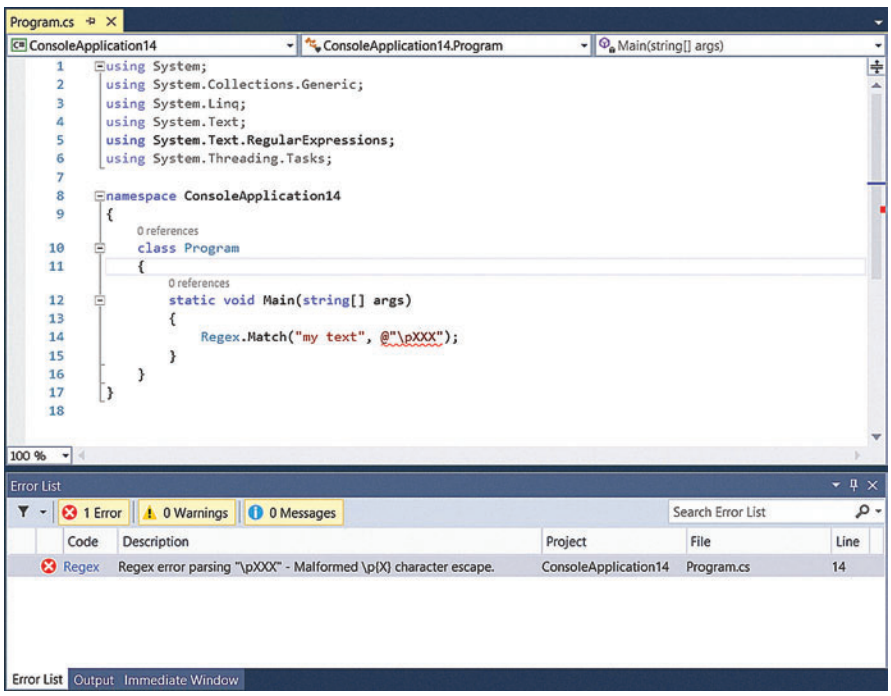


Figure 8 **Trying Out Your Diagnostic Analyzer**

Visual Studio. You should soon see a red squiggle on the pattern expression pointing out why it failed to parse, as shown in **Figure 8**.

If you see the squiggle, congratulations! If not, you can set a breakpoint inside the AnalyzeNode method, type a character inside the pattern string to trigger reanalysis, and then step through the analyzer code to see where the analyzer is bailing out early. You can also check your code against **Figure 7**, which shows the complete code for DiagnosticAnalyzer.cs.

Use Cases for Diagnostic Analyzers

To recap, starting from the analyzer template and writing about 30 lines of new code, you were able to identify and provide a squiggle for a real issue in your users' code. Most important, doing so didn't require you to become a deep expert in the operations of the C# compiler. You were able to stay focused on your target domain of regular expressions and use the Syntax Visualizer to guide you to the small set of syntax nodes and symbols relevant to your analysis.

There are many domains in your everyday coding where writing a diagnostic analyzer can be quick and useful:

- As a developer or lead on a team, you might see others make the same mistakes often when you do code reviews. Now, you can write a simple analyzer that squiggles those anti-patterns and check the analyzer into source control, ensuring that anyone who introduces such a bug will notice it as they're typing.
- As the maintainer of a shared layer that defines business objects for your organization, you might have business rules for correct use of these objects that are hard to encode in the type system, especially if they involve numerical values or if they involve steps in a process where some operations should always come before others. Now you can enforce these softer rules that govern use of your layer, picking up where the type system leaves off.

With the .NET Compiler Platform, Microsoft has done the heavy lifting to expose deep language understanding and rich code analysis for C# and Visual Basic.

- As the owner of an open source or commercial API package, you might be tired of answering the same questions repeatedly in the forums. You might even have written white papers and documentation and found that many of your customers continue to hit those same issues, as they haven't read what you wrote. Now you can bundle your API and the relevant code analysis into one NuGet package, ensuring that everyone using your API sees the same guidance from the start.

Hopefully, this article has inspired you to think about the analyzers you'll want to build to enhance your own projects. With the .NET Compiler Platform, Microsoft has done the heavy lifting to expose deep language understanding and rich code analysis for C# and Visual Basic—the rest is up to you!

What's Next?

So now you have Visual Studio showing error squiggles under invalid regex patterns. Can you do more?

If you've got the regular expressions domain knowledge to see not just what's wrong with a pattern string but also how to fix it, you can suggest a fix in the light bulb, as you saw in the template's default analyzer.

In the next article, I'll show how to write that code fix, as you learn how to make changes to your syntax trees. Stay tuned! ■

Handling Other Register Methods

You can dig into the Initialize method's context parameter to see the full set of Register methods you can call. The methods in **Figure A** let you hook various events in the compiler's pipeline.

Figure A **Register Methods To Hook Various Events**

RegisterSyntaxNodeAction	Triggered when a particular kind of syntax node has been parsed
RegisterSymbolAction	Triggered when a particular kind of symbol has been analyzed
RegisterSyntaxTreeAction	Triggered when the file's whole syntax tree has been parsed
RegisterSemanticModelAction	Triggered when a semantic model is available for the whole file
RegisterCodeBlockStartAction RegisterCodeBlockEndAction	Triggered before/after analysis of a method body or other code block
RegisterCompilationStartAction RegisterCompilationEndAction	Triggered before/after analysis of the entire project

A key point to note is that a top-level action registered with any Register method should never stash any state in instance fields on the analyzer type. Visual Studio will reuse one instance of that analyzer type for the whole Visual Studio session to avoid repeated allocations. Any state you store and reuse is likely to be stale when analyzing a future compilation and could even be a memory leak if it keeps old syntax nodes or symbols from being garbage collected.

If you need to retain state across actions, you should call RegisterCodeBlockStartAction or RegisterCompilationStartAction and store the state as locals within that action method. The context object passed to those actions lets you register nested actions as lambda expressions, and these nested actions can close over the locals in the outer actions in order to keep state.

Alex Turner *is a senior program manager for the Managed Languages team at Microsoft, where he's been brewing up C# and Visual Basic goodness on the .NET Compiler Platform ("Roslyn") project. He graduated with a master's degree in Computer Science from Stony Brook University and has spoken at Build, PDC, TechEd, TechDays and MIX conferences.*

Thanks to the following Microsoft technical experts for reviewing this article: *Bill Chiles and Lucian Wischik*

Visual C++ 2015 Brings Modern C++ to the Windows API

Kenny Kerr

Visual C++ 2015 is the culmination of a huge effort by the C++ team to bring modern C++ to the Windows platform. Over the last few releases, Visual C++ has steadily gained a rich selection of modern C++ language and library features that together make for an absolutely amazing environment in which to build universal Windows apps and components. Visual C++ 2015 builds on the remarkable progress introduced in those earlier releases and provides a mature compiler that supports much of C++11 and a subset of C++ 2015. You might argue about the level of completeness, but I think it's fair to say that the compiler supports the most important language features, enabling modern C++ to usher in a new era of library development for Windows. And that really is the key. As long as the compiler supports the development of efficient and elegant libraries, developers can get on with building great apps and components.

This article discusses:

- Modern C++
- Standard C++
- The Windows Runtime
- Component Object Model

Technologies discussed:

Visual C++ 2015, Windows Runtime

Rather than giving you a boring list of new features or providing a high-level whirlwind tour of capabilities, I'll instead walk you through the development of some traditionally complex code that is now, frankly, quite enjoyable to write, thanks to the maturity of the Visual C++ compiler. I'm going to show you something that's intrinsic to Windows and at the heart of practically every significant current and future API.

It's somewhat ironic that C++ is finally modern enough for COM. Yes, I'm talking about the Component Object Model that has been the foundation for much of the Windows API for years, and continues as the foundation for the Windows Runtime. While COM is undeniably tied to C++ in terms of its original design, borrowing much from C++ in terms of binary and semantic conventions, it has never been entirely elegant. Parts of C++ that were not deemed portable enough, such as dynamic_cast, had to be eschewed in favor of portable solutions that made C++ implementations more challenging to develop. Many solutions have been provided over the years to make COM more palatable for C++ developers. The C++/CX language extension is perhaps the most ambitious thus far by the Visual C++ team. Ironically, these efforts to improve Standard C++ support have left C++/CX in the dust and really make a language extension redundant.

To prove this point I'm going to show you how to implement IUnknown and IInspectable entirely in modern C++. There is nothing modern or appealing about these two beauties. IUnknown

continues to be the central abstraction for prominent APIs like DirectX. And these interfaces—with IInspectable deriving from IUnknown—sit at the heart of the Windows Runtime. I'm going to show you how to implement them without any language extensions, interface tables or other macros—just efficient and elegant C++ with oodles of rich type information that lets the compiler and developer have a great conversation about what needs to be built.

The main challenge is to come up with a way to describe the list of interfaces that a COM or Windows Runtime class intends to implement, and to do so in a way that's convenient for the developer and accessible to the compiler. Specifically, I need to make this list of types available such that the compiler can interrogate and even enumerate the interfaces. If I can pull that off I might be able to get the compiler to generate the code for the IUnknown QueryInterface method and, optionally, the IInspectable GetIids method, as well. It's these two methods that pose the biggest challenge. Traditionally, the only solutions have involved language extensions, hideous macros or a lot of hard-to-maintain code.

Both method implementations require a list of interfaces that a class intends to implement. The natural choice for describing such a list of types is a variadic template:

```
template <typename ... Interfaces>
class __declspec(novtable) Implements : public Interfaces ...
{
};
```

The novtable __declspec extended attribute keeps any constructors and destructors from having to initialize the vfptr in such abstract classes, which often means a significant reduction in code size. The Implements class template includes a template parameter pack, thus making it a variadic template. A parameter pack is a template parameter that accepts any number of template arguments. The trick is that parameter packs are normally used to allow functions to accept any number of arguments, but in this case I'm describing a template whose arguments will be interrogated purely at compile time. The interfaces will never appear in a function parameter list.

One use of those arguments is already plain to see. The parameter pack expands to form the list of public base classes. Of course, I'm still responsible for actually implementing those virtual functions, but at this point I can describe a concrete class that implements any number of interfaces:

```
class Hen : public Implements<IHen, IHen2>
{
};
```

Because the parameter pack is expanded to designate the list of base classes, it's equivalent to what I might have written myself, as follows:

```
class Hen : public IHen, public IHen2
{
};
```

The beauty of structuring the Implements class template in this way is that I can now insert the implementation of various boilerplate code into the Implements class template while the developer of the Hen class can use this unobtrusive abstraction and largely ignore the magic behind it all.

So far, so good. Now I'll consider the implementation of IUnknown itself. I should be able to implement it entirely inside the Implements class template, given the type of information the compiler now has at its disposal. IUnknown provides two facilities that are as essential to COM classes as oxygen and water are to humans. The first and perhaps simpler of the two is reference counting and is the means by which COM objects track their lifetime. COM prescribes a form of intrusive reference counting whereby each object is responsible for managing its own lifetime based on its awareness of how many outstanding references exist. This is in contrast to a reference counting smart pointer such as the C++11 shared_ptr class template, where the object has no knowledge of its shared ownership. You might argue about the pros and cons of the two approaches, but in practice the COM approach is often more efficient and it's just the way COM works so you have to deal with it. If nothing else, you'll likely agree that it's a horrible idea to wrap a COM interface inside a shared_ptr!

I'll begin with the only runtime overhead introduced by the Implements class template:

```
protected:

  unsigned long m_references = 1;

  Implements() noexcept = default;

  virtual ~Implements() noexcept
  {}
```

The defaulted constructor isn't really overhead in itself; it simply ensures the resulting constructor—which will initialize the reference count—is protected rather than public. Both the reference count and the virtual destructor are protected. Making the reference count accessible to derived classes allows for more complex class composition. Most classes can simply ignore this, but do notice that I'm initializing the reference count to one. This is in contrast to popular wisdom that suggests the reference count should initially be zero because no references have been handed out yet. That approach was popularized by ATL and undoubtedly influenced by Don Box's Essential COM, but it's quite problematic, as a study of the ATL source code can well attest. Starting with the assumption that the ownership of the reference will immediately be assumed by a caller or attached to a smart pointer provides for a far less error-prone construction process.

A virtual destructor is a tremendous convenience in that it allows the Implements class template to implement the reference counting rather than forcing the concrete class itself to provide the implementation. Another option would be to use the curiously recurring template pattern to avoid the virtual function. Normally I'd prefer such an approach, but it would complicate the abstraction slightly, and because a COM class by its very nature has a vtable, there's no compelling reason to avoid a virtual function here. With these primitives in place, it becomes a simple matter to implement both AddRef and Release inside the Implements class template. First, the AddRef method can simply use the InterlockedIncrement intrinsic to bump up the reference count:

```
virtual unsigned long __stdcall AddRef() noexcept override
{
  return InterlockedIncrement(&m_references);
}
```

That's largely self-explanatory. Don't be tempted to come up with some complex scheme whereby you might conditionally replace the InterlockedIncrement and InterlockedDecrement intrinsic functions with the C++ increment and decrement operators. ATL

attempts to do so at a great expense of complexity. If efficiency is your concern, rather spend your efforts avoiding spurious calls to AddRef and Release. Again, modern C++ comes to the rescue with its support for move semantics and its ability to move the ownership of references without a reference bump. Now, the Release method is only marginally more complex:

```
virtual unsigned long __stdcall Release() noexcept override
{
  unsigned long const remaining = InterlockedDecrement(&m_references);

  if (0 == remaining)
  {
    delete this;
  }

  return remaining;
}
```

The reference count is decremented and the result is assigned to a local variable. This is important as this result should be returned, but if the object were to be destroyed it would then be illegal to refer to the member variable. Assuming there are no outstanding references, the object is simply deleted via a call to the aforementioned virtual destructor. This concludes reference counting, and the concrete Hen class is still as simple as before:

```
class Hen : public Implements<IHen, IHen2>
{
};
```

Now it's time to consider the wonderful world of QueryInterface. Implementing this IUnknown method is a nontrivial exercise. I cover this extensively in my Pluralsight courses and you can read about the many weird and wonderful ways to roll your own implementation in "Essential COM" (Addison-Wesley Professional, 1998) by Don Box. Be warned that while this is an excellent text on COM, it's based on C++98 and doesn't represent modern C++ in any way. For the sake of space and time, I'll assume you have some familiarity with the implementation of QueryInterface and focus instead on how to implement it with modern C++. Here's the virtual method itself:

```
virtual HRESULT __stdcall QueryInterface(
  GUID const & id, void ** object) noexcept override
{
}
```

Given a GUID identifying a particular interface, QueryInterface should determine whether the object implements the desired interface. If it does, it must increment the reference count for the object and then return the desired interface pointer via the out parameter. If not, it must return a null pointer. Therefore, I'll start with a rough outline:

```
*object = // Find interface somehow

if (nullptr == *object)
{
  return E_NOINTERFACE;
}

static_cast<::IUnknown *>(*object)->AddRef();
return S_OK;
```

So QueryInterface first attempts to find the desired interface somehow. If the interface isn't supported, the requisite E_NOINTERFACE error code is returned. Notice how I've already taken care of the requirement to clear the resulting interface pointer on failure. You should think of QueryInterface very much as a binary operation. It either succeeds in finding the desired interface or not. Don't be tempted to get creative here and only conditionally respond favorably. Although there are some limited options allowed by the COM specification, most consumers will simply assume that the interface isn't supported, regardless of what failure code you might return. Any mistakes in your implementation will undoubtedly cause you no end of debugging misery. QueryInterface is too fundamental to mess around with. Finally, AddRef is called through the resulting interface pointer again to support some rare but permissible class composition scenarios. Those aren't explicitly supported by the Implements class template, but I'd rather set a good example here. It's important to keep in mind that the reference-counting operations are interface-specific rather than object-specific. You can't simply call AddRef or Release on any interface belonging to an object. You must honor the COM rules governing object identity, otherwise you risk introducing illegal code that will break in mysterious ways.

So how do I discover whether the requested GUID represents an interface that the class intends to implement? That's where I can return to the type information the Implements class template collects via its template parameter pack. Remember, my goal is to allow the compiler to implement this for me. I want the resulting code to be as efficient as if I had written it by hand, or better. I'll therefore do this query with a set of variadic function templates, function templates that themselves include template parameter packs. I'll start with a BaseQueryInterface function template to kick things off:

```
virtual HRESULT __stdcall QueryInterface(
  GUID const & id, void ** object) noexcept override
{
  *object = BaseQueryInterface<Interfaces ...>(id);
```

BaseQueryInterface is essentially a modern C++ projection of the IUnknown QueryInterface. Instead of returning an HRESULT, it returns the interface pointer directly. Failure is obviously indicated with a null pointer. It accepts a single function argument, the GUID identifying the interface to find. More important, I expand the class template's parameter pack in its entirety so that the BaseQueryInterface function can begin the process of enumerating the interfaces. You might at first think that because BaseQueryInterface is a member of the Implements class template it can simply access this list of interfaces directly, but I need to allow this function to peel off the first interface in the list, as follows:

```
template <typename First, typename ... Rest>
void * BaseQueryInterface(GUID const & id) noexcept
{
}
```

In this way, BaseQueryInterface can identify the first interface and leave the rest for a subsequent search. You see, COM has a number of specific rules to support object identity that QueryInterface must implement or at least honor. In particular, requests for IUnknown must always return the exact same pointer so a client can determine whether two interface pointers refer to the same object. Thus, the BaseQueryInterface function is a great place to implement some of these axioms. Therefore, I might begin by comparing the requested GUID with the first template argument that represents the first interface the class intends to implement. If that's not a match, I'll check whether IUnknown is being requested:

```
if (id == __uuidof(First) || id == __uuidof(::IUnknown))
{
  return static_cast<First *>(this);
}
```

Assuming one of these is a match, I simply return the unambiguous interface pointer for the first interface. The static_cast ensures the compiler won't trouble itself with the ambiguity of multiple interfaces based on IUnknown. The cast merely adjusts the pointer to find the correct location in the class vtable, and because all interface vtables start with IUnknown's three methods, this is perfectly valid.

While I'm here I might as well add optional support for IInspectable queries, as well. IInspectable is a rather odd beast. In some sense it's the IUnknown of the Windows Runtime because every Windows Runtime interface projected into languages like C# and JavaScript must derive directly from IInspectable rather than merely IUnknown alone. This is an unfortunate reality to accommodate the way the Common Language Runtime implements objects and interfaces—which is in contrast to the way C++ works and how COM has traditionally been defined. It also has some rather unfortunate performance ramifications when it comes to object composition, but that's a large topic I'll cover in a follow-up article. As far as QueryInterface is concerned, I just need to ensure that IInspectable can be queried should this be the implementation of a Windows Runtime class rather than simply a classic COM class. Although the explicit COM rules about IUnknown don't apply to IInspectable, I can simply treat the latter in much the same way here. But this presents two challenges. First, I need to discover whether any of the implemented interfaces derive from IInspectable. And, second, I need the type of one such interface so I can return a properly adjusted interface pointer without ambiguity. If I could assume that the first interface in the list would always be based on IInspectable, I might simply update BaseQueryInterface as follows:

```
if (id == __uuidof(First) ||
  id == __uuidof(::IUnknown) ||
  (std::is_base_of<::IInspectable, First>::value &&
  id == __uuidof(::IInspectable)))
{
  return static_cast<First *>(this);
}
```

Notice that I'm using the C++11 is_base_of type trait to determine whether the first template argument is an IInspectable-derived interface. This ensures that the subsequent comparison is excluded by the compiler should you be implementing a classic COM class with no support for the Windows Runtime. In this way I can seamlessly support both Windows Runtime and classic COM classes without any additional syntactic complexity for component developers and without any unnecessary runtime overhead. But this leaves the potential for a very subtle bug should you happen to list a non-IInspectable interface first. What I need to do is replace is_base_of with something that can scan the entire list of interfaces:

```
template <typename First, typename ... Rest>
constexpr bool IsInspectable() noexcept
{
  return std::is_base_of<::IInspectable, First>::value ||
    IsInspectable<Rest ...>();
}
```

IsInspectable still relies on the is_base_of type trait, but now applies it to each interface until a match is found. If no interfaces based on IInspectable are found, the terminating function is hit:

```
template <int = 0>
constexpr bool IsInspectable() noexcept
{
  return false;
}
```

I'll get back to the curious nameless default argument in a moment. Assuming IsInspectable returns true, I need to find the first IInspectable-based interface:

```
template <int = 0>
void * FindInspectable() noexcept
{
  return nullptr;
}

template <typename First, typename ... Rest>
void * FindInspectable() noexcept
{
  // Find somehow
}
```

I can again rely on the is_base_of type trait, but this time return an actual interface pointer should a match be found:

```
#pragma warning(push)
#pragma warning(disable:4127) // conditional expression is constant

if (std::is_base_of<::IInspectable, First>::value)
{
  return static_cast<First *>(this);
}

#pragma warning(pop)

return FindInspectable<Rest ...>();
```

BaseQueryInterface can then simply use IsInspectable along with FindInspectable to support queries for IInspectable:

```
if (IsInspectable<Interfaces ...>() && id == __uuidof(::IInspectable))
{
  return FindInspectable<Interfaces ...>();
}
```

Again, given the concrete Hen class:

```
class Hen : public Implements<IHen, IHen2>
{
};
```

The Implements class template will ensure the compiler generates the most efficient code whether IHen or IHen2 derives from IInspectable or simply from IUnknown (or some other interface). Now I can finally implement the recursive portion of QueryInterface to cover any additional interfaces, such as IHen2 in the previous example. BaseQueryInterface concludes by calling a FindInterface function template:

```
template <typename First, typename ... Rest>
void * BaseQueryInterface(GUID const & id) noexcept
{
  if (id == __uuidof(First) || id == __uuidof(::IUnknown))
  {
    return static_cast<First *>(this);
  }

  if (IsInspectable<Interfaces ...>() && id == __uuidof(::IInspectable))
  {
    return FindInspectable<Interfaces ...>();
  }

  return FindInterface<Rest ...>(id);
}
```

Notice that I'm calling this FindInterface function template in much the same way as I originally called BaseQueryInterface. In this case, I'm passing it the rest of the interfaces. Specifically, I'm expanding the parameter pack such that it can again identify the first interface in the rest of the list. But this presents a problem. Because the template parameter pack isn't expanded as function arguments, I can end up in a vexing situation where the language won't let me express what I really want. But more on that in a moment. This "recursive" FindInterface variadic template is as you might expect:

```
template <typename First, typename ... Rest>
void * FindInterface(GUID const & id) noexcept
{
  if (id == __uuidof(First))
  {
    return static_cast<First *>(this);
  }

  return FindInterface<Rest ...>(id);
}
```

It separates its first template argument from the rest, returning the adjusted interface pointer if there's a match. Otherwise, it calls itself until the interface list is exhausted. While I do loosely refer to this as compile-time recursion, it's important to note that this function template—and the other similar examples in the Implements class template—are not technically recursive, not even at compile time. Each instantiation of the function template calls a different instantiation of the function template. For example, FindInterface<IHen, IHen2> calls FindInterface<IHen2>, which calls FindInterface<>. In order for it to be recursive, FindInterface<IHen, IHen2> would need to call FindInterface<IHen, IHen2>, which it does not.

Nevertheless, keep in mind that this "recursion" happens at compile time and it's as if you'd written all those if statements by hand, one after the other. But now I hit a snag. How does this sequence terminate? When the template argument list is empty, of course. The problem is that C++ already defines what an empty template parameter list means:

```
template <>
void * FindInterface(GUID const &) noexcept
{
  return nullptr;
}
```

This is almost right, but the compiler will tell you that a function template isn't available for this specialization. And, yet, if I don't provide this terminating function, the compiler will fail to compile the final call when the parameter pack is empty. This is not a case for function overloading, because the list of arguments remains the same. Fortunately, the solution is simple enough. I can avoid the terminating function looking like a specialization by providing it with a nameless default argument:

```
template <int = 0>
void * FindInterface(GUID const &) noexcept
{
  return nullptr;
}
```

The compiler is happy, and if an unsupported interface is requested, this terminating function simply returns a null pointer and the virtual QueryInterface method will return the E_NOINTERFACE error code. And that takes care of IUnknown. If all you care about is classic COM, you can safely stop there as that's all you'll ever need. It's worth reiterating at this point that the compiler will optimize this QueryInterface implementation, with its various "recursive" function calls and constant expressions, such that the code is at least as good as you could write by hand. And the same can be achieved for IInspectable.

For Windows Runtime classes, there's the added complexity of implementing IInspectable. This interface isn't nearly as fundamental as IUnknown, providing a dubious collection of facilities compared to the absolutely essential functions of IUnknown. Still, I'll leave a discussion about that for a future article and focus here on an efficient and modern C++ implementation to support any Windows Runtime class. First, I'll get the GetRuntimeClassName and GetTrustLevel virtual functions out of the way. Both methods are relatively trivial to implement and are also rarely used so their implementations can largely be glossed over. The GetRuntimeClassName method should return a Windows Runtime string with the full name of the runtime class that the object represents. I'll leave that up to the class itself to implement should it decide to do so. The Implements class template can simply return E_NOTIMPL to indicate that this method isn't implemented:

```
HRESULT __stdcall GetRuntimeClassName(HSTRING * name) noexcept
{
  *name = nullptr;
  return E_NOTIMPL;
}
```

Likewise, the GetTrustLevel method simply returns an enumerated constant:

```
HRESULT __stdcall GetTrustLevel(TrustLevel * trustLevel) noexcept
{
  *trustLevel = BaseTrust;
  return S_OK;
}
```

Notice that I don't explicitly mark these IInspectable methods as virtual functions. Avoiding the virtual declaration allows the compiler to strip out these methods, should the COM class not actually implement any IInspectable interfaces. Now I'll turn my attention to the IInspectable GetIids method. This is even more error-prone than QueryInterface. Although its implementation isn't nearly as critical, an efficient compiler-generated implementation is desirable. GetIids returns a dynamically allocated array of GUIDs. Each GUID represents one interface that the object purports to implement. You might at first think this is simply a declaration of what the object supports via QueryInterface, but that's correct only at face value. The GetIids method might decide to withhold some interfaces from publication. Anyway, I'll begin with its basic definition:

```
HRESULT __stdcall GetIids(unsigned long * count, GUID ** array) noexcept
{
  *count = 0;
  *array = nullptr;
```

The first parameter points to a caller-provided variable that the GetIids method must set to the number of interfaces in the resulting array. The second parameter points to an array of GUIDs and is how the implementation conveys the dynamically allocated array back to the caller. Here, I've started by clearing both parameters, just to be safe. I now need to determine how many interfaces the class implements. I would love to say just use the sizeof operator, which can provide the size of a parameter pack, as follows:

```
unsigned const size = sizeof ... (Interfaces);
```

This is mighty handy and the compiler is happy to report the number of template arguments that would be present if this parameter pack is expanded. This is also effectively a constant expression, producing a value known at compile time. The reason this won't do, as I alluded to earlier, is because it's extremely common for implementations of GetIids to withhold some interfaces that they don't wish to share with everyone. Such interfaces are known as cloaked interfaces. Anyone can query for them via QueryInterface, but GetIids won't tell you that they're available. Thus, I need to provide a compile-time replacement for the variadic sizeof operator that excludes cloaked interfaces, and I need to provide some way to declare and identify such cloaked interfaces. I'll start

with the latter. I want to make it as easy as possible for component developers to implement classes so a relatively unobtrusive mechanism is in order. I can simply provide a Cloaked class template to "decorate" any cloaked interfaces:

```
template <typename Interface>
struct Cloaked : Interface {};
```

I can then decide to implement a special "IHenNative" interface on the concrete Hen class that's not known to all consumers:

```
class Hen : public Implements<IHen, IHen2, Cloaked<IHenNative>>
{
};
```

Because the Cloaked class template derives from its template argument, the existing QueryInterface implementation continues to work seamlessly. I've just added a bit of extra type information I can now query for, again at compile time. For that I'll define an IsCloaked type trait so I can easily query any interface to determine whether it has been cloaked:

```
template <typename Interface>
struct IsCloaked : std::false_type {};

template <typename Interface>
struct IsCloaked<Cloaked<Interface>> : std::true_type {};
```

I can now count the number of uncloaked interfaces again using a recursive variadic function template:

```
template <typename First, typename ... Rest>
constexpr unsigned CounInterfaces() noexcept
{
  return !IsCloaked<First>::value + CounInterfaces<Rest ...>();
}
```

And, of course, I'll need a terminating function that can simply return zero:

```
template <int = 0>
constexpr unsigned CounInterfaces() noexcept
{
  return 0;
}
```

The ability to do such arithmetic calculations at compile time with modern C++ is stunningly powerful and amazingly simple. I can now continue to flesh out the GetIids implementation by requesting this count:

```
unsigned const localCount = CounInterfaces<Interfaces ...>();
```

The one wrinkle is that the compiler's support for constant expressions is not yet very mature. While this is undoubtedly a constant expression, the compiler doesn't yet honor constexpr member functions. Ideally, I could mark the CountInterfaces function templates as constexpr and the resulting expression would likewise be a constant expression, but the compiler doesn't yet see it that way. On the other hand, I have no doubt that the compiler will have no trouble optimizing this code anyway. Now, if for whatever reason CounInterfaces finds no uncloaked interfaces, GetIids can simply return success because the resulting array will be empty:

```
if (0 == localCount)
{
  return S_OK;
}
```

Again, this is effectively a constant expression and the compiler will generate the code without a conditional one way or another. In other words, if there are no uncloaked interfaces, the remaining code is simply removed from the implementation. Otherwise, the implementation is compelled to allocate a suitably sized array of GUIDs using the traditional COM allocator:

```
GUID * localArray = static_cast<GUID *>(CoTaskMemAlloc(sizeof(GUID) * localCount));
```

Of course, this might fail, in which case I simply return the appropriate HRESULT:

```
if (nullptr == localArray)
{
  return E_OUTOFMEMORY;
}
```

At this point, GetIids has an array ready to be populated with GUIDs. As you might expect, I need to enumerate the interfaces one final time to copy each uncloaked interface's GUID to this array. I'll use a pair of function templates as I've done before:

```
template <int = 0>
void CopyInterfaces(GUID *) noexcept {}

template <typename First, typename ... Rest>
void CopyInterfaces(GUID * ids) noexcept
{
}
```

The variadic template (the second function) can simply use the IsCloaked type trait to determine whether to copy the GUID for the interface identified by its First template argument before incrementing the pointer. In that way, the array is traversed without having to keep track of how many elements it might contain or to where in the array it should be written. I also suppress the warning about this constant expression:

```
#pragma warning(push)
#pragma warning(disable:4127) // Conditional expression is constant

if (!IsCloaked<First>::value)
{
  *ids++ = __uuidof(First);
}

#pragma warning(pop)

CopyInterfaces<Rest ...>(ids);
```

As you can see, the "recursive" call to CopyInterfaces at the end uses the potentially incremented pointer value. And I'm almost done. The GetIids implementation can then conclude by calling CopyInterfaces to populate the array before returning it to the caller:

```
  CopyInterfaces<Interfaces ...>(localArray);

  *count = localCount;
  *array = localArray;
  return S_OK;
}
```

All the while, the concrete Hen class is oblivious to all the work the compiler is doing on its behalf:

```
class Hen : public Implements<IHen, IHen2, Cloaked<IHenNative>>
{
};
```

And that's as it should be with any good library. The Visual C++ 2015 compiler provides incredible support for Standard C++ on the Windows platform. It enables C++ developers to build beautifully elegant and efficient libraries. This supports both the development of Windows Runtime components in Standard C++, as well as their consumption from universal Windows apps written entirely in Standard C++. The Implements class template is just one example from Modern C++ for the Windows Runtime (see moderncpp.com). ■

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 at twitter.com/kennykerr.*

Thanks to the following Microsoft technical expert for reviewing this article: *James McNellis*

Connect(); the Past to the Future

I arrived at Microsoft in 2000, hired as a software test engineer supporting a product that eventually became known as "the browser that wouldn't die." It wasn't a term of endearment. The company and our industry were different then—mobility was limited to traditional laptops and PDAs, and Bill Gates' vision of a PC on every desk was still the primary driver of corporate and platform strategy at Microsoft. The Web was thriving, but so was the traditional PC business. Microsoft was at the height of its powers.

It wouldn't last. Microsoft endured a period now often referred to as the "lost decade," though few realize that the company actually flourished during that time. Revenue and profit increased at startling rates, amazing new products like Xbox and SharePoint created impressive new revenue streams, and core franchises like Windows and Office continued to perform exceptionally well. The accountants were happy, even if some of our partners were not.

> Developers have been the lifeblood of our company over the years, and you remain so today—thank you for sticking with us through thick and thin.

Then on Jan. 9, 2007, everything changed. Steve Jobs unveiled the first iPhone, and consumers and developers saw something sleek, sexy and eminently connected. Of course, Apple made one mistake—the company didn't provide a programming interface or SDK so developers could take advantage of this amazing new device, but that problem was quickly remedied. The era of mobility had truly begun, and Microsoft was nowhere to be found.

Instead, we offered the world Windows Vista, which was released three weeks after the Apple announcement, and Windows Mobile 6 a few days after that. Despite great new developer capabilities like Windows Presentation Foundation (WPF), Windows Communication Foundation and the Microsoft .NET Framework 3.0, history has shown that it was the wrong strategy. Windows Mobile 6 was a great choice for enterprise users, but it lacked the compelling and forward-looking capabilities of the iPhone. Those were tough days.

Fast forward to 2014, and all of that is ancient history. I'm not sure Microsoft ever truly lost its swagger, but if we did, it's returning. You feel it from our leadership, with Satya Nadella, Scott Guthrie, Terry Myerson, and the rest of the senior team confidently describing Microsoft's place in a "cloud-first, mobile-first" world. You see it in the actions of key leaders, who recognize that Microsoft is uniquely suited to tackle developer productivity in a multi-device world. You find it in the passion of Microsoft employees, who strongly believe that Microsoft is the best place to build amazing new software and experiences for a mobile world. And not incidentally, you notice it in the value of Microsoft stock, which after a decade of stasis has finally shown signs of life.

Considering my own journey at Microsoft, I frequently think about those dark days in the mid-aughts. I was a documentation manager for WPF, and suffered through the multiple "Longhorn" resets. Like many of you, I was deeply disappointed by our inability to meet our promises to modernize Windows and provide a great new developer API that would help us maintain our edge. But despite those failures—maybe because of them—I see a company that is now stronger, has learned from its past, and is imbued with greater resolve and capability.

Windows is still finding it's footing in the new mobile, connected world, but it's awesome to see the progress we've made with Windows Phone 8.1 and especially the new line of Surface Pro 3 devices. And what can I say about Visual Studio, which today is the best platform for building apps not just for Windows, but for Android and iOS, too. I feel fortunate to be working on products and services that enable people to enjoy whatever device they choose, with apps and services powered by Microsoft.

Developers have been the lifeblood of our company over the years, and you remain so today—thank you for sticking with us through thick and thin. As you see in this special edition and in all of the amazing announcements at our recent Connect(); event, our commitment to developers is stronger than it has ever been. Welcome to the "mobile-first, cloud-first" era. We're still here, and we're going to be here for a long, long time. ■

KEITH BOYD *manages the developer documentation team in the Cloud and Enterprise (C&E) division at Microsoft. In that capacity, he oversees editorial strategy for* MSDN Magazine. *Prior to arriving in C&E in 2013, he held the same role in Windows. When he's not at work, he's enjoying time with his family, including his wife, Leslie, and three sons, Aiden, Riley and Drew.*

Desktop. Web. Mobile. Your next great app starts here.

From interactive Desktop applications, to immersive Web and Mobile solutions, development tools built to meet your needs today and ensure your continued success tomorrow.

Download your free 30-day trial today and Experience the DevExpress Difference.

www.devexpress.com/try

DevExpress®

WIN ASP WPF SL VCL XAF CR

All New Releases!

.NET Controls for the Professional Developer

ComponentOne delivers Data Visualization, UI, Spreadsheet and Reporting Controls for Microsoft Visual Studio Development

Reporting	Data Visualization	Spreadsheet

ActiveReports

Easily create a variety of form-based, analytical and transaction reports. Visualize data with graphical components, design reports in the IDE & extend the reach of your reports with versatile viewers.

Studio Enterprise

Hundreds of data and UI controls for all .NET platforms including grids, charts, reports and schedulers. Make building next-gen UIs in today's apps easier with an array of samples with source code.

Spread Studio

A Microsoft Excel-like spreadsheet control with the functionally of an advanced data grid, for adding power to any .NET application. Includes a formula engine and flexible tabular layout.

Download your free trials at
ComponentOne.com

Visual Studio Partner

ComponentOne a division of GrapeCity