Create mobile apps with HTML5, JavaScript and Visual Studio

DevExtreme Mobile is a single page application (SPA) framework for your next Windows Phone, iOS and Android application, ready for online publication or packaged as a store-ready native app using Apache Cordova (PhoneGap). With DevExtreme, you can target today's most popular mobile devices with a single codebase and create interactive solutions that will amaze.

Get started today...

- Leverage your existing Visual Studio expertise.
- Build a real app, not just a web page.
- Deliver a native UI and experience on all supported devices.
- Use over 30 built-in touch optimized widgets.

Learn more and download your free trial
devexpress.com/mobile

DevExpress™

THE MICROSOFT JOURNAL FOR DEVELOPERS

NOVEMBER 2013 VOL 28 NO 11

msdn magazine

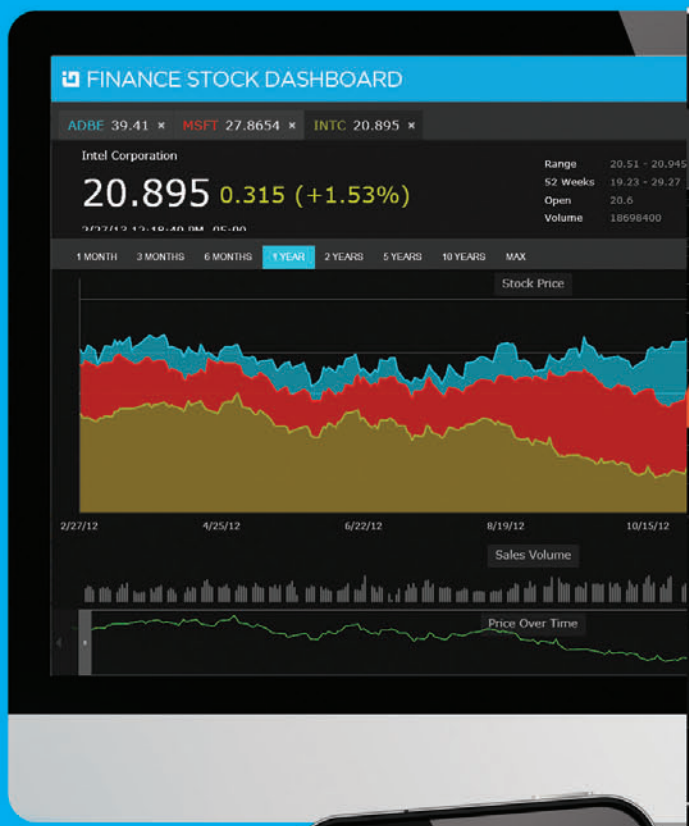
Build Office 365 Apps with Visual Studio 2013.....26

COLUMNS

Microsoft

Desktop

Deliver high performance, scalable and stylable touch-enabled enterprise applications in the platform of your choice.



Native Mobile

Develop rich, device-specific user experience for iOS, Android, and Windows Phone, as well as mobile cross-platform apps with Mono-Touch.

Download Your Free Trial

infragistics.com/enterprise-READY

Dr.Dobb's JOLT 2013 BEST PROGRAMMER LIBRARIES

Cross-Device

Build standards-based, touch-enabled HTML5 & jQuery experiences for desktop, tablet, and mobile delivery, including multi-device targeting with frameworks such as PhoneGap and MVC.

UX Prototyping

Explore design ideas through rapid, user-centered wireframing, prototyping, and evaluation before investing a single line of code.

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

Infragistics Sales US 800 231 8588 • Europe +44 (0) 800 298 9055 • India +91 80 4151 8042 • APAC (+61) 3 9982 4545

msdn magazine

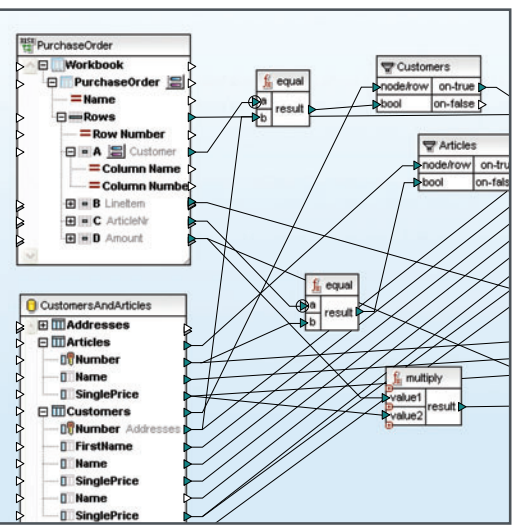
NOVEMBER 2013 VOLUME 28 NUMBER 11

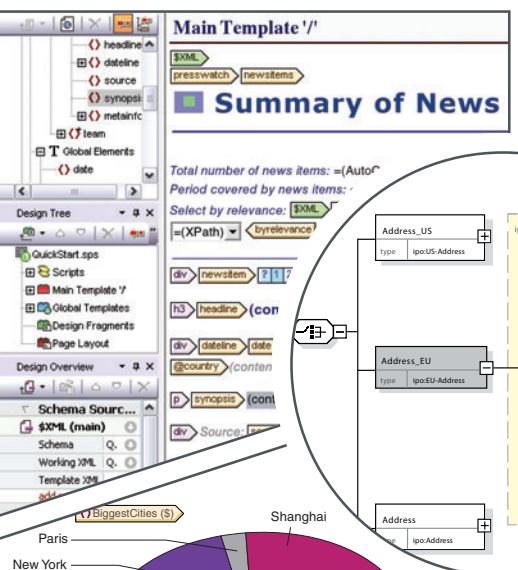
MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. $35.00, International $60.00.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

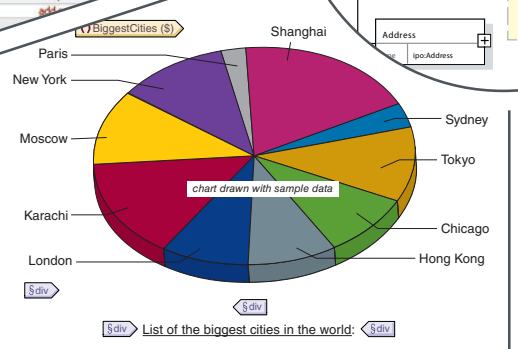
Bring your XML development projects to light with the complete set of tools from Altova®

ALTOVA® missionkit®

Experience how Altova MissionKit®, a software development suite of industrial-strength XML, SQL, and data integration tools, can simplify even the most advanced XML development projects.







Altova MissionKit includes multiple, tightly-integrated XML tools:

XMLSpy® – industry-leading XML editor

- Support for all XML-based technologies
- Industry's strongest validation engine with Smart Fix
- Graphical editing views, powerful debuggers, code generation, & more

MapForce® – any-to-any data mapping & integration tool

- Drag-and-drop data conversion
- Mapping of XML, DBs, EDI, Excel®, XBRL, flat files & Web services
- Automation via MapForce Server

StyleVision® – visual XSLT stylesheet & report designer

- Graphical XSLT stylesheet & report design for XML, XBRL, & SQL databases
- Output to HTML, PDF, Word & more
- Automation via StyleVision Server

Download a 30 day free trial!

Try before you buy with a free, fully functional trial from **www.altova.com**

MICHAEL DESMOND

Standing on Principle

It's one thing to write code. It's quite another to write code that won't become difficult and expensive to manage a few years down the road.

Jason Bock knows the difference. As a practice lead for app dev at software development outfit Magenic, and coauthor of the book "Metaprogramming in .NET" (Manning Publications, 2013), Bock has seen more than his share of promising software projects gone bad due to poor practices. He says it doesn't have to be that way.

At the upcoming Visual Studio Live! conference (vslive.com), Nov. 18-22, in Orlando, Fla., Bock will present the session, "Modern .NET Development Practices and Principles," which aims to get developers up to speed on key concepts such as loose coupling, unit testing and SOLID principles. SOLID stands for Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion. It describes five core principles of object-oriented programming designed to maximize the maintainability and extensibility of code.

"A fair amount of systems are produced rather quickly, and the cost in the long run is a code base that can be very difficult to maintain," says Bock. "Sometimes developers veer away from concepts like dependency injection and mocking because they don't fully understand them."

In fact, about half the developers Bock polls fail to employ the core concepts and principles he describes in his presentation. The thing is, until six years ago, Bock himself was one of those developers.

"It took me a while to really dive in and figure out how these concepts work, but once I did, things clicked pretty quickly, and now it's just a habit to incorporate those ideas into the projects I'm on," Bock says. "If I can motivate attendees to take those first steps, then I've accomplished what I've set out to do."

Adding unit tests to a code base takes work, says Bock, but there are things developers can do to ease the transition. He tells coders to start small and target unit tests at key areas, such as code that's often accessed by other pieces of the system. He also urges developers to add unit tests to new code. This can be tough to do when the system itself isn't geared to having unit tests in place, says Bock, "but at least it's a start."

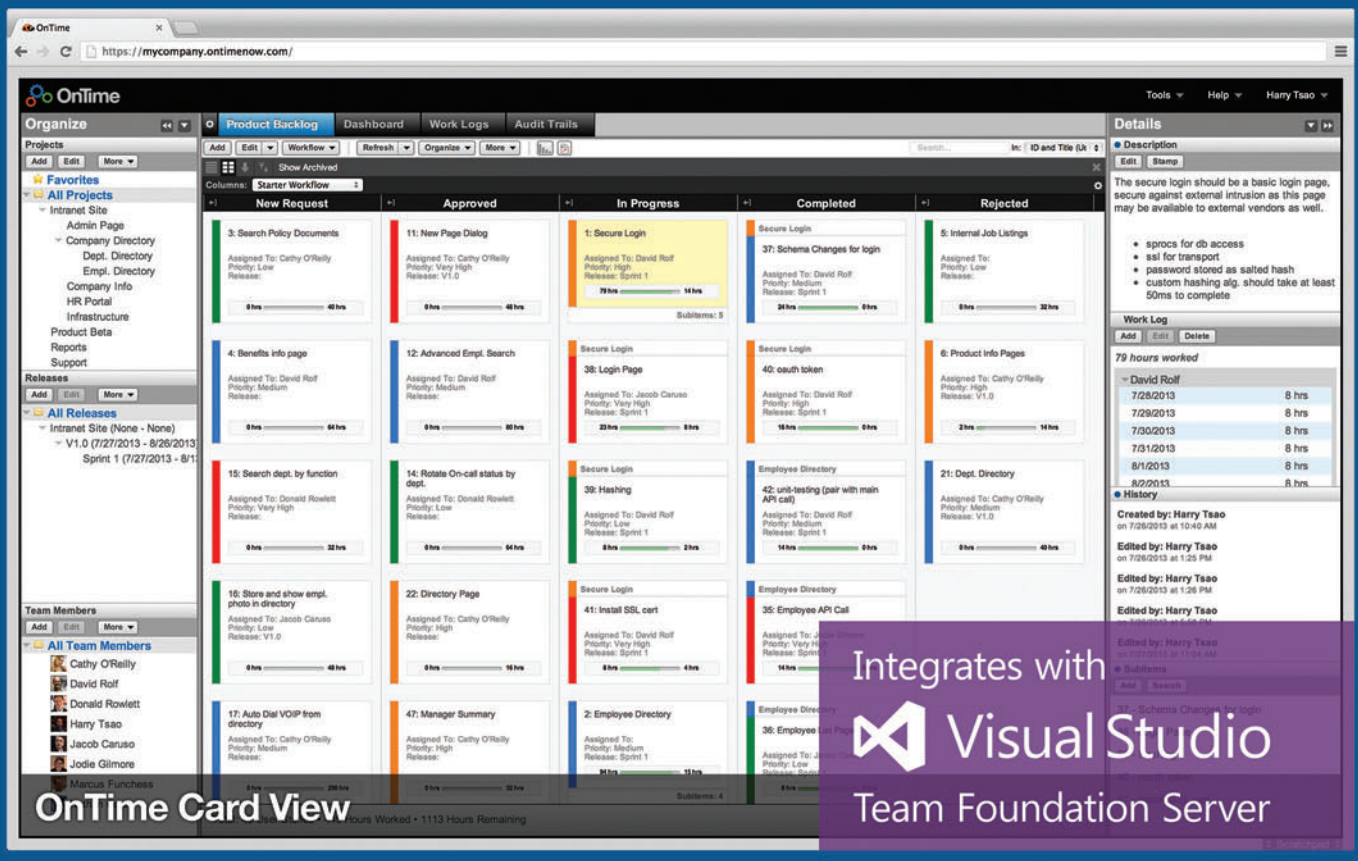
Bock warns there's "no magic answer here." Systems that lack distinct layers and automated tests are likely to be something of a mess. Unit tests on projects such as this will almost certainly be difficult. Ultimately, he says, success is about commitment.

"Sometimes developers veer away from concepts such as dependency injection and mocking because they don't fully understand them."

"You have to do it from the beginning, and you have to be committed to writing those tests," he says. "The last two projects I've been on have had thousands of unit tests. Does that mean they were perfect? No. But I had a lot more assurance that when I changed something in the code I would know if I broke something or not. And I can run all of them with a simple keystroke (CTRL+R, A in Visual Studio 2012). That pays off in the long run. Even if a developer is under pressure to produce code quickly, they also owe it to themselves to have a set of tests in place so, in the future, they don't end up in a place of despair."

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

the #1 selling scrum software



OnTime Scrum agile project management software

Want **three extra months of dev time** a year? Operate like our OnTime Scrum customers who have been able to ship their software 24% faster!

So how are they doing it? Our fast, customizable interface allows users save time and zip from one task to another on one screen—no clicking around. From there **Card View** creates a visually appealing, intuitive system for managing user stories that integrates seamlessly into Visual Studio and TFS source control—perfect for any **Microsoft development environment**.

Is it any wonder that OnTime Scrum is the **#1 selling Scrum software**?

just **$7** per user per month

Get 10% off your OnTime subscription with this link:

OnTimeNow.com/MSDN

Plus: learn about our solutions for bug tracking, help desk & customer management, and project wikis

axosoft

800.653.0024 • www.ontimenow.com • www.axosoft.com • @axosoft

DINO ESPOSITO

Programming CSS: Do More with 'LESS'

In this article, I'll discuss Web development using the LESS Framework for dynamic generation of CSS content.

No doubt that CSS represented a big leap forward with its promise—largely fulfilled—of completely separating content from presentation of Web pages. Even though CSS is (or should be?) in the realm of designers, it heralds the principle of separation of concerns, to which nearly every developer is sensitive. So use of CSS picked up quickly and has become deeply ingrained into Web development to the point that it sometimes struggles to keep up with modern Web sites.

The point isn't that CSS is insufficient to style modern, graphically rich and appealing Web sites, but rather a purely declarative language isn't always appropriate to express complex and interconnected declarations of styles. Thankfully, browsers can still make sense of any CSS as long as it's correctly written—but can we say the same for humans?

A relatively new direction in Web development aims to build an infrastructure around CSS so developers and designers can produce the same CSS in a more sustainable way. The final stylesheet for the browser doesn't change, but the manner in which it's produced should be different, easier to read and more manageable.

This field of Web development started a few years ago and is now reaching maturity, as several available frameworks can help you out with dynamic CSS content generation. I'll provide an executive summary of one of these frameworks—the LESS Framework—and show how it can be integrated with ASP.NET MVC solutions.

Why LESS?

One of the biggest issues developers address with LESS is repetition of information. As a software developer, you probably know the "don't repeat yourself" (DRY) principle and apply it every day. The major benefit of DRY is that it reduces the places where the same information is stored, and thus the number of places where it should be updated.

In plain CSS, you simply have no DRY issues. For example, in some other scenarios, if some color is used in multiple classes and you have to change it, you likely have no better way than updating it in every single occurrence. CSS classes let you define the appearance of certain elements and reuse them across pages to style related elements in the same way. While CSS classes certainly reduce repetition, they're sometimes insufficient in other aspects.

One problem with CSS classes is they operate at the level of the semantic HTML element. In building various CSS classes, you often face repetition of small pieces of information such as colors or widths. You can't easily have a class for each of these repeatable small pieces. Even if you manage to have a CSS class for nearly any repeatable style, such as colors and widths, then when it comes to style the semantic element—say, a container—you should concatenate multiple CSS classes together to achieve the desired effect.

If you've ever used a framework such as Bootstrap for designing your Web page, you know what I mean. Here's an example:

```
<a class="btn btn-primary" ... />
```

A relatively new direction in Web development aims to build an infrastructure around CSS so developers and designers can produce the same CSS in a more sustainable way.

The anchor is first set to be a button (class btn) and then a particular flavor of button (class btn-primary). This approach works, but it might require some significant work to plan ahead for the classes you need. It results in overhead in Web projects that often are on the brink of deadlines.

Figure 1 **Comprehensive Code for Displaying Gradients on a Wide Range of Browsers**

```
/* Old browsers fallback */
background-color: #ff0000;
background: url(images/red_gradient.png);
background-repeat: repeat-x;

/* Browser specific syntax */
background: -moz-linear-gradient(
  left,  #fceabb 0%, #fccd4d 50%, #f8b500 51%, #fbdf93 100%);
background: -Webkit-linear-gradient(
  left, #fceabb 0%,#fccd4d 50%,#f8b500 51%,#fbdf93 100%);
background: -o-linear-gradient(
  left, #fceabb 0%,#fccd4d 50%,#f8b500 51%,#fbdf93 100%);
background: -ms-linear-gradient(
  left, #fceabb 0%,#fccd4d 50%,#f8b500 51%,#fbdf93 100%);

/* Standard syntax */
background: linear-gradient(
  to right, #fceabb 0%,#fccd4d 50%,#f8b500 51%,#fbdf93 100%);
```

Figure 2 **Mixins in the LESS Framework**

```
/*  Mixins  */
.shadow(@color) {
  box-shadow: 3px 3px 2px @color;
}
.text-box(@width) {
  .shadow(#555);
  border: solid 1px #000;
  background-color: #dddd00;
  padding: 2px;
  width: @width;
}

/*  CSS classes  */
.text-box-mini {
  .text-box(50px);
}
.text-box-normal {
  .text-box(100px);
}
.text-box-large {
  .text-box(200px);
}
```

A dynamic stylesheet language such as LESS represents a sort of lateral thinking. You don't spend any time trying to make your plain CSS smarter; you simply use different tools—mostly languages—to generate it. LESS, therefore, is a framework that adds programmer-friendly concepts to CSS coding, such as variables, blocks and functions.

Strictly related to dynamic CSS generation is the problem of processing it to plain CSS for the browser to consume. The client can process the LESS code through ad hoc JavaScript code or preprocess it on the server so the browser just receives final CSS.

A good way to understand the role of LESS variables is by looking into CSS gradients.

Setting up LESS in ASP.NET MVC

I'll demonstrate what it takes to use LESS from within an ASP.NET MVC application. To start out, I'll focus on client-side processing of the LESS code. In the HEAD section of the layout file, add the following:

```
<link rel="stylesheet/less"
  type="text/css"
 href="@Url.Content("~/content/less/mysite.less")" />
<script type="text/javascript"
 src="@Url.Content("~/content/scripts/less-1.3.3.min.js")"></script>
```

This assumes you've created a Content/Less folder in the project to contain all your LESS files. You'll need a JavaScript file to do the actual LESS-to-CSS processing within the browser. You can get the script file from lesscss.org. I'll review a few scenarios where LESS proves useful.

LESS in Action: Variables

A good way to understand the role of LESS variables is by looking into CSS gradients. For years, designers used small GIF files to paint the background of HTML containers with gradients. More recently, browsers added CSS support for gradients. These are also part of the official CSS3 standard through the linear-gradient

msdn magazine

STATEMENT OF OWNERSHIP, MANAGEMENT AND CIRCULATION

1. Publication Title: MSDN Magazine
2. Publication Number: 1528-4859
3. Filing Date: 9/30/13
4. Frequency of Issue: Monthly
5. Number of Issues Published Annually: 12
6. Annual Subscription Price: US $35, International $60
7. Complete Mailing Address of Known Office of Publication: 9201 Oakdale Ave., Ste. 101, Chatsworth, CA 91311
8. Complete Mailing Address of the Headquarters of General Business Offices of the Publisher: Same as above.
9. Full Name and Complete Mailing Address of Publisher, Editor, and Managing Editor:
Dan LaBianca, Vice President/Group Publisher, 14901 Quorum Dr., Ste. 425, Dallas, TX 75254
Michael Desmond, Editor-in-Chief, 600 Worcester Rd., Ste. 204, Framingham, MA 01702
Wendy Hernandez, Group Managing Editor, 4 Venture, Ste. 150, Irvine, CA 92618
10. Owner(s): 1105 Media, Inc. dba: 101 Communications LLC, 9201 Oakdale Ave, Ste. 101, Chatsworth, CA 91311. Listing of shareholders in 1105 Media, Inc.
11. Known Bondholders, Mortgagees, and Other Security Holders Owning or Holding 1 Percent or more of the Total Amount of Bonds, Mortgages or Other Securities:
Nautic Partners V, L.P., 50 Kennedy Plaza, 12th Flr., Providence, RI 02903
Kennedy Plaza Partners III, LLC, 50 Kennedy Plaza, 12th Flr., Providence, RI 02903
Alta Communications 1X, L.P., 1X-B, L.P., Assoc., LLC, 28 State St., Ste. 1801, Boston, MA 02109
12. The tax status has not changed during the preceding 12 months.
13. Publication Title: MSDN Magazine
14. Issue date for Circulation Data Below: September 2013
15. Extent & Nature of Circulation:

	Average No. Copies Each Month During Preceding 12 Months	No. Copies of Single Issue Published Nearest to Filing Date
a. Total Number of Copies (Net Press Run)	84,335	82,491
b. Legitimate Paid/and or Requested Distribution		
1. Outside County Paid/Requested Mail Subscriptions Stated on PS Form 3541	59,823	63,884
2. In-County Paid/Requested Mail Subscriptions Stated on PS Form 3541	0	0
3. Sales Through Dealers and Carriers, Street Vendors, Counter Sales, and Other Paid or Requested Distribution Outside USPS®	5,030	4,980
4. Requested Copies Distributed by Other Mail Classes Through the USPS	0	0
c. Total Paid and/or Requested Circulation	64,853	68,864
d. Nonrequested Distribution		
1. Outside County Nonrequested Copies Stated on PS Form 3541	16,307	12,486
2. In-County Nonrequested Copies Distribution Stated on PS Form 3541	0	0
3. Nonrequested Copies Distribution Through the USPS by Other Classes of Mail	0	0
4. Nonrequested Copies Distributed Outside the Mail	2,990	986
e. Total Nonrequested Distribution	19,297	13,472
f. Total Distribution	84,150	82,336
g. Copies not Distributed	185	155
h. Total	84,335	82,491
i. Percent paid and/or Requested Circulation	77.07%	83.64%

16. ☒ **Total Circulation includes elections copies. Report circulation on PS Form 3526X worksheet.**
17. Publication of Statement of Ownership for a Requester Publication is required and will be printed in the November 2013 issue of this publication.
18. I certify that all information furnished on this form is true and complete:
David Seymour, Director, Print and Online Production

If you are using PS Form 3526R and claiming electronic copies complete below:

a. Requested and Paid Electronic Copies	16,700	16,146
b. Total Requested and Paid Print Copies (Line 15C) + Paid Electronic Copies	81,553	85,010
c. Total Requested Copy Distribution (Line 15F) + Paid Electronic Copies	100,850	98,482
d. Percent Paid and/or Requested Circulation (Both Print & Electronic Copies)	80.87%	86.32%

☒ **I Certify that 50% of all my distributed copies (Electronic & Print) are legitimate requests.**

syntax and its variations. Unfortunately, if you want to make sure the gradient is picked up by the widest possible range of browsers, you have to resort to something like the code in **Figure 1**.

The code in **Figure 1** is nearly unreadable. Worse yet, it must be repeated anywhere you want that gradient. Also, if you want to change the gradient color slightly (or simply the saturation or fading), the only option is editing all occurrences manually. Without beating around the bush, this can be extremely tough. However, it's the only way it can work in plain CSS.

To find a better solution, you need to look outside CSS and enter the territory of LESS. In LESS, you define the CSS for the gradient once and refer to it by name wherever appropriate. Here's an example:

```
.background-gradient-orange { background: #fceabb; ... }
.container { .background-gradient-orange; }
```

The class named background-gradient-orange is embedded by name in the class container and any other class where appropriate. The definition of the gradient, though, is kept in one place.

There's nothing revolutionary in this if you look at it from a developer's perspective. However, it uses a feature—variables—that just doesn't exist in CSS. The preceding syntax, in fact, won't work if you save and reference the file as a plain stylesheet. Some code is required to turn the extended syntax into plain CSS. The LESS JavaScript parser does just this and expands variables to their actual CSS content.

Variables apply also to scalar values such as colors or sizes. Consider the following LESS code:

```
@black: #111;

#main {  color: @black; }
.header { background-color: @black; }
```

The parser expands the @black variable to the assigned value and replaces it all over the file. The net effect is that you change the actual color in one place and changes ripple through the file automatically.

LESS in Action: Imports

You can split your LESS code across multiple files, reference files and contained classes where necessary. Suppose, for example, you create a gradients.less file with the following content:

```
.background-gradient-orange { background: #fceabb; ... }
```

In another LESS file, say main.less, you can reference any gradients by importing the file:

```
@import "gradients";
.container { .background-gradient-orange; }
```

If gradients.less (the extension isn't strictly required) lives in a different folder, you should indicate path information in the call to import.

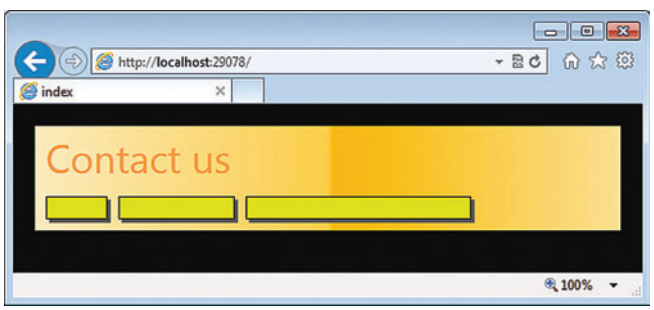


Figure 3 **LESS Mixins in Action**

LESS Mixins

I called the LESS artifact for gradients a variable. To be picky, that's not entirely correct. In LESS, a variable encompasses a single value. A container for a CSS class is known as a mixin. This is similar to a function but doesn't contain any custom logic. Just like a function, a LESS mixin can take and process parameters. Consider the code in **Figure 2** that demonstrates a mixin.

In **Figure 2**, a mixin named shadow defines the styles for a box shadow and exposes the color as an external parameter. Similarly, the text-box mixin defines the basic appearance of an input field. It imports the shadow definition and keeps the width parametric. In this way, defining three classes for input fields of different sizes (mini, normal and large) is a breeze. More important, it requires a fraction of the editing and takes only minimal effort to update (see **Figure 3**).

> A container for a CSS class is known as a mixin, which is similar to a function but doesn't contain any custom logic.

Mixins can accept multiple parameters and also a variable number of parameters. In addition, individual parameters support default values:

```
.mixin(@color: #ff0) { ... }
```

LESS isn't an expression of a rich programming language, and by design it lacks commands to indicate conditions or loops. However, the behavior of a mixin can still be different depending on the passed value. Suppose you want to give a larger button a thicker border and font. You define a parametric mixin named button and use the keyword "when" to bind settings to a condition. The condition must be based on a single parameter:

```
.button (@size) when (@size < 100px) {
 padding: 3px;
 font-size: 0.7em;
 width: @size *2;
}
.button (@size) when (@size >= 100px) {
  padding: 10px;
  font-size: 1.0em;
  font-weight: 700;
  background-color: red;
  width: @size *3;
}
```

You apply different settings, but you also can use basic operations to multiply the size by a factor. Next, use mixins in actual CSS classes:

```
.push-button-large {
  .button(150px);
}
.push-button-small {
  .button(80px);
}
```

The results of running this code are shown in **Figure 4**.

LESS comes with a long list of predefined functions for manipulating colors. You have functions to darken, lighten and saturate colors by a percentage and fade colors in and out by a percentage, as shown here:

```
.push-button {
  background-color: fade(red, 30%);
}
```

For full documentation about the functions supported by LESS, check out lesscss.org.

Nesting Classes

Personally, I find rather annoying the need to repeat CSS blocks to indicate sibling styles. A typical example is:

```
#container h1 { ... }
#container p { ... }
#container p a { ... }
#container img { ... }
```

In well-written plain CSS, you can actually avoid much of the repetition, but the manner in which styles are laid out—using a flat listing—isn't optimal. In this case, a bit of a hierarchy is preferable. In LESS, you can nest style rules like this:

```
.container {
  h1 {
    font-size: 0.8em;
   color: fade(#333, 30%);
   a {
     color: #345;
     &:hover {color: red;}
    }
  }
}
```

Once processed, the preceding LESS code produces the following styles:

```
.container h1
.container h1 a
.container h1a:hover
```

Server-Side Processing

You can download the LESS code as is and process it on the client through JavaScript code. You can also preprocess on the server and download it to the client as plain CSS. In the former case, everything works as if you were using plain CSS files: Server-side changes are applied to the client with the next page refresh.

Server-side preprocessing might be a better option if you have performance concerns and are dealing with large, complex CSS files. Server-side preprocessing takes place every time you modify the CSS on the server. You can manually take care of the extra step at the end of the build process. You preprocess LESS code to CSS using the LESS compiler from the command line. The compiler is part of the dotless NuGet package you install for server-side work.

> Regardless of the tool, a CSS preprocessor is definitely something you want to consider for extensive Web programming.

In ASP.NET MVC 4, however, you can integrate the LESS Framework with the bundle mechanism covered in my October 2013 column, "Programming CSS: Bundling and Minification" (msdn.microsoft.com/magazine/dn451436). This ensures the LESS-to-CSS transformation is performed whenever you make a request for a LESS file. It also ensures caching is managed properly via the If-Modified-Since header. Finally, you can mix together parsing and minifying. To integrate LESS in ASP.NET MVC, first download and install the dotless NuGet package. Second, add the following code to the BundleConfig class:

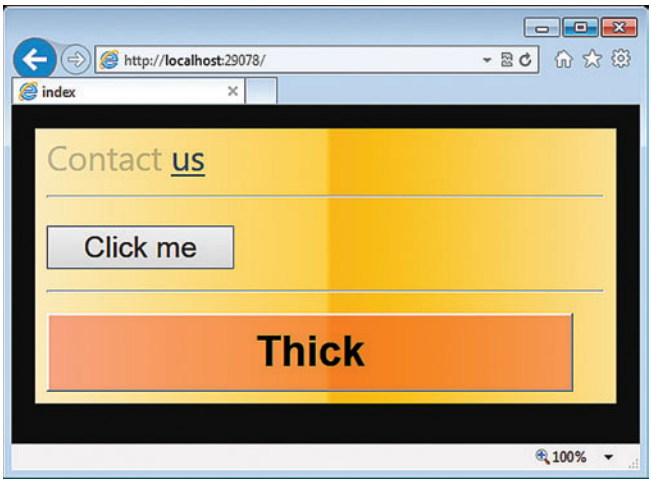


Figure 4 **Effects of Using LESS Mixins in CSS Classes**

```
var lessBundle =
  new Bundle("~/myless").IncludeDirectory("~/content/less", "*.less");
lessBundle.Transforms.Add(new LessTransform());
lessBundle.Transforms.Add(new CssMinify());
bundles.Add(lessBundle);
```

The bundle will package all .less files found in the specified folder. The LessTransform class is responsible for the LESS-to-CSS transformation. The class uses the dotless API to parse LESS scripts. The code for LessTransform is fairly simple:

```
public class LessTransform : IBundleTransform
{
  public void Process(BundleContext context, BundleResponse response)
  {
    response.Content = dotless.Core.Less.Parse(response.Content);
    response.ContentType = "text/css";
  }
}
```

More Intelligent Tools

LESS isn't the only CSS preprocessor out there. Syntactically Awesome Stylesheets (Sass) is another popular one (sass-lang.com), for just one example. The bottom line is that regardless of the tool, a CSS preprocessor is definitely something you want to consider for extensive Web programming. Whether you're a graphic designer or a developer, more intelligent tools to manage and organize CSS code are almost a necessity. They're even better when they're also integrated into the Web platform. Finally, note that Visual Studio 2012 and Visual Studio 2013 offer excellent support for LESS (and related technologies) through the Web Essentials extension, which you can download at vswebessentials.com. Also, the LESS editor is available in Visual Studio 2012 Update 2 and in the upcoming Visual Studio 2013. ■

Dino Esposito *is the author of "Architecting Mobile Solutions for the Enterprise" (Microsoft Press, 2012) and the upcoming "Programming ASP.NET MVC 5" (Microsoft Press). A technical evangelist for the .NET and Android platforms at JetBrains and frequent speaker at industry events worldwide, Esposito shares his vision of software at software2cents.wordpress.com and on Twitter at twitter.com/despos.*

Thanks to the following technical expert for reviewing this article: *Mads Kristensen (Microsoft)*

KENNY KERR

Exploring Fonts with DirectWrite and Modern C++

DirectWrite is an incredibly powerful text layout API. It powers practically all of the leading Windows applications and technologies, from the Windows Runtime (WinRT) implementation of XAML and Office 2013, to Internet Explorer 11 and more. It's not a rendering engine in itself, but has a close relationship with Direct2D, its sibling in the DirectX family. Direct2D is, of course, the premier hardware-accelerated, immediate-mode graphics API.

You can use DirectWrite with Direct2D to provide hardware-accelerated text rendering. To avoid any confusion, I haven't written too much about DirectWrite in the past. I didn't want you to think Direct2D is just the DirectWrite rendering engine. Direct2D is so much more than that. Still, DirectWrite has a lot to offer, and in this month's column I'll show you some of what's possible with DirectWrite and look at how modern C++ can help simplify the programming model.

The DirectWrite API

I'll use DirectWrite to explore the system font collection. First, I need to get hold of the DirectWrite factory object. This is the starting point for any application that wants to use the impressive typography of DirectWrite. DirectWrite, like so much of the Windows API, relies on the essentials of COM. I need to call the DWriteCreateFactory function to create the DirectWrite factory object. This function returns a COM interface pointing to the factory object:

```
ComPtr<IDWriteFactory2> factory;
```

> Use DirectWrite with Direct2D to provide hardware-accelerated text rendering.

The IDWriteFactory2 interface is the latest version of the DirectWrite factory interface introduced with Windows 8.1 and DirectX 11.2 earlier this year. IDWriteFactory2 inherits from IDWriteFactory1, which in turn inherits from IDWriteFactory. The latter is the original DirectWrite factory interface that exposes the bulk of the factory's capabilities.

Given the preceding ComPtr class template, I'll call the DWriteCreateFactory function:

```
HR(DWriteCreateFactory(DWRITE_FACTORY_TYPE_SHARED,
  __uuidof(factory),
  reinterpret_cast<IUnknown **>(factory.GetAddressOf())));
```

DirectWrite includes a Windows service called the Windows Font Cache Service (FontCache). This first parameter indicates whether the resulting factory contributes font usage to this cross-process cache. The two options are DWRITE_FACTORY_TYPE_SHARED and DWRITE_FACTORY_TYPE_ISOLATED. Both SHARED and ISOLATED factories can take advantage of font data that's already cached. Only SHARED factories contribute font data back to the cache. The second parameter indicates specifically which version of the DirectWrite factory interface I'd like it to return in the third and final parameter.

Given the DirectWrite factory object, I can get right to the point and ask it for the system font collection:

```
ComPtr<IDWriteFontCollection> fonts;
HR(factory->GetSystemFontCollection(fonts.GetAddressOf()));
```

The GetSystemFontCollection method has an optional second parameter that tells it whether to check for updates or changes to the

Figure 1 **Enumerating Fonts with the DirectWrite API**

```
ComPtr<IDWriteFactory2> factory;

HR(DWriteCreateFactory(DWRITE_FACTORY_TYPE_SHARED,
  __uuidof(factory),
  reinterpret_cast<IUnknown **>(factory.GetAddressOf())));

ComPtr<IDWriteFontCollection> fonts;
HR(factory->GetSystemFontCollection(fonts.GetAddressOf()));

wchar_t locale[LOCALE_NAME_MAX_LENGTH];

VERIFY(GetUserDefaultLocaleName(locale, _countof(locale)));

unsigned const count = fonts->GetFontFamilyCount();

for (unsigned familyIndex = 0; familyIndex != count; ++familyIndex)
{
  ComPtr<IDWriteFontFamily> family;

  HR(fonts->GetFontFamily(familyIndex, family.GetAddressOf()));

  ComPtr<IDWriteLocalizedStrings> names;
  HR(family->GetFamilyNames(names.GetAddressOf()));

  unsigned nameIndex;
  BOOL exists;

  HR(names->FindLocaleName(locale, &nameIndex, &exists));

  if (exists)
  {
    wchar_t name[64];

    HR(names->GetString(nameIndex, name, countof(name)));

    wprintf(L"%s\n", name);
  }
}
```

set of installed fonts. Fortunately, this parameter defaults to false, so I don't have to think about it unless I want to ensure recent changes are reflected in the resulting collection. Given the font collection, I can get the number of font families in the collection as follows:

```
unsigned const count = fonts->GetFontFamilyCount();
```

Then I use the GetFontFamily method to retrieve individual font family objects using a zero-based index. A font family object represents a set of fonts that share a name—and, of course, a design—but are distinguished by weight, style and stretch:

```
ComPtr<IDWriteFontFamily> family;

HR(fonts->GetFontFamily(index, family.GetAddressOf()));
```

The IDWriteFontFamily interface inherits from the IDWriteFontList interface, so I can enumerate the individual fonts within the font family. It's reasonable and useful to be able to retrieve the name of the font family. Family names are localized, however, so it's not as straightforward as you might expect. I first need to ask the font family for a localized strings object that will contain one family name per supported locale:

```
ComPtr<IDWriteLocalizedStrings> names;
HR(family->GetFamilyNames(names.GetAddressOf()));
```

I can also enumerate the family names, but it's common to simply look up the name for the user's default locale. In fact, the IDWriteLocalizedStrings interface provides the FindLocaleName method to retrieve the index of the localized family name. I'll start by calling the GetUserDefaultLocaleName function to get the user's default locale:

```
wchar_t locale[LOCALE_NAME_MAX_LENGTH];

VERIFY(GetUserDefaultLocaleName(locale, countof(locale)));
```

Then I pass this to the IDWriteLocalizedStrings FindLocaleName method to determine whether the font family has a name localized for the current user:

```
unsigned index;
BOOL exists;

HR(names->FindLocaleName(locale, &index, &exists));
```

If the requested locale doesn't exist in the collection, I might fall back to some default such as "en-us." Assuming that exists, I can use the IDWriteLocalizedStrings GetString method to get a copy:

```
if (exists)
{
  wchar_t name[64];

  HR(names->GetString(index, name, _countof(name)));
}
```

If you're worried about the length, you can first call the GetStringLength method. Just make sure you have a large enough buffer. **Figure 1** provides a complete listing, showing how this all comes together to enumerate the installed fonts.

A Touch of Modern C++

If you're a regular reader, you'll know I've given DirectX, and Direct2D in particular, the modern C++ treatment. The dx.h header (dx.codeplex.com) also covers DirectWrite. You can use it to simplify the code I've presented thus far quite dramatically. Instead of calling DWriteCreateFactory, I can simply call the CreateFactory function from the DirectWrite namespace:

```
auto factory = CreateFactory();
```

Getting hold of the system font collection is equally simple:

```
auto fonts = factory.GetSystemFontCollection();
```

Figure 2 **Enumerating Fonts with dx.h**

```
auto factory = CreateFactory();
auto fonts = factory.GetSystemFontCollection();

wchar_t locale[LOCALE_NAME_MAX_LENGTH];
VERIFY(GetUserDefaultLocaleName(locale, _countof(locale)));

for (auto family : fonts)
{
  auto names = family.GetFamilyNames();
  unsigned index;

  if (names.FindLocaleName(locale, index))
  {
    wchar_t name[64];
    names.GetString(index, name);

    wprintf(L"%s\n", name);
  }
}
```

Enumerating this collection is where dx.h really shines. I don't have to write a traditional for loop. I don't have to call the GetFontFamilyCount and GetFontFamily methods. I can simply write a modern range-based for loop:

```
for (auto family : fonts)
{
  ...
}
```

This is really the same code as before. The compiler (with the help of dx.h) is generating it for me and I get to use a far more natural programming model, making it easier to write code that's both correct and efficient. The preceding GetSystemFontCollection method returns a FontCollection class that includes an iterator that will lazily fetch font family objects. This lets the compiler efficiently implement the range-based loop. **Figure 2** provides a complete listing. Compare it with the code in **Figure 1** to appreciate the clarity and potential for productivity.

DirectWrite does much more than just enumerate fonts.

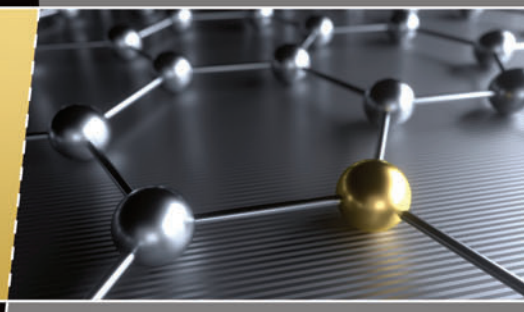
A Font Browser with the Windows Runtime

DirectWrite does much more than just enumerate fonts. I'm going to take what I've shown you thus far, combine it with Direct2D, and create a simple font browser app. In my August 2013 column (msdn.microsoft.com/magazine/dn342867), I showed you how to write the fundamental WinRT app model plumbing in standard C++. In my October 2013 column (msdn.microsoft.com/magazine/dn451437), I showed you how to render inside this CoreWindow-based app with DirectX and specifically Direct2D. Now I'm going to show you how to extend that code to use Direct2D to render text with the help of DirectWrite.

Since those columns were written, Windows 8.1 was released, and it changed a few things about the manner in which DPI scaling is

Empower Your Customers

New! v5.0



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

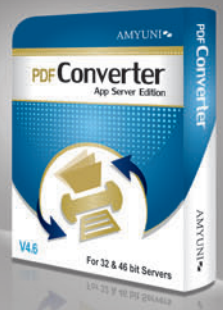
Complete Suite of Accurate PDF Components

- 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 certified PDF Converter printer driver
- Export PDF documents into other formats such as JPeg, PNG, XAML or HTML5

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 and Windows 8
- 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

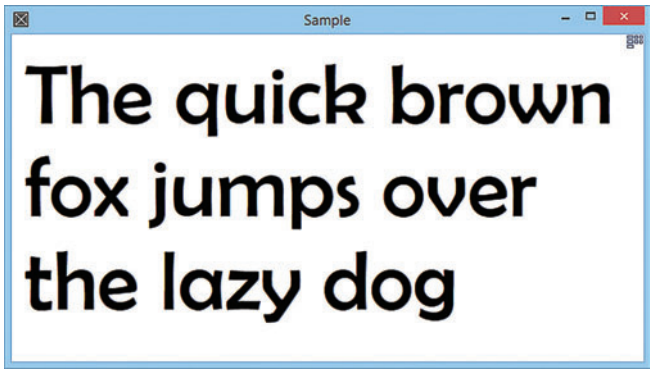


Figure 3 **The Font Browser**

handled in both modern and desktop apps. I'm going to cover DPI in detail in the future, so I'll leave those changes for the time being. I'm simply going to focus on augmenting the SampleWindow class I began in August and extended in October to support text rendering and simple font browsing.

The first thing to do is to add the DirectWrite Factory2 class as a member variable:

```
DirectWrite::Factory2 m_writeFactory;
```

Inside the SampleWindow CreateDeviceIndependentResources method, I can create the DirectWrite factory:

```
m_writeFactory = DirectWrite::CreateFactory();
```

I can also get the system font collection and user's default locale there, in preparation for enumerating the font families:

```
auto fonts = m_writeFactory.GetSystemFontCollection();
wchar_t locale[LOCALE_NAME_MAX_LENGTH];
VERIFY(GetUserDefaultLocaleName(locale, _countof(locale)));
```

I'm going to make the app cycle through the fonts as the user presses the up and down arrow keys. Rather than continually enumerating the collection via the COM interfaces, I'll just copy the font family names to a standard set container:

```
set<wstring> m_fonts;
```

Now I can simply use the same range-based for loop from **Figure 2** inside CreateDeviceIndependentResources to add the names to the set:

```
m_fonts.insert(name);
```

With the set populated, I'll start the app off with an iterator pointing to the beginning of the set. I'll store the iterator as a member variable:

```
set<wstring>::iterator m_font;
```

The SampleWindow CreateDeviceIndependentResources method concludes by initializing the iterator and calling the CreateTextFormat method, which I'll define in a moment:

```
m_font = begin(m_fonts);
CreateTextFormat();
```

Before Direct2D can draw some text for me, I need to create a text format object. To do that, I need both the font family name and the desired font size. I'm going to let the user change the font size with the left and right arrow keys, so I'll start by adding a member variable to keep track of the size:

```
float m_size;
```

The Visual C++ compiler will soon let me initialize nonstatic data members such as this within a class. For now, I need to set it to some reasonable default in the SampleWindow's constructor. Next, I need to define the CreateTextFormat method. It's just a wrapper around the DirectWrite factory method of the same name, but it updates a member variable that Direct2D can use to define the format of the text to be drawn:

```
TextFormat m_textFormat;
```

The CreateTextFormat method then simply retrieves the font family name from the set iterator and combines it with the current font size to create a new text format object:

```
void CreateTextFormat()
{
  m_textFormat = m_writeFactory.CreateTextFormat(m_font->c_str(),m_size);
}
```

I've wrapped it up, so that in addition to calling it initially at the end of CreateDeviceIndependentResources, I can also call it every time the user presses one of the arrow keys to change the font family or size. This leads to the question of how to handle key presses in the WinRT app model. In a desktop application, this involves handling the WM_KEYDOWN message. Fortunately, CoreWindow provides the KeyDown event, which is the modern equivalent of this message. I'll start by defining the IKeyEventHandler interface that my SampleWindow will need to implement:

```
typedef ITypedEventHandler<CoreWindow *, KeyEventArgs *> IKeyEventHandler;
```

I can then simply add this interface to my SampleWindow list of inherited interfaces and update my QueryInterface implementation accordingly. I then just need to provide its Invoke implementation:

```
auto __stdcall Invoke(
  ICoreWindow *,IKeyEventArgs * args) -> HRESULT override
{
  ...

  return S_OK;
}
```

The IKeyEventArgs interface provides much the same information as the LPARAM and WPARAM did for the WM_KEYDOWN message. Its get_VirtualKey method corresponds to the latter's WPARAM, indicating which nonsystem key is pressed:

```
VirtualKey key;
HR(args->get_VirtualKey(&key));
```

Similarly, its get_KeyStatus method corresponds to WM_KEYDOWN's LPARAM. This provides a wealth of information about the state surrounding the key press event:

```
CorePhysicalKeyStatus status;
HR(args->get_KeyStatus(&status));
```

As a convenience to the user, I'll support acceleration when the user presses and holds one of the arrow keys down, to resize the rendered font more rapidly. To support this, I'll need another member variable:

```
unsigned m_accelerate;
```

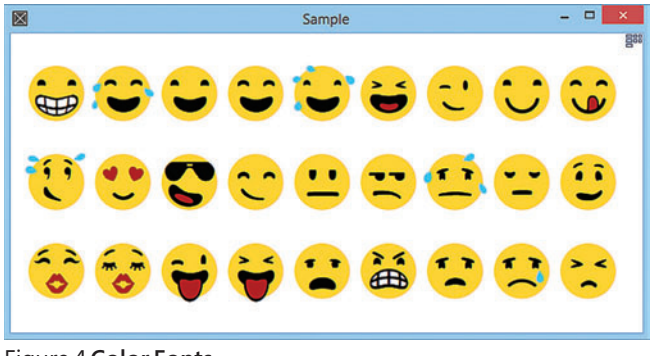


Figure 4 **Color Fonts**

telerik

deliver more than expected

Telerik DevCraft Q3

RELEASE: OCTOBER 2013

- Telerik's full stack of .NET controls and tools for the professional developer
- 350+ UI controls and reporting for all Microsoft platforms
- Productivity tools for faster coding, debugging and profiling

See what's new in Q3 2013 Release & download your 30 day free trial at

www.telerik.com

I can then use the event's key status to determine whether to change the font size by a single increment or an increasing amount:

```
if (!status.WasKeyDown)
{
  m_accelerate = 1;
}
else
{
  m_accelerate += 2;
  m_accelerate = std::min(20U, m_accelerate);
}
```

I've capped it so acceleration doesn't go too far. Now I can simply handle the various key presses individually. First is the left arrow key to reduce the font size:

```
if (VirtualKey_Left == key)
{
  m_size = std::max(1.0f, m_size - m_accelerate);
}
```

I'm careful not to let the font size become invalid. Then there's the right arrow key to increase the font size:

```
else if (VirtualKey_Right == key)
{
  m_size += m_accelerate;
}
```

Next, I'll handle the up arrow key by moving to the previous font family:

```
if (begin(m_fonts) == m_font)
{
  m_font = end(m_fonts);
}

--m_font;
```

Then I carefully loop around to the last font, should the iterator reach the beginning of the sequence. Next, I'll handle the down arrow key by moving to the next font family:

```
else if (VirtualKey_Down == key)
{
  ++m_font;

  if (end(m_fonts) == m_font)
  {
      m_font = begin(m_fonts);
  }
}
```

Here, again, I'm careful to loop around to the beginning this time, should the iterator reach the end of the sequence. Finally, I can simply call my CreateTextFormat method at the end of the event handler to recreate the text format object.

All that remains is to update my SampleWindow Draw method to draw some text with the current text format. This should do the trick:

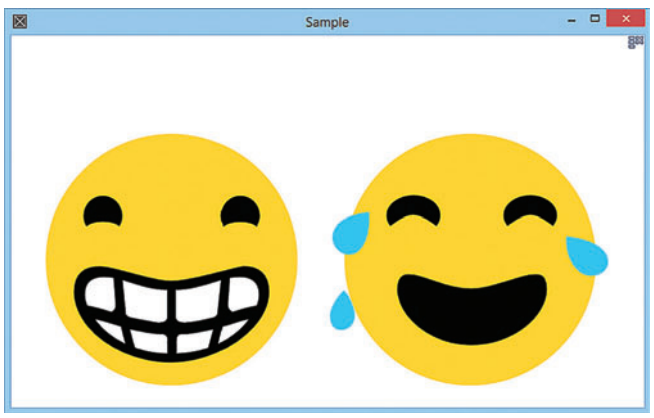


Figure 5 **Scaled Color Fonts**

```
wchar_t const text [] = L"The quick brown fox jumps over the lazy dog";

m_target.DrawText(text, _countof(text) - 1,
  m_textFormat,
  RectF(10.0f, 10.0f, size.Width - 10.0f, size.Height - 10.0f),
  m_brush);
```

The Direct2D render target's DrawText method directly supports DirectWrite. Now DirectWrite can handle text layout, and with amazingly fast rendering. That's all it takes. **Figure 3** gives you an idea of what to expect. I can press the up and down arrow keys to cycle through the font families and press the left and right arrow keys to scale the font size. Direct2D automatically re-renders with the current selection.

> Windows 8.1 introduced a new feature called color fonts, doing away with a number of suboptimal solutions for implementing multicolored fonts.

Did You Say Color Fonts?

Windows 8.1 introduced a new feature called color fonts, doing away with a number of suboptimal solutions for implementing multicolored fonts. Naturally, it all comes down to DirectWrite and Direct2D to make it happen. Fortunately, it's as simple as using the D2D1_DRAW_TEXT_OPTIONS_ENABLE_COLOR_FONT constant when calling the Direct2D DrawText method. I can update my SampleWindow's Draw method to use the corresponding scoped enum value:

```
m_target.DrawText(text, _countof(text) - 1,
  m_textFormat,
  RectF(10.0f, 10.0f, size.Width - 10.0f, size.Height - 10.0f),
  m_brush);
DrawTextOptions::EnableColorFont);
```

Figure 4 shows the font browser again, this time with some Unicode emoticons.

Color fonts really shine when you realize you can scale them automatically without losing quality. I can press the right arrow key in my font browser app and get a closer look at the detail. You can see the result in **Figure 5**.

Giving you color fonts, hardware-accelerated text rendering, and elegant and efficient code, DirectWrite comes to life with the help of Direct2D and modern C++. ■

Kenny Kerr *is a computer programmer based in Canada, 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 technical expert for reviewing this article: *Worachai Chaoweeraprasit (Microsoft)*

Building Blocks for Global Data Quality Success

A strong foundation for enterprise data starts with Melissa Data.

Our powerful, scalable data cleansing and integration tools help you profile, cleanse, consolidate, and enrich your contact data. Gain a better understanding of your customer, vendor and supplier data; improve deliverability; increase cost savings; and enhance your operational efficiencies with Melissa Data.

Advanced Functionalities:

- Verify, correct, and enhance addresses for 240+ countries
- Add lat/long coordinates to addresses all over the world
- Match and consolidate data to create the golden record
- Append missing contact data like phone numbers and email addresses
- Integrate into many technologies with multiplatform capabilities

www.MelissaData.com
or call 1-800-MELISSA (635-4772)

MELISSA DATA®
Your Partner in Data Quality

BRUNO TERKALY AND
RICARDO VILLALOBOS

Migrating Database Workloads to the Cloud

One of the biggest trends in IT today is moving database workloads to the cloud. We frequently get questions about the issues involved when migrating relational database workloads into Windows Azure, so we decided to reach out to various product teams and practitioners in the field to get some concrete answers to these challenges. There's a spectrum of offerings available that must be carefully considered prior to migration, and a host of issues including cost, compatibility, flexibility, maintenance, compliance and scale, to name a few.

This month's column is about moving your SQL Server database to the public cloud. This means we won't discuss the private cloud or other on-premises options. Naturally, beyond just the data, there are a number of issues to consider when moving an entire application to the cloud, such as caching, identity and legacy code.

Many customers are confused about all the options in the cloud, especially when it comes to hosting relational data. After all, the Microsoft relational database story is big and comprehensive. In this month's column, we want to demystify your options when it comes to SQL Server and data storage with the Windows Azure platform.

Most IT decisions are based on cost and value. A good rule of thumb to remember is as you increase the abstraction from the hardware and start to virtualize your technologies, efficiencies are gained, resulting in lower provisioning and maintenance costs. However, there's a trade-off when you virtualize hardware—you might lower costs but at the same time decrease the level of compatibility with existing on-premises databases, because you have less control over the configuration of the database with respect to its underlying hardware. One of our goals here is to describe those trade-offs so that you can make better decisions.

Figure 1 highlights some of the options. There are at least four ways you can leverage SQL Server, as the numbered circles indicate. Notice that both cloud-based and on-premises solutions are depicted. The top half of **Figure 1** describes the public cloud. If you drill further into the public cloud, you'll see it consists of two main pillars: Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). The bottom half of the figure illustrates the options with respect to on-premises SQL Server. You can either host your own private cloud or follow the traditional method and run SQL Server on raw, physical hardware, free of any virtualization technology.

The Windows Azure Portal

We'll leverage three Windows Azure components to deploy our on-premises database to the cloud, illustrating both the IaaS and PaaS models: Virtual Machines, or VMs (IaaS); Windows Azure SQL Database (PaaS); and Storage. They can be accessed through the management portal as shown in **Figure 2**. The first two are fairly obvious, but the Storage section might be a surprise. Windows Azure Storage is needed to hold the BACPAC file, which contains the database schema and the data stored in the database. You can create this file with SQL Server Management Studio (SSMS); it's the logical equivalent of a database backup. SSMS will conveniently place this BACPAC file in Windows Azure Storage, which is an ideal location because it can be imported directly from the cloud versions of SQL Server, as seen in options 3 and 4 in **Figure 1**.

Four Scenarios

Referring back to **Figure 1**, you can see that SQL Server maps to four main use cases. As you move through the four scenarios in order, efficiencies are gained, resulting in lower costs. However, the

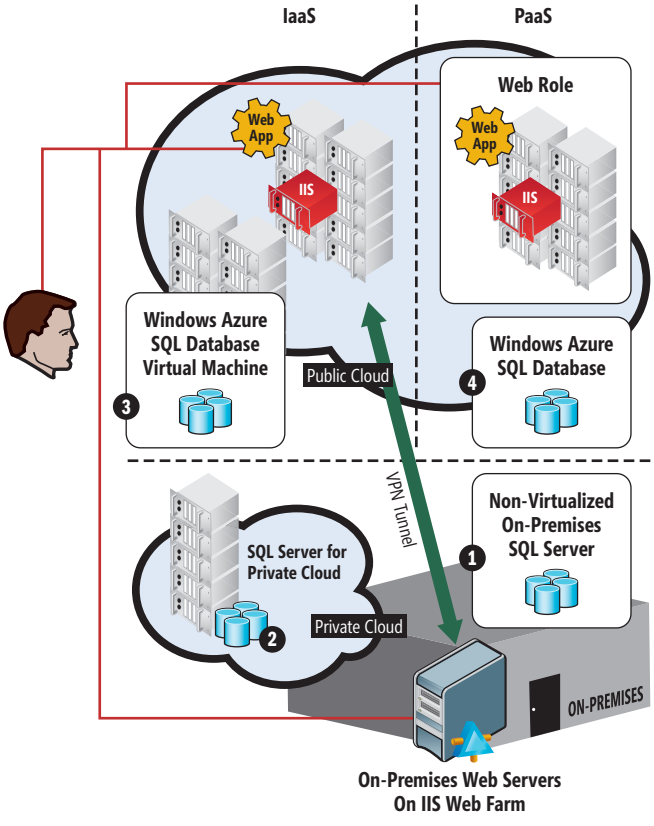


Figure 1 **SQL Server Hosting Options**

Figure 2 **The Windows Azure Management Portal**

options correspondingly impact your control over hardware and configuration, affecting the level of compatibility as you deploy on-premises databases to the cloud.

Old School, Raw Iron Scenario No. 1 is about running SQL Server using traditional approaches, which means it isn't virtualized and can be highly customized by IT personnel and DBAs. Naturally, because this puts IT staff in full control of exactly how everything is configured and how it functions, there's significantly more work in provisioning, maintaining, and scaling the databases. Preparing the cluster for high availability (HA) and performing backup and restore are just two of the many resource-intensive responsibilities that must be managed by a DBA.

On-Premises, Private Cloud Scenario No. 2 depicts SQL Server deployed to a private cloud, which leverages virtualization to optimize the use of hardware resources. This technology allows you to pool SQL Server databases using Windows Server 2012 Hyper-V for efficient use of compute, network and storage. It lets you support an elastic infrastructure for your databases, so you can scale up and down more effectively. From a management perspective, this option provides built-in self-service capabilities, so you can provision databases using Microsoft System Center 2012. However, you're still responsible for procuring the hardware and keeping it up-to-date with software patches.

Windows Azure SQL Server Database VM Windows Azure SQL Server for VMs (IaaS) makes it possible for you to host your cluster in a Microsoft datacenter. The key advantage this public cloud-hosted option offers is a high level of compatibility with existing on-premises SQL Server installations. Because this option supports a full-featured version of SQL Server, you're able to leverage HA, transparent data encryption, auditing, business intelligence (BI) such as analysis services and reporting services, distributed transactions, support for active directory (Active Directory Federation Services, or AD FS 2.0), and more. Moreover, you can build your own VM or leverage a template from the Windows Azure image gallery (described later). Finally, this offering allows you to build your own virtual private networks (VPNs) easily, bridging your cloud resources to your on-premises network and providing a seamless experience that blurs the lines between the massive Microsoft datacenters and your internal servers.

SQL Server for VMs also supports the new Always-On feature, which lets you have up to five complete copies of a database maintained by each of the participating instances. Each SQL Server can host the database on its local storage, or at some other location. This provides a cloud-hosted, enterprise-level alternative to database mirroring.

Another advantage of Windows Azure VMs is that they allow you to leverage the capabilities of Windows Azure Storage, which means that the OS and disk drives are automatically persisted in Windows Azure by default, providing three automatic copies of your data inside the datacenter and allowing the optional use of geo-replication. In short, leveraging Windows Azure SQL Server for VMs makes it possible to migrate your on-premises database applications to Windows Azure, and minimize, if not eliminate, the need to modify the application and the database.

> Some companies have reduced their database costs by more than 90 percent by using Windows Azure SQL Database.

Windows Azure SQL Database To clear up any confusion about terminology, be aware that Windows Azure SQL Database used to be called SQL Azure. Windows Azure SQL Database is both a subset and a superset of the traditional SQL Server, and is hosted in the cloud as a service.

Because Windows Azure SQL Database is indeed a database as a service, there are some real advantages. First, it can be fantastically economical—some companies have reduced their database costs by more than 90 percent by using Windows Azure SQL Database. Second, it provides self-managing capabilities, enabling organizations to provision data services for applications throughout the enterprise without adding to the support burden of the central IT department. Third, the service replicates three copies of your data, and in the case of a hardware failure, it provides automatic failover.

But there are limitations. The maximum database size is 150GB, though this can be increased through sharding (the horizontal partitioning of data). This approach is also subject to more latency issues, due to the fact that it runs on a shared infrastructure in the Microsoft datacenter. Another drawback is that you must expect transient faults and program your code accordingly. Finally, Windows Azure SQL Database represents a subset of SQL Server, meaning that some features, such as XML, system stored procedures, distributed transactions, synonyms, CLR procedures, Full Text Search and the ability to have linked servers are not supported. Because of these limitations, migrating to Windows Azure SQL Database may not work for some scenarios. As you can see, this

Figure 3 **The Image Gallery for Windows Azure Virtual Machines**

option isn't as compatible with on-premises SQL Server instances as the previous option, Windows Azure SQL Server Database VM.

Considerations

When migrating your database from an on-premises solution to a public cloud, there are several aspects to consider. One important factor is performance, which takes into account CPU utilization, disk I/O, memory constraints, and network throughput and latency. You also need to think about how much disk space is needed as well as the number of VMs. Network topology is a concern if database applications require connectivity to on-premises resources, potentially requiring you to establish a VPN, which is another offering of the Windows Azure platform. Compliance and security are also extremely important, especially when it comes to customer data—specifically the location, transfer and handling of data. There are significant liabilities with respect to medical history, test and laboratory results, and insurance information, to name just a few.

Migrating a SQL Server Database to a Windows Azure VM

Although Microsoft has released a community technology preview (CTP) version of a deployment wizard, we'll take a more manual approach that gives a better idea of what happens during the deployment process. We'll essentially create a BACPAC file from our on-premises database and then import it into our Windows Azure SQL Server VM.

Before creating a BACPAC file, we recommend two preliminary steps. First, disconnect all users from the database and perform a traditional backup. This preserves the integrity of the transaction log and ensures the database is deployed correctly and in its entirety. We absolutely discourage the creation of a BACPAC file from a live production database. BACPAC files should be created from a backup image. Next, create a Windows Azure SQL Server VM. You can create your own VM and upload it if you wish, but an easier path is to leverage the Windows Azure image gallery that exists at the Windows Azure Management Portal. It allows you to simply choose from a list of available VM images.

To start, sign in to the Windows Azure Management Portal. On the command bar, click New | Virtual Machine | From Gallery. **Figure 3** lists the various flavors of SQL Server and Windows Server from which you can choose.

After you select an image from the gallery, you can choose from a list of hardware options. **Figure 4** shows that at the high end you can choose 8 cores and 56GB of RAM.

> When migrating your database from an on-premises solution to a public cloud, there are several aspects to consider.

Export a BACPAC of Your On-Premises Database First, keep in mind that this process is specific to SQL Server 2012.

1. To create a BACPAC file, start SSMS and connect to the on-premises instance.
2. Next, in Object Explorer, expand the node for the instance from which you want to export the database and create the resulting BACPAC.
3. Right-click the database name, click Tasks, then select Export Data-tier Application. A wizard will appear and you'll be asked for the storage destination of the BACPAC file.

The great thing here is that the wizard will let you store the BACPAC file in Windows Azure Storage in a Microsoft datacenter, which can dramatically simplify the import process from the Windows Azure SQL Server VM. If you don't have a storage account, you'll need to go to the Windows Azure Management Portal and create one first. Make sure to record the storage account name and management key associated with it, as they're required by the wizard. Assuming you encounter no errors, you're now finished creating the BACPAC file. Note that this process can take from several minutes to several hours depending on the size of your database. Moreover, as mentioned previously, you need to be aware that it will disrupt service to existing users who are connected, so you should first do a traditional backup before creating a BACPAC file.

The BACPAC file is just a blob in Windows Azure Storage. Once you've successfully created it, you can start Visual Studio and view the file using the Server Explorer (or similar tooling that lets you browse Windows Azure Storage). The URL for your BACPAC file will look something like this: https://[your-storage-account-name].blob.core.windows.net/[your-container-name]/[your-database-name].bacpac.

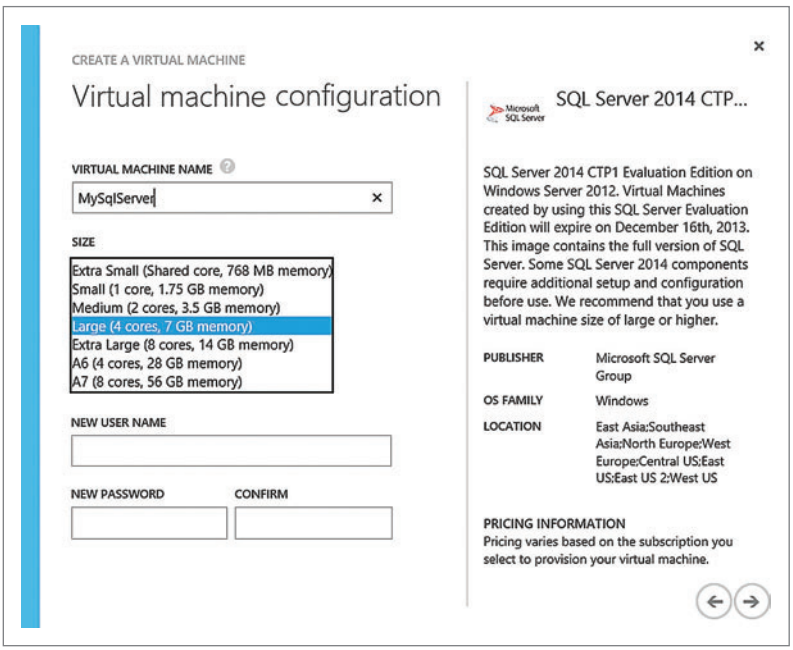


Figure 4 **Available Virtual Machine Configurations**


REPORTING
ActiveReports 7
With royalty-free licensing, create:
Form-based reports, such as invoices & insurance documents
Transaction reports, such as sales & accounting
Analytical reports, such as sales & budget analysis
& portfolio analysis
Download your free trial @
www.componentone.com
ComponentOne
a division of GrapeCity
© 2013 GrapeCity, inc. All rights reserved. All other product and brand names are trademarks and/or registered trademarks of their respective holders.

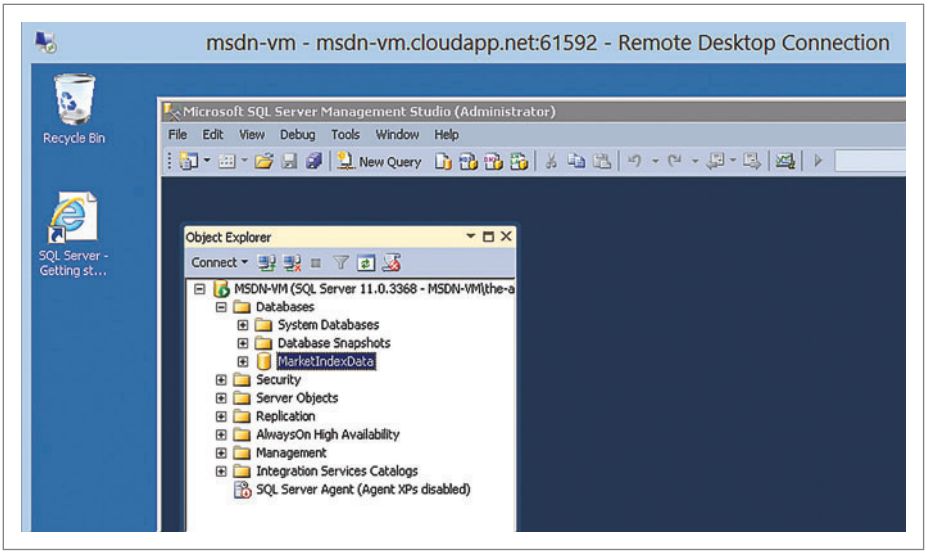


Figure 5 **Successful Import of MarketIndexData**

Remoting into Your SQL Server Database VM You now need to remote into the SQL Server Database VM you created previously. Note that the login credentials will not be your Windows Azure subscription credentials, but rather the credentials you specified when you created the VM. In our case, the login name was "msdn-vm\the-admin" plus the password we entered.

Importing the BACPAC File from Windows Azure Storage Once you're logged into the VM, you can import the BACPAC file into the cloud-hosted VM.

1. Start SSMS, then connect to the local instance of SQL Server.
2. In Object Explorer, right-click on Databases and then select the Import Data-tier Application menu item to launch the wizard. The wizard is similar to the one used to create the BACPAC for the on-premises database. Once again, enter the storage account details, such as the name of the storage account and the management key. The wizard will display the container and the name of the BACPAC file.
3. Click on Next to complete the import database process. **Figure 5** illustrates a successful import of the database into the cloud-hosted VM in a Microsoft datacenter.

Migrating a SQL Server Database to a Windows Azure SQL Database

As noted earlier, Microsoft has a PaaS database offering known as Windows Azure SQL Database. To import the BACPAC file (the packaged on-premises database you just created), log in to the Windows Azure Management Portal. Select SQL Databases, then Import from the Command Window. You'll be asked to enter the URL for the BACPAC file residing in Windows Azure storage as a blob file, as seen in **Figure 6**. This is the same URL used previously for the BACPAC file. This is one of the great advantages of BACPAC files: they can be used for importing both Windows Azure SQL Database and Windows Azure SQL Server Database VMs.

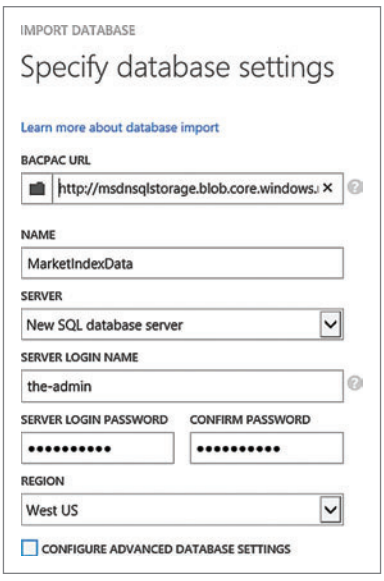


Figure 6 **Importing the BACPAC File**

When the import is complete, you'll be able to open the database in SSMS, just as we did with the Windows Azure SQL Server Database VMs. In fact, SSMS works with all four options previously presented. You'll just need to get the server name from the Windows Azure Management Portal for the database, and you'll need to enable specific client IP addresses at the portal. You'll find more information at the Tools and Utilities Support (Windows Azure SQL Database) page at bit.ly/Vh8jGg.

Wrapping Up

The Windows Azure platform has been evolving quickly, especially in the area of VMs and SQL Server database technology. Customers are aggressively migrating their databases to the cloud in an effort to reduce costs, as well as to leverage other resources, such as storage, caching, identity, VPNs, messaging and more. While Windows Azure SQL Database can offer tremendous cost advantages, it doesn't provide the level of compatibility required by many enterprise customers. In future releases of SSMS (2014), deployment wizards are expected to automate the manual process illustrated in this month's column. When these wizards do arrive, the lessons learned here will ensure you know how they work and what they do. ■

Bruno Terkaly *is a developer evangelist for Microsoft. His depth of knowledge comes from years of experience in the field, writing code using a multitude of platforms, languages, frameworks, SDKs, libraries and APIs. He spends time writing code, blogging and giving live presentations on building cloud-based applications, specifically using the Windows Azure platform. You can read his blog at blogs.msdn.com/b/brunoterkaly.*

Ricardo Villalobos *is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in the supply chain management industry. Holding different technical certifications, as well as a master's degree in business administration from the University of Dallas, he works as a platform evangelist in the Globally Engaged Partners DPE group for Microsoft. You can read his blog at blog.ricardovillalobos.com.*

Terkaly and Villalobos jointly present at large industry conferences. They encourage readers of Windows Azure Insider to contact them for availability. Reach them at bterkaly@microsoft.com or Ricardo.Villalobos@microsoft.com.

Thanks to the following technical expert for reviewing this article: *Robin Shahan (consultant)*

Spread Controls for

Windows Forms & ASP.NET

WPF & Silverlight

WinRT

Spread Studio for .NET contains our new cross-platform spreadsheet controls for Windows Forms, ASP.NET, WPF, WinRT, and Silverlight in one amazing package.

Experience for yourself:

Excel®-like grid. Native Microsoft® Excel® Compatibility - same look & feel for your users

Flexible & familiar spreadsheet architecture, advanced charting & powerful formula library

Create financial modeling & risk analysis, budgeting, insurance, scientific applications & more

NEW

Spread Studio for .NET

ComponentOne®
a division of GrapeCity®

Download your free trial @
www.componentone.com

Build Office 365 Cloud Business Apps with Visual Studio 2013

Mike Morton, Jim Nakashima and Heinrich Wendel

The requirements, expectations and importance of business applications have never been greater than they are today. Modern business applications need to access data available inside and outside the organization. They need to connect individuals across the organization and help them collaborate with each other in rich and interesting ways. The applications themselves need to be available on multiple types of devices and form factors, such as smartphones, tablets, laptops and desktops with various screen sizes.

You need a platform to provide a set of services to meet the core requirements of these applications. You also need a toolset that lets you productively build those applications while integrating with existing DevOps processes across the organization.

This article discusses:

- Creating a Cloud Business App project in Visual Studio 2013
- Defining a data model for the example app
- Designing the UI
- Adding business logic
- Integrating social capabilities
- Publishing the app
- Using continuous integration

Technologies discussed:

Office 365, SharePoint, Visual Studio 2013, Team Foundation Server

In this article, we'll show how Visual Studio 2013 helps you build these modern business applications. We'll create a recruiting app that manages job postings and candidates, providing experiences that seamlessly extend the Office 365 and Windows Azure platforms and use Office 365 services such as identity and social.

We'll show how Visual Studio helps you be productive across the complete lifecycle of a Cloud Business App from building, running and testing, to publishing and using continuous integration.

Creating a New Project

Start by launching Visual Studio 2013. Click File | New Project. The Cloud Business App template is available under the Apps node of Office/SharePoint for both Visual Basic and Visual C# (see **Figure 1**). This categorization is based on the language used in the middle tier; the client is HTML and JavaScript.

A Cloud Business App comprises four projects:

- The Server project, which is a basic ASP.NET project used to add tables and connect to data sources
- The standard app-for-SharePoint project, which provides a connection to Office 365
- The HTMLClient project, a JavaScript project in which you define the UI for your app (the screens)
- The Cloud Business App project, which ties all of the projects together. You can see the Visual Studio project structure in **Figure 2**.

We'll go through how to use each of these projects.

Defining the Data Model

Data lies at the heart of every business application. Cloud Business Apps offer a variety of ways to interact with that data. Let's start by defining a new data model using the Table Designer. We'll later deploy this data model to a Windows Azure SQL Database. We'll use the Table Designer to define a Candidate entity. **Figure 3** shows the details of the data model. Entities are made up of properties, which are either a simple data type such as string or integer, or a business type such as URL, Email Address or Person. Specific validation logic and unique visualizations are built into the Visual Studio tooling and runtime.

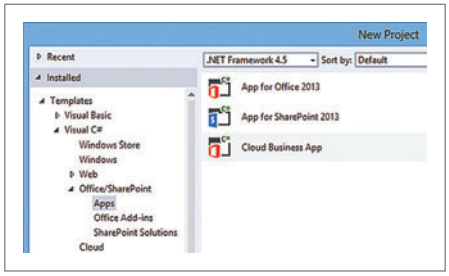


Figure 1 **Creating a New Cloud Business App Project in Visual Studio 2013**

A Cloud Business App is an app for SharePoint. It has access to data in the Office 365 tenant where it runs. Information workers use SharePoint on a daily basis to create, manage and share data in a collaborative environment. For our scenario, we'll assume a SharePoint list is currently being used to track open job postings. **Figure 4** shows this list, including data that was entered.

Let's attach data to this list using the Add Data Source action in the context menu of the Data Sources folder. Select SharePoint in the first page of the wizard and enter the URL for the SharePoint site against which you plan to develop. (If you don't have a SharePoint site for development, you can get one from dev.office.com—it's free for a year with an MSDN subscription.) The next page shows all lists and document libraries available on the given SharePoint site. Import the Job Postings list by selecting it and finishing the dialog.

Relating data from two different data sources—SharePoint and SQL in this case—and writing business logic that works over both is one of the most powerful features you get when building Cloud Business Apps. Let's add a relationship between Candidates and Job Postings using the Table Designer. We'll set up a one-to-many relationship using the Add Relationship action in the Table Designer. We'll specify that every Candidate can apply to just one Job Posting. **Figure 5** shows the exact configuration of the relationship. Now that we've defined the data model, we'll change our focus to providing a UI for the app.

Designing the UI

In today's world it's no longer enough to create UIs that just work. Users expect apps that work across a breadth of devices, from smartphones and tablets to traditional desktops. They also expect apps to be usable with touch as well as keyboards for heavy data input. Apps should look consistent throughout the company and have specific branding. Cloud Business Apps provide you with templates, layouts and controls to easily accomplish these nontrivial tasks.

We'll create a set of three screens and define the navigation flow between them. The first screen will show a tiled list to let you browse all candidates in the database. From here, you can navigate to a second screen to add a new candidate, or drill into the details of an existing candidate on a third screen.

Add the first screen by selecting the Add Screen action in the context menu of the Screens folder in the HTMLClient project. The screen template dialog offers three templates you can use as good starting points. For the first screen, we'll use the "Browse Data Screen" template. Make sure the Candidate entity is selected in the Screen Data dropdown and complete the dialog.

> Data lies at the heart of every business application. Cloud Business Apps offer a variety of ways to interact with data.

The Screen Designer displayed in **Figure 6** is composed of two parts. The data bound to the screen is shown in the side panel on the left, and a hierarchical view of your screen is in the main area.

Select the List in the screen designer and set up the Item Tap Action from its properties sheet. Instead of writing your own method using JavaScript code, you can select from a set of commonly used predefined methods. Choose Candidates.viewSelected, and

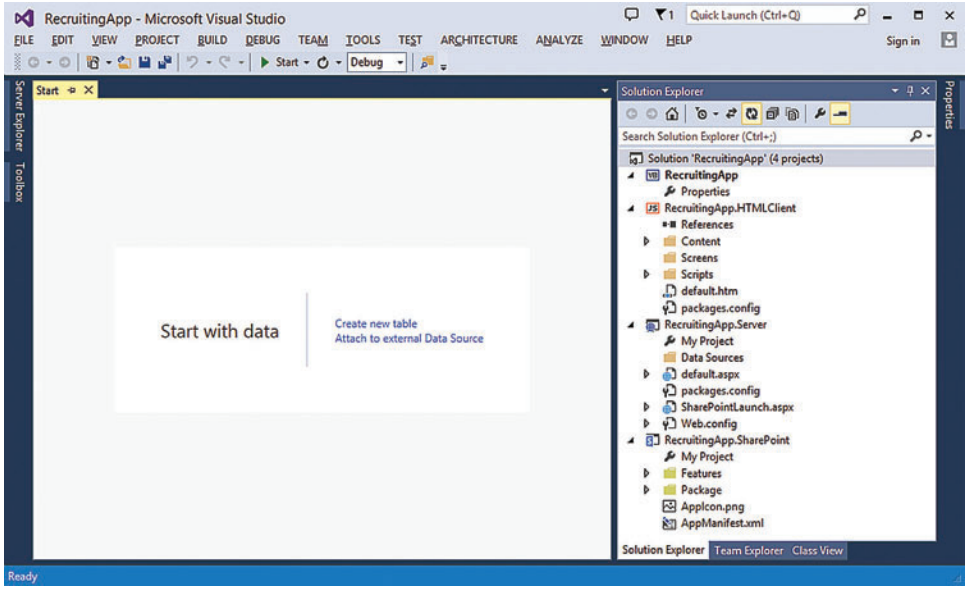


Figure 2 **Cloud Business App Structure**

Figure 3 **Candidate Entity Details**

Property Name	Data Type
Name	String
Phone	Phone Number
Email	Email Address
ReferredBy	Person
InterviewDate	Date (not required)

follow the wizard to create the details screen. Back on the first screen, select the Command Bar and add a new button. This time use the Candidates.addAndEditNew method and create an Add Edit Details Screen.

Now that the basic screens and navigation between them are defined, let's tailor their appearance. You can render a list of entities in three ways: using the standard list, a tabular view or a tile list. The list is a good fit for more complex layouts, whereas the tabular view is optimized for data-intensive tasks.

For now, we'll choose the tile list, which makes better use of screen real estate and scales best across different screen sizes. Remove all child items from the tile list node except Name, Phone and Email. Finally, select a better visualization for the command from the set of icons included by changing the Icons property in the properties sheet.

The App in Action

It's time to take a look at the running app. Press F5 in Visual Studio to run it in the browser. The first time the app is deployed to your SharePoint developer site, a dialog asks you to approve a set of permissions. Subsequent F5s will skip this step unless you change the SharePoint project within your solution.

The app is loaded using the Single-Page Application (SPA) architecture with the UI completely rendered by client-side code. Communication with the server is performed using asynchronous calls, resulting in a fast and fluid experience. One example where you can see this is the list of candidates on the home screen, which uses incremental scrolling. When reaching the end of the lists, the next set of items is dynamically loaded in the background without blocking the user.

All layouts adapt well to different form factors using responsive design techniques. Resizing the browser window gives you a quick way to see how the app looks on a phone or tablet, as shown in **Figure 7**. Everything has been optimized for touch but works equally well on a desktop using the keyboard and mouse.

Now let's add a new candidate. The dialog displayed gives you the expected set of input fields and a picker that lets the user choose the Job Position from the attached SharePoint List. Select the ReferredBy value using an auto-complete textbox that shows suggestions based on your organization's user directory—for example, something like Active Directory.

You can see additional Office 365 integration on the details screen, which shows presence information for each person. A Lync contact card shows detailed information and allows for rich interaction such as initiating an IM conversation, sending an e-mail or scheduling a meeting.

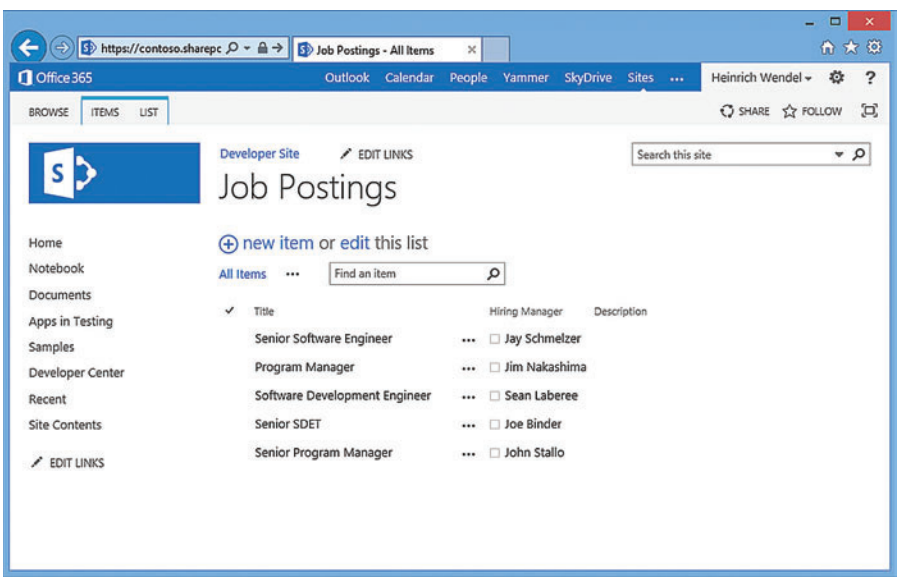


Figure 4 **A Job Posting List in SharePoint**

NEW HOSTING

> 300 Gbit/s network connection

Geo-redundancy

Web analytics

NetObjects Fusion® 2013

PHP 5.4

CDN

Free mode or safe mode

MySQL

SEO

Newsletter Tool

Over 140 apps

Drupal™, WordPress, Joomla!™, TYPO3, Magento® and many more...

Guaranteed Performance

Daily Backup

Mobile Website Builder

Adobe® Dreamweaver® CS5.5 included

COMPLETE PACKAGES FOR PROFESSIONALS STARTING AT

$ 1.99 /month*

MEMBER OF united internet

Call 1 (877) 461-2631
or buy online

1 TRIAL 30 DAY MONEY BACK GUARANTEE

1 MONTH FLEXIBLE PAYMENT OPTIONS

1 CALL SPEAK WITH AN EXPERT 24/7

1&1®

1and1.com

* 1&1 Web Hosting packages come with a 30 day money back guarantee, no minimum contract term, and no setup fee. Price reflects 36 month pre-payment option for 1&1 Basic package. Price goes to regular $5.99 per month price after 36 months. Some features listed only available with package upgrade or as add-on options. See website for full details.

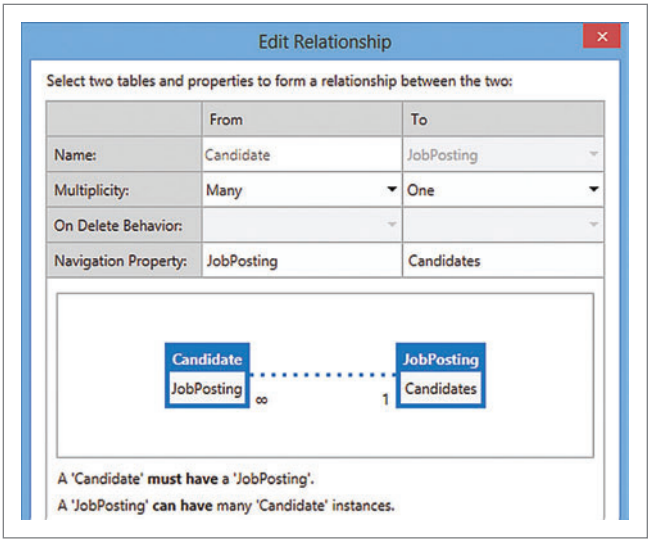


Figure 5 **The One-to-Many Relationship Between Candidates and Job Postings**

The fields that track when and by whom an entity was created and last modified are added automatically. These help fulfill common auditing requirements without writing any code. If they aren't required for your scenario, you can switch back to the screen designer without stopping the app, remove the fields and save the screen. Refresh your browser—there's no need to restart the app in Visual Studio—and the changes are reflected in the app immediately.

Adding Business Logic

The productive development experience we've described so far lets you focus your energy on the unique value of the app: the business logic. Let's say that only employees from the human resources department should be allowed to schedule interviews for candidates. This kind of organizational information is usually stored in Active Directory.

In Office 365, Active Directory is surfaced to developers via the User Profile Service. A strongly typed API has the most commonly used properties, such as a user's department. You can add custom properties specific to your organization to the User Profile service using the SharePoint Admin center. You can also retrieve these in code using standard SharePoint APIs to talk to the User Profile service directly.

Business logic is written on the middle tier, which is represented by the server project in this solution. The Table Designer Write Code dropdown provides you with an extensive set of entry points into the data pipeline of your app. Here we're handling the validate event and using the Application object to query the current logged-on user's department:

```
if (entity.InterviewDate !=
  null && Application.Current.User.Department != "HR") {
  results.AddPropertyError("Only HR can schedule interviews",
  entity.Details.Properties.InterviewDate);
}
```

> Social networks are used for communication and collaboration. Users expect modern business apps to integrate with these social networks.

Start your app again, or just refresh your browser if you didn't stop the app already. Now, trying to set the interview date for a candidate will result in an error because you aren't a member of the human resources department. This logic runs on the middle tier, which means the validation rule will run regardless of how clients connect to the service.

Integrating Enterprise Social

Social networks are used for communication and collaboration. Users expect modern business apps to integrate with these social networks. Cloud Business Apps enable social integration with just a few clicks using the SharePoint Newsfeed feature. In this first version, you must configure SharePoint to use SharePoint Newsfeeds rather than Yammer.

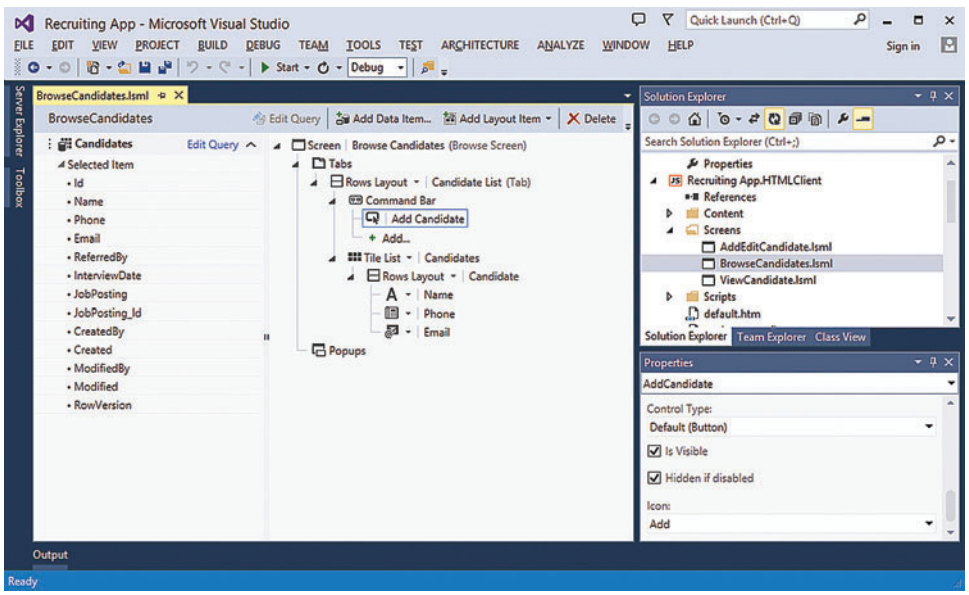


Figure 6 **The Screen Designer Showing the Browse Candidates Screen**

Design-Time Options Open the Candidate entity in the Table Designer and view the Properties window. The Social section has options for generating a post to the Newsfeed on creation and on update of the entity. Enable both of these. You can control which properties trigger a social update using the "Choose post triggers..." link. Select only the InterviewDate post trigger, as shown in **Figure 8**. The Summary property lets you select what data about the entity will be shown in the post.

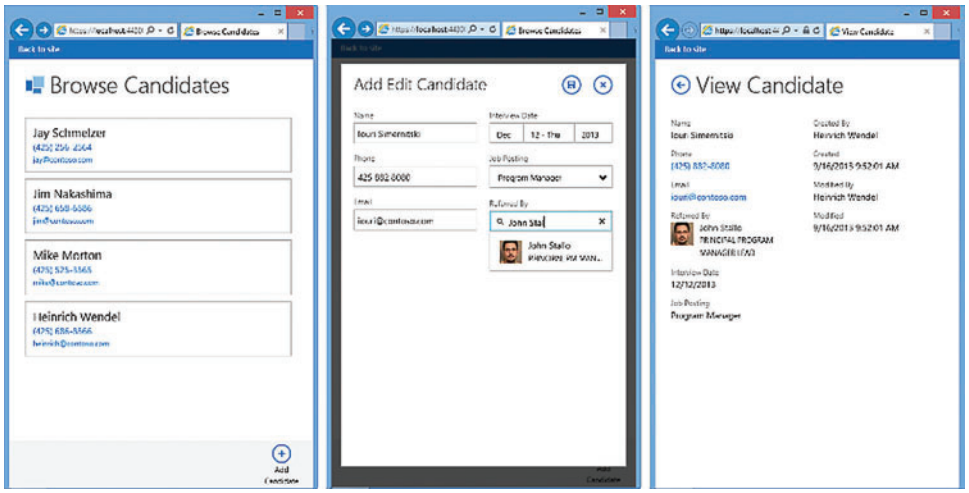


Figure 7 **All Three Screens of the Running Cloud Business App on a Mobile Device**

Runtime Newsfeed Run the app to see the social integration that has been configured. Notice the Newsfeed and Follow links in the top-right corner. The Newsfeed link opens a new window to a page that shows the app's activity. Think of this as the app's internal social feed. The Newsfeed page has an entry for each newly created Candidate and each Candidate for whom the interview date changed. Clicking on the post takes you directly to the Candidate details screen. In addition, users can start a new conversation or Like or Reply to any of the posts. Add a new Candidate or change the interview date of an existing Candidate to see examples of this activity being generated on the Newsfeed page, as shown in **Figure 9**.

Having the Newsfeed option is helpful if you want to look at the activity from within the app, but there's an easier way to follow the app's activity on a regular basis. Clicking the Follow link on the home screen will cause the same social posts and conversations to be posted to the user's personal Newsfeed. This lets the user follow what's going on in this app or any other Cloud Business App, Person or Document in a centralized location.

Publish

Now that you've built and tested your app locally, it's a good time to deploy the app to other employees in your organization. Select Publish from the context menu of the Cloud Business App project in Solution Explorer. The publish dialog walks you through the choices you have when publishing a Cloud Business App: SharePoint hosting method, hosting of app services, data connections and SharePoint authentication.

Let's take an optimal path for publishing the Cloud Business App to Office 365 using a secure Windows Azure Web Site to host the app services and Windows Azure SQL Database to host the database.

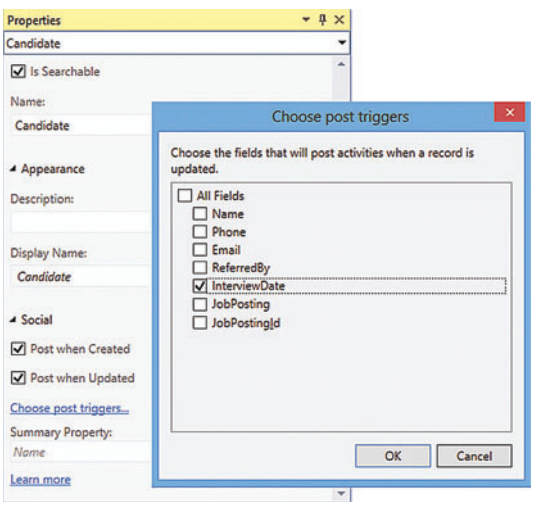


Figure 8 **Social Properties and Post Triggers Dialog**

You can publish the app to the Office Store or your company's app catalog. If the goal were to create an app available for the public, then the Office Store makes sense. If the app is intended only for internal use in your organization, then using your app catalog is the way to go. The app catalog lets you push the app out to many sites automatically. This simplifies acquisition of the app for the end users and management. Only a single instance of the app actually exists, even though users can access it from many different sites.

SharePoint Hosting Method Select the Provider-hosted option, which gives you more control over how you'll deploy different parts of the app. It also provides greater visibility into how the app is performing, makes it easy to scale the different layers of the app and provides full control over the lifetime of the app's data.

Hosting of Application Services Now you need to decide between hosting the application services in Windows Azure or on an IIS server. Windows Azure is the easiest and fastest way to create a hosting environment. You don't have to set up a machine or even configure IIS, as Windows Azure takes care of this for you.

Next, select the specific Windows Azure subscription to use. If you're signed in to your Windows Azure account, select the subscription you'd like to use. Otherwise, you'll need to sign in to see the available subscriptions.

The Service Type tab (see **Figure 10**) provides options for using a Windows Azure Web Site or Cloud Service. Choose the Web Site option. The Web Site option is a great choice for Cloud Business Apps because of how quickly you can push the app out, the low cost for hosting it and the ease with which the app can be scaled up if it becomes popular. Using the Windows Azure Web Site option still gives you access to other Windows Azure services such as SQL databases, caching, BLOB storage and service bus.

Next, select the specific Windows Azure Web Site you want to use. If none are listed, or if you want to create a new one, use the link to go to the Windows Azure Portal. Finally, choose the HTTPS option so communication is safely encrypted.

Data Connections Now it's time to configure the database connection to use for the app. If you've already linked a SQL database to the Web Site, then the connection string will be filled in correctly. If not, you can create a new linked resource by following the appropriate steps in the article, "How to Configure Web Sites," in the

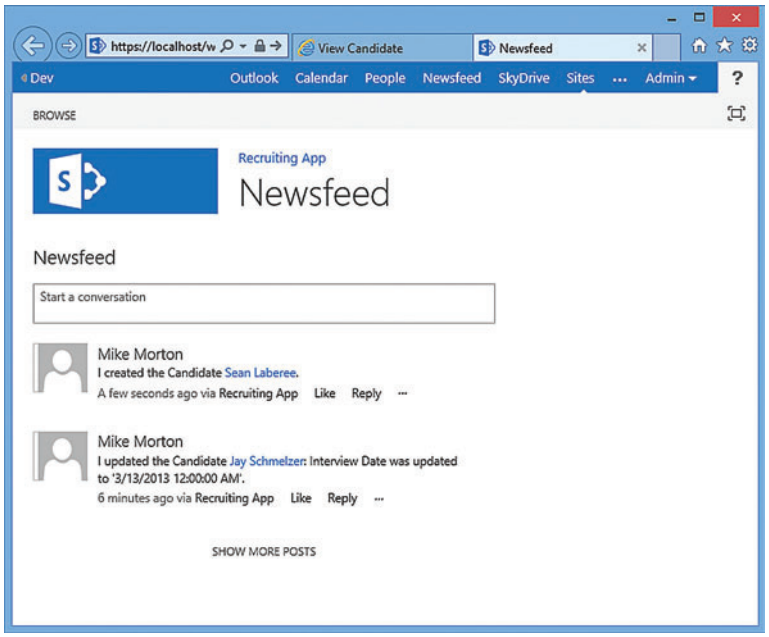


Figure 9 **The Cloud Business App Newsfeed Page**

Windows Azure documentation at bit.ly/151m3GM. You could also simply use the connection string of an existing SQL database. On the Attached Data Sources tab, make sure the connection is pointing to the appropriate endpoint—often connection endpoints are different based on whether you're publishing to production or a different environment.

SharePoint Authentication The last thing you need to configure is the type of SharePoint authentication to use. This is required to enable the application services hosted on the Web server—Windows Azure in this case—to communicate with SharePoint. In the SharePoint Authentication tab, you need to select the "Use a client secret" option because you're using Office 365. To use this type of authentication, you'll need to acquire a Client Id and Client Secret. These are basically a user ID and password for the app. They're required in order to use Open Authorization (OAuth), which is an open standard for app authorization.

We'll be publishing the app to the app catalog because we only want it to be used by people within the organization. Therefore, to get a Client Id and Client Secret, you need to do the following:

- Sign in to the Office 365 SharePoint site to which you plan to deploy the app and navigate to http://YourSharePointSiteName/_layouts/15/appregnew.aspx (**Figure 11** shows the AppRegNew.aspx page).
- Click the Generate button for both Client Id and Client Secret.
- Enter the name of your app as the Title and the domain for the Windows Azure Web Site as the App Domain.
- Leave the Redirect URI textbox empty and click the Create button.

Back on the publish dialog window in Visual Studio, copy the Client Id and Client Secret from AppRegNew.aspx confirmation page. On the next tab, the URL of your Windows Azure Web Site should already be filled in.

Publish the Cloud Business App You should now see the Summary tab showing a recap of the options you just configured. Click the Publish button to publish the Cloud Business App to the various endpoints. After deployment is finished, a File Explorer window will open showing the .app file. This is the SharePoint app package you need to upload to your app catalog. To learn more about this, see the steps in the Microsoft Office support article at bit.ly/1bnlrRY.

> Continuous integration can save you time by automatically publishing your app throughout the development process.

At this point, the app is in the app catalog and you can install it on one or more sites.

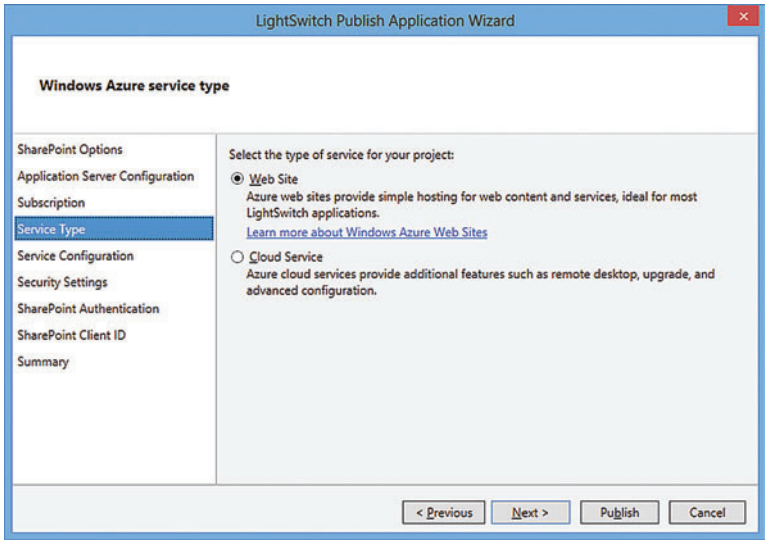


Figure 10 **The Publish Dialog**

Continuous Integration

Continuous integration can save you time by automatically publishing your app throughout the development process. In team environments where multiple developers are constantly making changes, continuous integration ensures your app will be compiled, packaged, and published in an automated and consistent manner.

Team Foundation Server To take advantage of continuous integration for Cloud Business Apps, you either need Visual Studio Team Foundation Server (TFS) 2013 installed on-premises or use TFS in the cloud, Team Foundation Service (see tfs.visualstudio.com). If you're using TFS 2013 on-premises, make sure you download the TFS build process template and install

it for your team project. Currently, the new process template only supports deploying the SharePoint part of a Cloud Business App to Office 365-hosted SharePoint sites. On-premises SharePoint sites aren't yet supported.

Create a Build Definition A build definition describes the continuous integration process and consists of a build process template (a .xaml file describing a workflow) and configuration data unique to your environment. To create a new build definition in Visual Studio, make sure you're connected to your TFS machine and your app is checked in to source code control. On the Builds panel in Team Explorer, select New Build Definition.

Following are the key aspects of a build definition and a more detailed description of the new parameters specific to Cloud Business Apps (you'll see six tabs on the left after clicking to create a new build definition):

- **General:** Select this to supply the name, description and queue processing rules.
- **Trigger:** Select this to specify when you want a build definition to start. You might want to set this to manual until you've confirmed the build definition operates as intended.
- **Source Settings:** Select this to specify what you're building.
- **Build Defaults:** Select this to specify the build controller that will process the build definition and where the output should be placed.
- **Process:** Select this to choose the build process template to use and parameters to pass into the workflow.
- **Retention Policy:** Select this to specify how long build outputs should be retained.

Three new parameters for a Cloud Business App build definition are located on the Process tab. First, select the build process template created for Cloud Business Apps. Click the "Show details" expander and choose either the TfvcContinuousDeploymentTemplate.12.xaml template or the GitContinuousDeploymentTemplate.12.xaml template. Expand the Deployment Settings section to configure the new parameters (shown in **Figure 12**), which are:

- **Path to Deployment Settings:** This parameter contains the path to an XML file that contains publishing information about the Cloud Business App, including the database connection string for the intrinsic database and endpoints for any attached data sources.
- **SharePoint Deployment Environment Name:** This parameter contains the SharePoint URL, username and password that will be used to connect to the SharePoint server where the app will be published.
- **Windows Azure Deployment Environment Name:** This parameter contains the required information for connecting to the Windows Azure subscription.

Queue a Build Definition Now that you've supplied the necessary configuration data to take advantage of continuous integration, it's

time to kick off a build. Locate the build definition you just created in the Builds panel and select Queue New Build from the context menu. Leave everything as is in the dialog that's displayed and queue the build. It will be picked up by the build controller when it's free.

To see the builds in the Build Explorer, select View My Builds from the Actions dropdown on the Builds panel. You'll see that the Build Explorer has two tabs: one for queued, in-progress and recently completed builds; and one for builds completed earlier. You can see a log for any build by selecting it in the explorer, including additional diagnostics information if the build has failed.

Get Started Today

We've shown you highly productive experiences for defining data and screens that let you quickly get an app up and running. The app has a professional-looking UI that blends with SharePoint and is integrated with a set of Office 365 services such as Lync presence and contact card, person picker and social. We've shown you how you can use the Microsoft .NET Framework to customize your app and—when you're ready—how to publish your app and configure continuous integration as part of your development process.

Start building Cloud Business Apps today. All you need is Visual Studio 2013 and an Office 365 developer subscription to use as a sandboxed developer environment. MSDN subscribers are eligible for a one-time, 12-month, single-user Office 365 Developer subscription. Go to bit.ly/1fvYeAT to learn more. ■

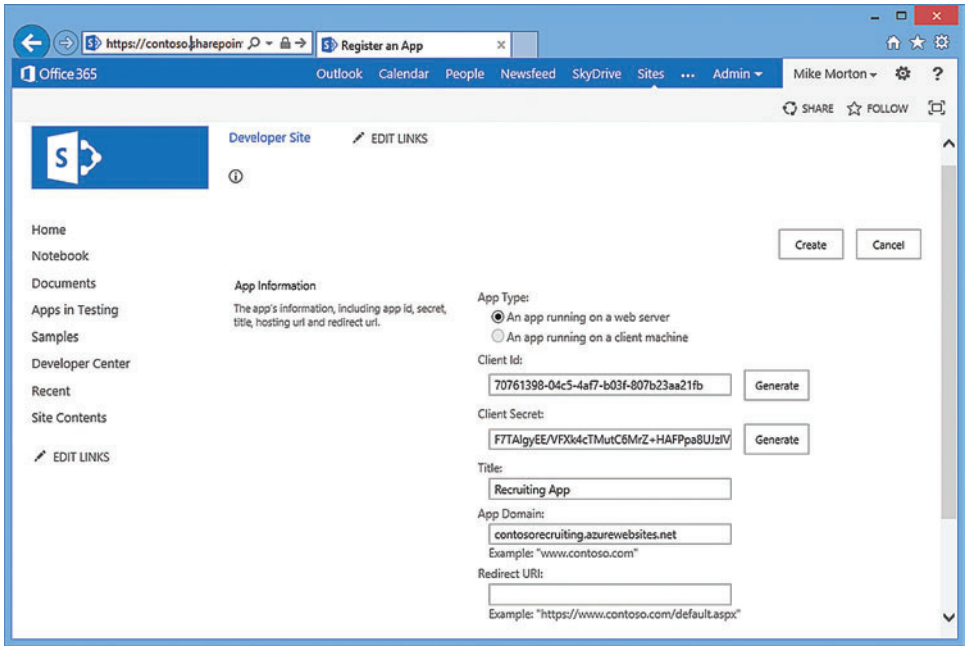


Figure 11 **The AppRegNew.aspx Page**

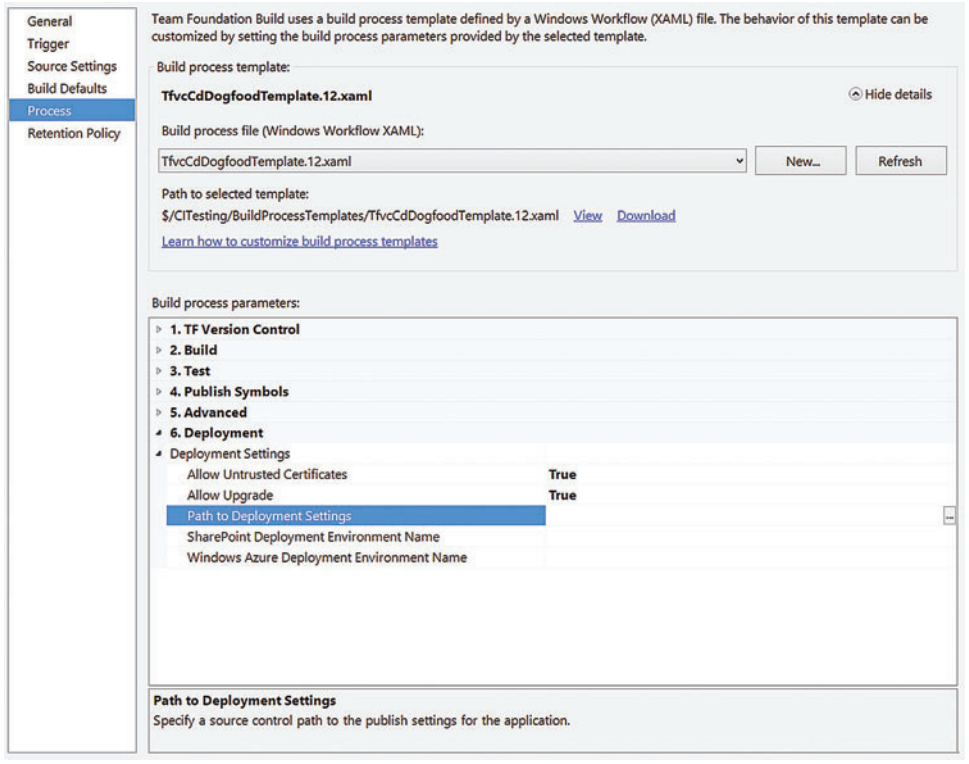


Figure 12 **Build Definition Dialog**

Mike Morton *has worked at Microsoft for many years in both the Information Technology group and in the product groups as a developer, architect and other roles. He is currently a program manager on the Visual Studio team that creates the tools for developing apps for Microsoft Office and SharePoint as well as the new Cloud Business Apps. Recently Morton has spent time building the new "Napa" tools, which work completely in the browser.*

Jim Nakashima *is a principal PM on the Visual Studio Cloud Platform Tools team focusing on building great end-to-end developer experiences for Microsoft Office and SharePoint. Prior to his current role, Nakashima spent a number of years working on Windows Azure, which fueled his interest in the cloud and developer services that he now applies to Visual Studio.*

Heinrich Wendel *is a program manager on the Visual Studio team that builds the Cloud Business Apps tooling at Microsoft. Before joining the product group, he worked as a consultant designing business solutions for customers leveraging the SharePoint platform. Wendel is passionate about the modern UX and tries to bring its benefits to developers and end users alike.*

Thanks to the following Microsoft technical experts for reviewing this article: *Joe Binder and Brian Moore*

JOIN US AT BOOTH #205 AT

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

Document	Medical	Multimedia	Imaging
OCR, Barcode & Forms Recognition	*DICOM*	*Playback, Capture & Conversion*	*Viewers*
PDF Read, Write & Edit	*PACS*	*MPEG-2 Transport Stream*	*Image processing*
Cleanup and Preprocessing	*Medical Workstation*	*DVR*	*150+ Formats*

C++ C# JavaScript VB Objective-C Java

.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

LIVE EVENT

20 LIVE! CELEBRATING TWENTY YEARS

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

WE'VE GOT YOUR TICKET TO CODE!

INTENSE TAKE-HOME TRAINING FOR DEVELOPERS, SOFTWARE ARCHITECTS AND DESIGNERS

Celebrating 20 years of education and training for the developer community, Visual Studio Live! returns to Orlando – and we've got your ticket to Code! Visual Studio Live! Orlando is where developers, software architects and designers will connect for five days of unbiased and cutting-edge education on the Microsoft platform.

SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

WHETHER YOU ARE A:

- Developer
- Programmer
- Software Architect
- Software Designer

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

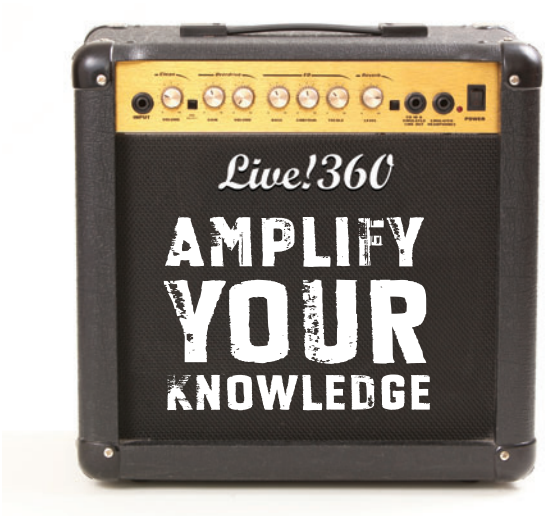
SESSIONS ARE FILLING UP FAST - REGISTER TODAY!

Use Promo Code ORLNOV4

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

ORLANDO | NOVEMBER 18-22, 2013

Royal Pacific Resort at Universal Orlando



LIVE! 360
IT EVENTS WITH PERSPECTIVE

Visual Studio Live! Orlando is part of Live! 360, the ultimate IT and Developer line-up. This means you'll have access to four (4) events, 20 tracks, and hundreds of sessions to choose from – mix and match sessions to create your own, custom event line-up – it's like no other conference available today!

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

SQL Server LIVE!
TRAINING FOR DBAs AND IT PROS

SharePoint LIVE!
TRAINING FOR COLLABORATION

Modern Apps LIVE!
MODERN APPS FROM START TO FINISH

TURN THE PAGE FOR MORE EVENT DETAILS

vslive.com/orlando | live360events.com

ORLANDO | **NOVEMBER 18-22, 2013**

Royal Pacific Resort at Universal Orlando

AGENDA AT-A-GLANCE

Windows 8.1 / WinRT	Web and JavaScript Development	Visual Studio / .NET Framework	Azure / Cloud Computing	Mobile Development	Data Management

START TIME	END TIME	Visual Studio Live! Pre-Conference: Sunday, November 17, 2013
4:00 PM	9:00 PM	Pre-Conference Registration - Royal Pacific Resort Conference Center
6:00 PM	9:00 PM	Dine-A-Round Dinner @ Universal CityWalk

START TIME	END TIME	Visual Studio Live! Pre-Conference Workshops: Monday, November 18, 2013 (Separate entry fee required)		
6:30 AM	8:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries		
8:00 AM	5:00 PM	**VSM01** - Build an Application in a Day on Windows 8 - *Philip Japikse*	**VSM02** - Data-Centric Single Page Applications with Knockout, JQuery, Breeze, and Web API - *Brian Noyes*	**VSM03** - End-to-End Service Orientation: Designing, Developing, & Implementing Using WCF and the Web API - *Miguel Castro*
5:00 PM	7:00 PM	EXPO Preview		
7:00 PM	8:00 PM	Live! 360 Keynote: To Be Announced		

START TIME	END TIME	Visual Studio Live! Day 1: Tuesday, November 19, 2013			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:00 AM	Visual Studio Live! Keynote: To Be Announced			
9:00 AM	9:30 AM	Networking Break • Visit the EXPO			
9:30 AM	10:45 AM	**VST01** -What's New in Windows 8.1 for Developers - *Brian Peek*	**VST02** - Patterns for JavaScript - *John Papa*	**VST03** - Overview and What's New in Windows Azure - *Eric D. Boyd*	**VST04** - What's New in Visual Studio 2013? - *Brian Noyes*
11:00 AM	12:15 PM	**VST05** - Controlling Hardware Using Windows 8.1 - *Brian Peek*	**VST06** - Jump-Start for Building Single Page Apps - *John Papa*	**VST07** - IaaS in Windows Azure with Virtual Machines - *Eric D. Boyd*	**VST08** - What's New in the .NET 4.5 BCL - *Jason Bock*
12:15 PM	2:00 PM	Lunch • Visit the EXPO			
2:00 PM	3:15 PM	**VST09** - A Lap Around Windows Phone 8 Development - *David Isbitski*	**VST10** - WCF & Web API: Can We All Just Get Along?!? - *Miguel Castro*	**VST11** - Solving Security and Compliance Challenges with Hybrid Clouds - *Eric D. Boyd*	**VST12** - How to Be a C# Ninja in 10 Easy Steps - *Benjamin Day*
3:15 PM	4:15 PM	Networking Break • Visit the EXPO			
4:15 PM	5:30 PM	**VST13** - Developing a Modern Mobile App Strategy - *Todd Anglin*	**VST14** - Building Rich Data HTML Client Apps with Breeze.js - *Brian Noyes*	**VST15** - Cloud Connected Apps with Azure Mobile Services - *David Isbitski*	**VST16** - Software Testing with Visual Studio 2013 and Team Foundation Server 2013 - *Benjamin Day*
5:30 PM	7:30 PM	Exhibitor Reception			

START TIME	END TIME	Visual Studio Live! Day 2: Wednesday, November 20, 2013			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:00 AM	Live! 360 Keynote: To Be Announced			
9:15 AM	10:30 AM	**VSW01** - What's New in WPF 4.5? - *Walt Ritscher*	**VSW02** - Slice Development Time with ASP.NET MVC, Visual Studio, and Razor - *Philip Japikse*	**VSW03** - Build the Next YouTube: Windows Azure Media Services - *Sasha Goldshtein*	**VSW04** - NoSQL for the SQL Guy - *Ted Neward*
10:30 AM	11:00 AM	Networking Break • Visit the EXPO			
11:00 AM	12:15 PM	**VSW05** - Learning UX Design Principles Through Windows 8 Apps - *Billy Hollis*	**VSW06** - Doing More with LESS for CSS - *Todd Anglin*	**VSW07** - JavaScript, Meet Cloud: Node.js on Windows Azure - *Sasha Goldshtein*	**VSW08** - Gaining the Knowledge of the Open Data Protocol (OData) - *Christopher Woodruff*
12:15 PM	1:45 PM	Round Table Lunch • Visit the EXPO			
1:45 PM	3:00 PM	**VSW09** - Building Windows Store Business Apps with Prism - *Brian Noyes*	**VSW10** - Building Mobile Cross-Platform Apps with HTML5 & PhoneGap - *Nick Landry*	**VSW11** - Applied Windows Azure - *Vishwas Lele*	**VSW12** - Seeking Life Beyond Relational: RavenDB - *Sasha Goldshtein*
3:00 PM	4:00 PM	Networking Break • Visit the EXPO • Expo Raffle @ 3:30 p.m.			
4:00 PM	5:15 PM	**VSW13** - Migrating from WPF to WinRT - *Rockford Lhotka*	**VSW14** - Connecting to Data from Windows Phone 8 - *Christopher Woodruff*	**VSW15** - Windows Azure in the Enterprise: Hybrid Scenarios - *Vishwas Lele*	**VSW16** - Session To Be Announced
8:00 PM	10:00 PM	Live! 360 Evening Event			

START TIME	END TIME	Visual Studio Live! Day 3: Thursday, November 21, 2013			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**VSH01** - Windows 8 HTML/JS Apps for the ASP.NET Developer - *Adam Tuliper*	**VSH02** - Create Data Driven Web Sites with WebMatrix 3 and ASP.NET Web Pages - *Rachel Appel*	**VSH03** - Making the Most of the TFS Service - *Brian Randell*	**VSH04** - iOS Development Survival Guide for the .NET Guy - *Nick Landry*
9:30 AM	10:45 AM	**VSH05** - Interaction and Navigation Patterns in Windows 8 Apps - *Billy Hollis*	**VSH06** - Building Real-Time, Multi-User Interactive Web and Mobile Applications Using SignalR - *Marcel de Vries*	**VSH07** - Continuous Integration Builds and TFS - *Brian Randell*	**VSH08** - iOS Development Survival Guide for the .NET Guy, Part 2 - *Nick Landry*
11:00 AM	12:15 PM	**VSH09** - Enhancing the Windows 8 Start Screen with Tiles, Toast and Notifications - *Walt Ritscher*	**VSH10** - Build Data-Driven Mobile Web Apps with ASP.NET MVC and jQuery Mobile - *Rachel Appel*	**VSH11** - IntelliTrace, What is it and How Can I Use it to My Benefit? - *Marcel de Vries*	**VSH12** - Session To Be Announced
12:15 PM	1:30 PM	Lunch			
1:30 PM	2:45 PM	**VSH13** - Adventures in Unit Testing: TDD vs. TED - *Ben Hoelting*	**VSH14** - Maximizing Entity Framework 6 in Your Web Projects - *Adam Tuliper*	**VSH15** - Modern .NET Development Practices and Principles - *Jason Bock*	**VSH16** - Sharing Up to 80% of Code Building Mobile Apps for iOS, Android, WP 8 and Windows 8 - *Marcel de Vries*
3:00 PM	4:15 PM	**VSH17** - Blend for Visual Studio: Design Your XAML or HTML5\CSS3 UI Faster - *Ben Hoelting*	**VSH18** - Patterns and Tools for Parallel Programming - *Marcel de Vries*	**VSH19** - Static Analysis in .NET - *Jason Bock*	**VSH20** - Talk to Me. Using Speech in Your Windows Phone App - *Walt Ritscher*
4:30 PM	5:45 PM	Live! 360 Conference Wrap-up			

START TIME	END TIME	Visual Studio Live! Post-Conference Workshops: Friday, November 22, 2013 (Separate entry fee required)	
7:00 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries	
8:00 AM	5:00 PM	**VSF01** - Workshop: NoSQL - *Ted Neward*	**VSF02** - Workshop: Visual Studio ALM—To Infinity and Beyond - *Brian Randell*

**Speakers and sessions subject to change*

Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET

Mike Wasson

Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work happens on the client side, in JavaScript. For the traditional ASP.NET developer, it can be difficult to make the leap. Luckily, there are many open source JavaScript frameworks that make it easier to create SPAs.

In this article, I'll walk through creating a simple SPA app. Along the way, I'll introduce some fundamental concepts for building SPAs, including the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) patterns, data binding and routing.

This article discusses:

- Creating the service layer and an AJAX Web client for the sample app
- The MVC and MVVM patterns
- Data binding
- Building a Web client using Knockout.js
- Building a Web client using Ember.js

Technologies discussed:

Single-Page Applications, ASP.NET Web API, Knockout.js, Ember.js, AJAX and HTML5

Code download available at:

github.com/MikeWasson/MoviesSPA

About the Sample App

The sample app I created is a simple movie database, shown in **Figure 1**. The far-left column of the page displays a list of genres. Clicking on a genre brings up a list of movies within that genre. Clicking the Edit button next to an entry lets you change that entry. After making edits, you can click Save to submit the update to the server, or Cancel to revert the changes.

I created two different versions of the app, one using the Knockout.js library and the other using the Ember.js library. These two libraries have different approaches, so it's instructive to compare them. In both cases, the client app was fewer than 150 lines of JavaScript. On the server side, I used ASP.NET Web API to serve JSON to the client. You can find source code for both versions of the app at github.com/MikeWasson/MoviesSPA.

(Note: I created the app using the release candidate [RC] version of Visual Studio 2013. Some things might change for the released to manufacturing [RTM] version, but they shouldn't affect the code.)

Background

In a traditional Web app, every time the app calls the server, the server renders a new HTML page. This triggers a page refresh in the browser. If you've ever written a Web Forms application or PHP application, this page lifecycle should look familiar.

In an SPA, after the first page loads, all interaction with the server happens through AJAX calls. These AJAX calls return data—*not* markup—usually in JSON format. The app uses the JSON data to update the page dynamically, without reloading the page. **Figure 2** illustrates the difference between the two approaches.

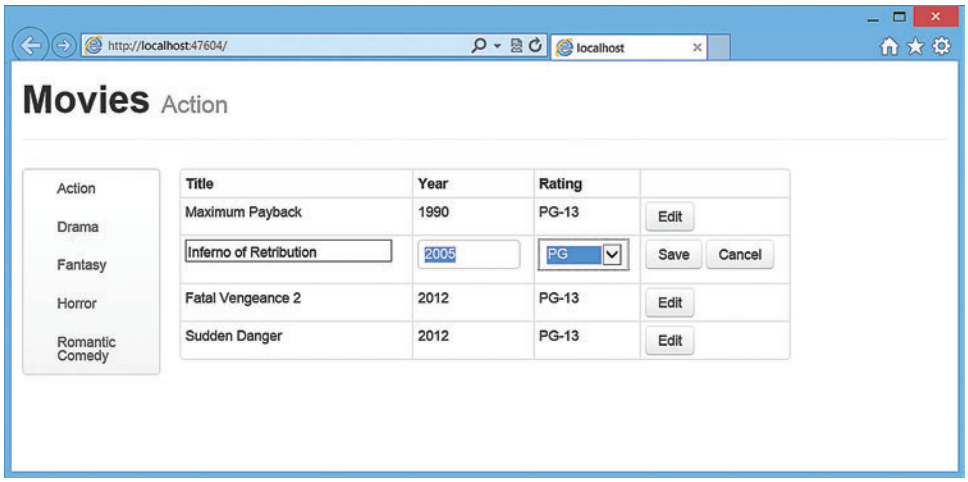


Figure 1 **The Single-Page Application Movie Database App**

purely as a service layer. The client just needs to know what HTTP requests to send. It doesn't care how the server implements things on the back end.

With this architecture, the client and the service are independent. You could replace the entire back end that runs the service, and as long as you don't change the API, you won't break the client. The reverse is also true—you can replace the entire client app without changing the service layer. For example, you might write a native mobile client that consumes the service.

One benefit of SPAs is obvious: Applications are more fluid and responsive, without the jarring effect of reloading and re-rendering the page. Another benefit might be less obvious and it concerns how you architect a Web app. Sending the app data as JSON creates a separation between the presentation (HTML markup) and application logic (AJAX requests plus JSON responses).

This separation makes it easier to design and evolve each layer. In a well-architected SPA, you can change the HTML markup without touching the code that implements the application logic (at least, that's the ideal). You'll see this in action when I discuss data binding later.

In a pure SPA, all UI interaction occurs on the client side, through JavaScript and CSS. After the initial page load, the server acts

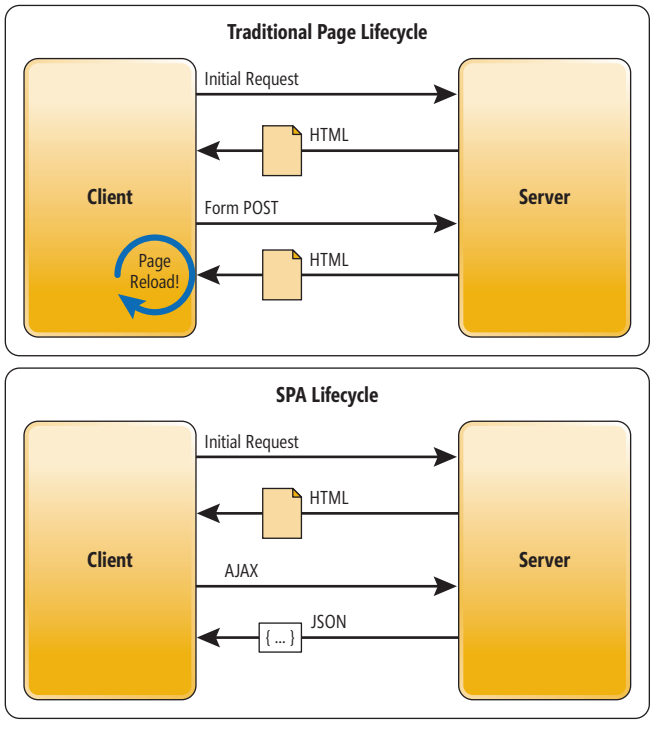


Figure 2 **The Traditional Page Lifecycle vs. the SPA Lifecycle**

Creating the Visual Studio Project

Visual Studio 2013 has a single ASP.NET Web Application project type. The project wizard lets you select the ASP.NET components to include in your project. I started with the Empty template and then added ASP.NET Web API to the project by checking Web API under "Add folders and core references for:" as shown in **Figure 3**.

The new project has all the libraries needed for Web API, plus some Web API configuration code. I didn't take any dependency on Web Forms or ASP.NET MVC.

Notice in **Figure 3** that Visual Studio 2013 includes a Single Page Application template. This template installs a skeleton SPA built on Knockout.js. It supports log in using a membership database or external authentication provider. I didn't use the template in my app because I wanted to show a simpler example starting from scratch. The SPA template is a great resource, though, especially if you want to add authentication to your app.

Creating the Service Layer

I used ASP.NET Web API to create a simple REST API for the app. I won't go into detail about Web API here—you can read more at asp.net/web-api.

First, I created a Movie class that represents a movie. This class does two things:

- Tells Entity Framework (EF) how to create the database tables to store the movie data.
- Tells Web API how to format the JSON payload.

You don't have to use the same model for both. For example, you might want your database schema to look different from your JSON payloads. For this app, I kept things simple:

```
namespace MoviesSPA.Models
{
  public class Movie
  {
    public int ID { get; set; }
    public string Title { get; set; }
    public int Year { get; set; }
    public string Genre { get; set; }
    public string Rating { get; set; }
  }
}
```

Next, I used Visual Studio scaffolding to create a Web API controller that uses EF as the data layer. To use the scaffolding,

right-click the Controllers folder in Solution Explorer and select Add | New Scaffolded Item. In the Add Scaffold wizard, select "Web API 2 Controller with actions, using Entity Framework," as shown in **Figure 4**.

Figure 5 shows the Add Controller wizard. I named the controller MoviesController. The name matters, because the URIs for the REST API are based on the controller name. I also checked "Use async controller actions" to take advantage of the new async feature in EF 6. I selected the Movie class for the model and selected "New data context" to create a new EF data context.

The wizard adds two files:

- MoviesController.cs defines the Web API controller that implements the REST API for the app.
- MovieSPAContext.cs is basically EF glue that provides methods to query the underlying database.

Figure 6 shows the default REST API the scaffolding creates.

Values in curly brackets are placeholders. For example, to get a movie with ID equal to 5, the URI is /api/movies/5.

I extended this API by adding a method that finds all the movies in a specified genre:

```
public class MoviesController : ApiController
{
  public IQueryable<Movie> GetMoviesByGenre(string genre)
  {
    return db.Movies.Where(m =>
      m.Genre.Equals(genre, StringComparison.OrdinalIgnoreCase));
  }
  // Other code not shown
```

The client puts the genre in the query string of the URI. For example, to get all movies in the Drama genre, the client sends a GET request to /api/movies?genre=drama. Web API automatically binds the query parameter to the genre parameter in the GetMoviesByGenre method.

Creating the Web Client

So far, I've just created a REST API. If you send a GET request to /api/movies?genre=drama, the raw HTTP response looks like this:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Date: Tue, 10 Sep 2013 15:20:59 GMT
Content-Length: 240

[{"ID":5,"Title":"Forgotten
Doors","Year":2009,"Genre":"Drama","Rating":"R"},
{"ID":6,"Title":"Blue Moon June","Year":1998,"Genre":"
Drama","Rating":"PG-13"},{"ID":7,"Title":"The Edge of
the Sun","Year":1977,"Genre":"Drama","Rating":"PG-13"}]
```

Now I need to write a client app that does something meaningful with this. The basic workflow is:

- UI triggers an AJAX request
- Update the HTML to display the response payload
- Handle AJAX errors

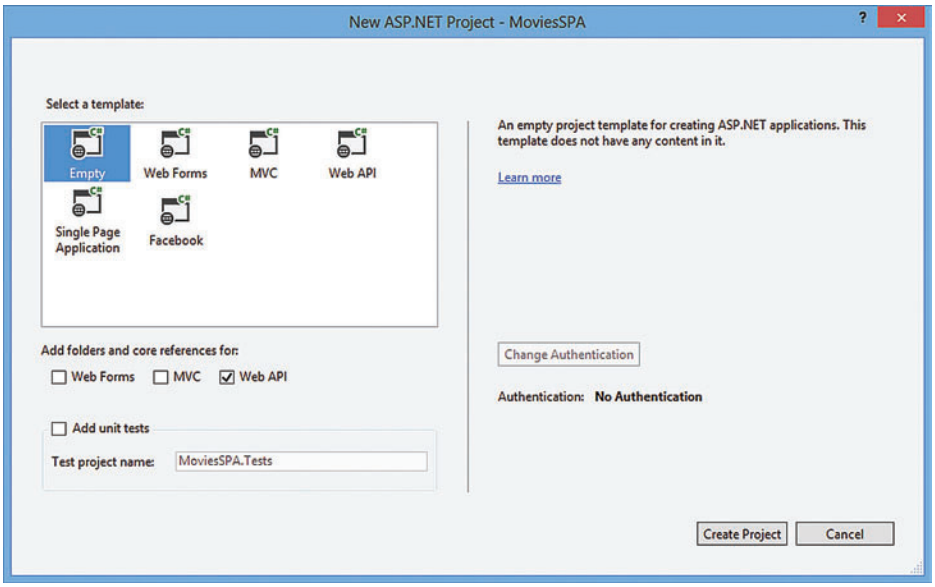


Figure 3 **Creating a New ASP.NET Project in Visual Studio 2013**

You could code all of this by hand. For example, here's some jQuery code that creates a list of movie titles:

```
$.getJSON(url)
  .done(function (data) {
    // On success, "data" contains a list of movies
    var ul = $("<ul></ul>")
    $.each(data, function (key, item) {
      // Add a list item
      $('<li>', { text: item.Title }).appendTo(ul);
    });
  $('#movies').html(ul);
});
```

This code has some problems. It mixes application logic with presentation logic, and it's tightly bound to your HTML. Also, it's tedious to write. Instead of focusing on your app, you spend your time writing event handlers and code to manipulate the DOM.

The solution is to build on top of a JavaScript framework. Luckily, you can choose from many open source JavaScript frameworks. Some of the more popular ones include Backbone, Angular, Ember, Knockout, Dojo and JavaScriptMVC. Most use some variation of the MVC or MVVM patterns, so it might be helpful to review those patterns.

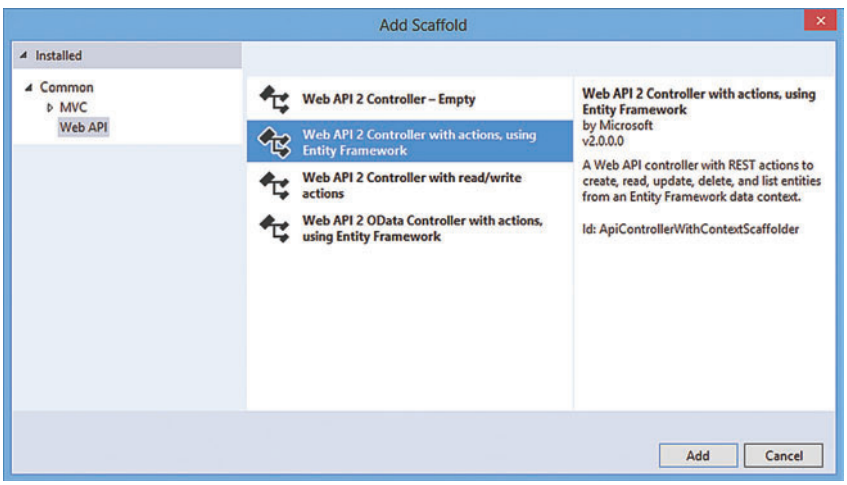
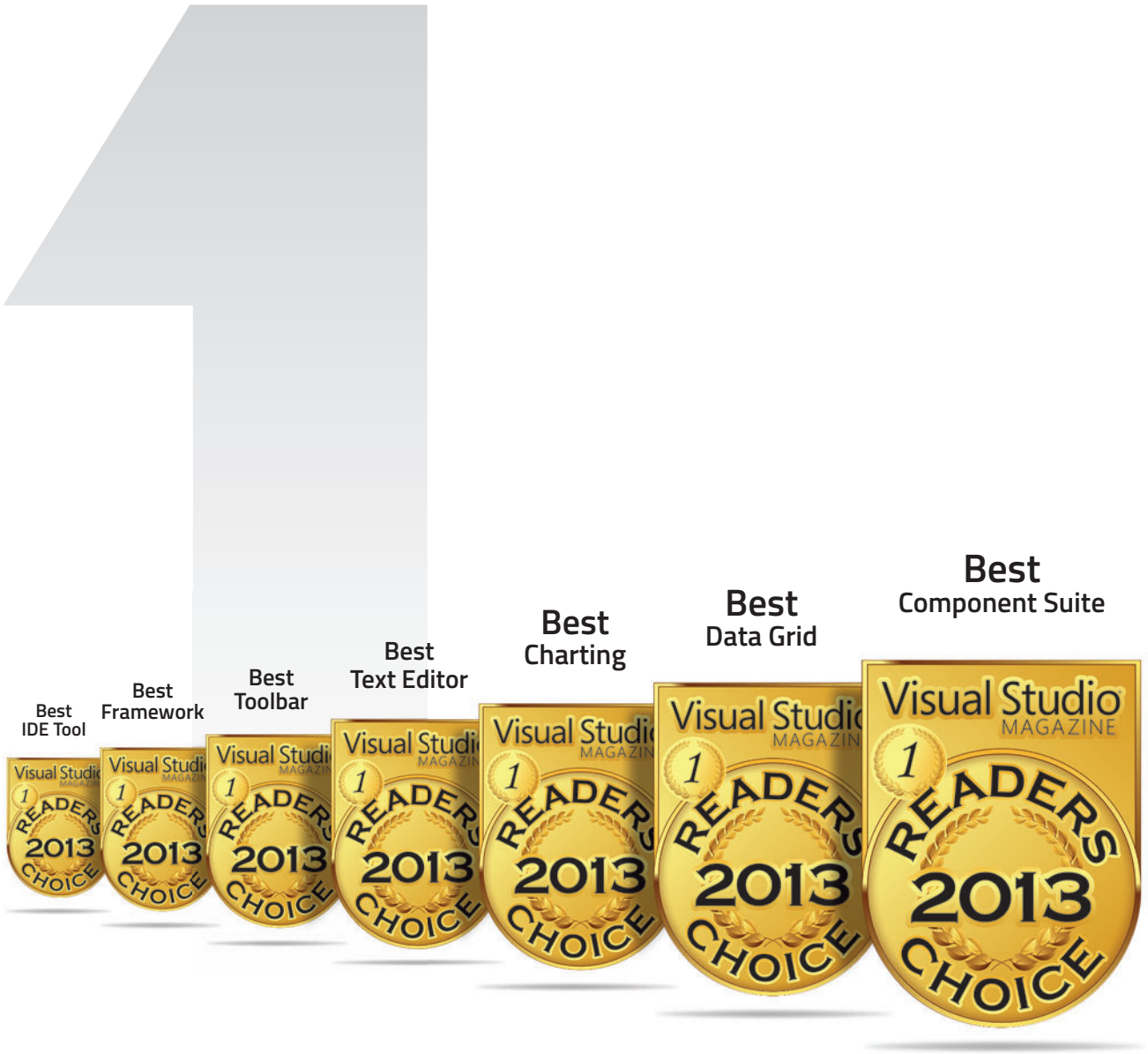


Figure 4 **Adding a Web API Controller**

Thank You!

We want to extend our deepest thanks and appreciation to all our loyal customers who took the time to vote for their favorite DevExpress products in this year's Visual Studio Magazine Reader's Choice Awards.



Learn more and download your free trial
devexpress.com/try

DevExpress™

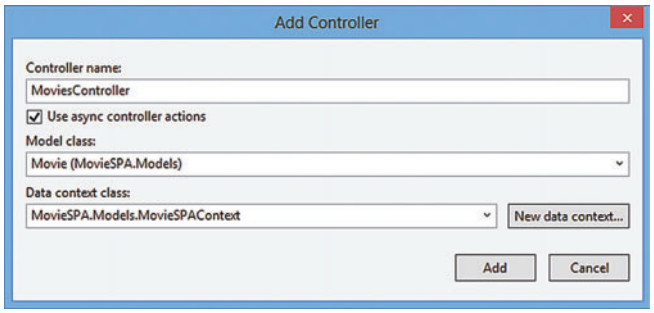


Figure 5 **The Add Controller Wizard**

The MVC and MVVM Patterns

The MVC pattern dates back to the 1980s and early graphical UIs. The goal of MVC is to factor the code into three separate responsibilities, shown in **Figure 7**. Here's what they do:

- The model represents the domain data and business logic.
- The view displays the model.
- The controller receives user input and updates the model.

A more recent variant of MVC is the MVVM pattern (see **Figure 8**). In MVVM:

- The model still represents the domain data.
- The view model is an abstract representation of the view.
- The view displays the view model and sends user input to the view model.

In a JavaScript MVVM framework, the view is markup and the view model is code.

MVC has many variants, and the literature on MVC is often confusing and contradictory. Perhaps that's not surprising for a design pattern that started with Smalltalk-76 and is still being used in modern Web apps. So even though it's good to know the theory, the main thing is to understand the particular MVC framework you're using.

Building the Web Client with Knockout.js

For the first version of my app, I used the Knockout.js library. Knockout follows the MVVM pattern, using data binding to connect the view with the view model.

To create data bindings, you add a special data-binding attribute to the HTML elements. For example, the following markup binds the span element to a property named genre on the view model. Whenever the value of genre changes, Knockout automatically updates the HTML:

```
<h1><span data-bind="text: genre"></span></h1>
```

Bindings can also work in the other direction—for example, if the user enters text into a text box, Knockout updates the corresponding property in the view model.

Figure 6 **The Default REST API Created by the Web API Scaffolding**

HTTP Verb	URI	Description
GET	/api/movies	Get a list of all movies
GET	/api/movies/{id}	Get the movie with ID equal to {id}
PUT	/api/movies/{id}	Update the movie with ID equal to {id}
POST	/api/movies	Add a new movie to the database
DELETE	/api/movies/{id}	Delete a movie from the database

The nice part is that data binding is declarative. You don't have to wire up the view model to the HTML page elements. Just add the data-binding attribute and Knockout does the rest.

I started by creating an HTML page with the basic layout, with no data binding, as shown in **Figure 9**.

(Note: I used the Bootstrap library to style the app, so the real app has a lot of extra elements and CSS classes to control the formatting. I left these out of the code examples for clarity.)

Creating the View Model

Observables are the core of the Knockout data-binding system. An observable is an object that stores a value and can notify subscribers when the value changes. The following code converts the JSON representation of a movie into the equivalent object with observables:

```
function movie(data) {
  var self = this;
  data = data || {};

  // Data from model
  self.ID = data.ID;
  self.Title = ko.observable(data.Title);
  self.Year = ko.observable(data.Year);
  self.Rating = ko.observable(data.Rating);
  self.Genre = ko.observable(data.Genre);
};
```

Figure 10 shows my initial implementation of the view model. This version only supports getting the list of movies. I'll add the editing features later. The view model contains observables for the list of movies, an error string and the current genre.

Notice that movies is an observableArray. As the name implies, an observableArray acts as an array that notifies subscribers when the array contents change.

The getByGenre function makes an AJAX request to the server for the list of movies and then populates the self.movies array with the results.

When you consume a REST API, one of the trickiest parts is handling the asynchronous nature of HTTP. The jQuery ajax function returns an object that implements the Promises API. You can use a Promise object's *then* method to set a callback that's invoked when the AJAX call completes successfully and another callback that's invoked if the AJAX call fails:

```
app.service.byGenre(genre).then(addMovies, onError);
```

Data Bindings

Now that I have a view model, I can data bind the HTML to it. For the list of genres that appears in the left side of the screen, I used the following data bindings:

```
<ul data-bind="foreach: genres">
  <li><a href="#"><span data-bind="text: $data"></span></a></li>
</ul>
```

The data-bind attribute contains one or more binding declarations, where each binding has the form "binding: expression." In this example, the foreach binding tells Knockout to loop through the contents of the genres array in the view model. For each item in the array, Knockout creates a new <li> element. The text binding in the sets the span text equal to the value of the array item, which in this case is the name of the genre.

Right now, clicking on the genre names doesn't do anything, so I added a click binding to handle click events:

```
<li><a href="#" data-bind="click: $parent.getByGenre">
  <span data-bind="text: $data"></span></a></li>
```

Windows. Web. Mobile.
Your next great app starts here.

DevExpress .NET controls, frameworks and libraries were built with you in mind. Built for those who demand the highest quality and expect the best performance... for those who require reliable tools engineered to meet today's needs and address tomorrow's requirements.

Experience the DevExpress difference today and download your free 30-day trial and let's build great apps, together.

WinForms · ASP.NET · WPF · Silverlight · Windows 8 XAML · HTML JS · Reporting · DevExtreme Mobile

Learn more and download your free trial
devexpress.com/try

DevExpress™

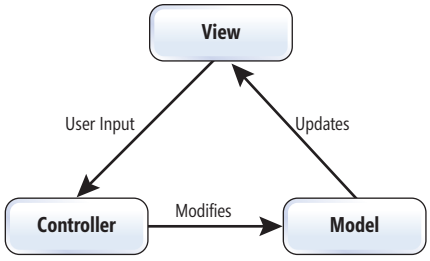


Figure 7 **The MVC Pattern**

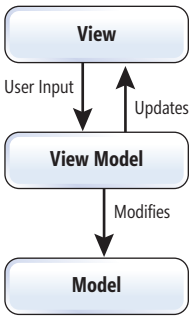


Figure 8 **The MVVM Pattern**

This binds the click event to the getByGenre function on the view model. I needed to use $parent here, because this binding occurs within the context of the foreach. By default, bindings within a foreach refer to the current item in the loop.

To display the list of movies, I added bindings to the table, as shown in **Figure 11**.

In **Figure 11**, the foreach binding loops over an array of movie objects. Within the foreach, the text bindings refer to properties on the current object.

The visible binding on the <table> element controls whether the table is rendered. This will hide the table if the movies array is empty.

Finally, here are the bindings for the error message and the "No records found" message (notice that you can put complex expressions into a binding):

```
<p data-bind="visible: error, text: error"></p>
<p data-bind="visible: !error() && movies().length == 0">No records found.</p>
```

Making the Records Editable

The last part of this app is giving the user the ability to edit the records in the table. This involves several bits of functionality:

- Toggling between viewing mode (plain text) and editing mode (input controls).
- Submitting updates to the server.
- Letting the user cancel an edit and revert to the original data.

To track the viewing/editing mode, I added a Boolean flag to the movie object, as an observable:

```
function movie(data) {
  // Other properties not shown
  self.editing = ko.observable(false);
};
```

I wanted the table of movies to display text when the editing property is false, but switch to input controls when editing is true. To accomplish this, I used the Knockout if and ifnot bindings, as shown in **Figure 12**. The "<!-- ko -->" syntax lets you include if and ifnot bindings without putting them inside an HTML container element.

The value binding sets the value of an input control. This is a two-way binding, so when the user types something in the text field or changes the dropdown selection, the change automatically propagates to the view model.

Figure 9 **Initial HTML Layout**

```
<!DOCTYPE html>
<html>
<head>
  <title>Movies SPA</title>
</head>
<body>

  <ul>
    <li><a href="#"><!-- Genre --></a></li>
  </ul>

  <table>
    <thead>
      <tr><th>Title</th><th>Year</th><th>Rating</th>
      </tr>
    </thead>
    <tbody>
      <tr><td><!-- Title --></td><td><!-- Year --></td>
        <td><!-- Rating --></td></tr>
    </tbody>
  </table>

  <p><!-- Error message --></p>
  <p>No records found.</p>
</body>
</html>
```

Figure 10 **The View Model**

```
var ViewModel = function () {
  var self = this;

  // View model observables
  self.movies = ko.observableArray();
  self.error = ko.observable();
  self.genre = ko.observable();  // Genre the user is currently browsing

  // Available genres
  self.genres = ['Action', 'Drama', 'Fantasy', 'Horror', 'Romantic Comedy'];

  // Adds a JSON array of movies to the view model
  function addMovies(data) {
    var mapped = ko.utils.arrayMap(data, function (item) {
      return new movie(item);
    });
    self.movies(mapped);
  }

  // Callback for error responses from the server
  function onError(error) {
    self.error('Error: ' + error.status + ' ' + error.statusText);
  }

  // Fetches a list of movies by genre and updates the view model
  self.getByGenre = function (genre) {
    self.error(''); // Clear the error
    self.genre(genre);
    app.service.byGenre(genre).then(addMovies, onError);
  };

  // Initialize the app by getting the first genre
  self.getByGenre(self.genres[0]);
}
// Create the view model instance and pass it to Knockout
ko.applyBindings(new ViewModel());
```

Figure 11 **Adding Bindings to the Table to Display a List of Movies**

```
<table data-bind="visible: movies().length > 0">
  <thead>
    <tr><th>Title</th><th>Year</th><th>Rating</th><th></th><th></th></tr>
  </thead>
  <tbody data-bind="foreach: movies">
    <tr>
      <td><span data-bind="text: Title"></span></td>
      <td><span data-bind="text: Year"></span></td>
      <td><span data-bind="text: Rating"></span></td>
      <td><!-- Edit button will go here --></td>
    </tr>
  </tbody>
</table>
```

I bound the button click handlers to functions named save, cancel and edit on the view model.

The edit function is easy. Just set the editing flag to true:

```
self.edit = function (item) {
  item.editing(true);
};
```

Save and cancel were a bit trickier. In order to support cancel, I needed a way to cache the original value during editing. Fortunately, Knockout makes it easy to extend the behavior of observables. The code in **Figure 13** adds a store function to the observable class. Calling the store function on an observable gives the observable two new functions: revert and commit.

Now I can call the store function to add this functionality to the model:

```
function movie(data) {
  // ...
  // New code:
  self.Title = ko.observable(data.Title).store();
  self.Year = ko.observable(data.Year).store();
  self.Rating = ko.observable(data.Rating).store();
  self.Genre = ko.observable(data.Genre).store();
};
```

Figure 14 shows the save and cancel functions on the view model.

Building the Web Client with Ember

For comparison, I wrote another version of my app using the Ember.js library.

Figure 12 **Enabling Editing of Movie Records**

```
<tr>
  <!-- ko if: editing -->
  <td><input data-bind="value: Title" /></td>
  <td><input type="number" class="input-small" data-bind="value: Year" /></td>
  <td><select class="input-small"
    data-bind="options: $parent.ratings, value: Rating"></select></td>
  <td>
    <button class="btn" data-bind="click: $parent.save">Save</button>
    <button class="btn" data-bind="click: $parent.cancel">Cancel</button>
  </td>
  <!-- /ko -->

  <!-- ko ifnot: editing -->
  <td><span data-bind="text: Title"></span></td>
  <td><span data-bind="text: Year"></span></td>
  <td><span data-bind="text: Rating"></span></td>
  <td><button class="btn" data-bind="click: $parent.edit">Edit</button></td>
  <!-- /ko -->
</tr>
```

Figure 13 **Extending ko.observable with Revert and Commit**

```
ko.observable.fn.store = function () {
  var self = this;
  var oldValue = self();

  var observable = ko.computed({
    read: function () {
      return self();
    },
    write: function (value) {
      oldValue = self();
      self(value);
    }
  });

  this.revert = function () {
    self(oldValue);
  }
  this.commit = function () {
    oldValue = self();
  }
  return this;
}
```

Figure 14 **Adding Save and Cancel Functions**

```
self.cancel = function (item) {
  revertChanges(item);
  item.editing(false);
};

self.save = function (item) {
  app.service.update(item).then(
    function () {
      commitChanges(item);
    },
    function (error) {
      onError(error);
      revertChanges(item);
    }).always(function () {
      item.editing(false);
  });
}

function commitChanges(item) {
  for (var prop in item) {
    if (item.hasOwnProperty(prop) && item[prop].commit) {
      item[prop].commit();
    }
  }
}

function revertChanges(item) {
  for (var prop in item) {
    if (item.hasOwnProperty(prop) && item[prop].revert) {
      item[prop].revert();
    }
  }
}
```

An Ember app starts with a routing table, which defines how the user will navigate through the app:

```
window.App = Ember.Application.create();

App.Router.map(function () {
  this.route('about');
  this.resource('genres', function () {
    this.route('movies', { path: '/:genre_name' });
  });
});
```

The first line of code creates an Ember application. The call to Router.map creates three routes. Each route corresponds to a URI or URI pattern:

```
/#/about
/#/genres
/#/genres/genre_name
```

Figure 15 **The Application-Level Handlebars Template**

```
<script type="text/x-handlebars" data-template-name="application">
  <div class="container">
    <div class="page-header">
      <h1>Movies</h1>
    </div>
    <div class="well">
      <div class="navbar navbar-static-top">
        <div class="navbar-inner">
          <ul class="nav nav-tabs">
            <li>{{#linkTo 'genres'}}Genres{{/linkTo}} </li>
            <li>{{#linkTo 'about'}}About{{/linkTo}} </li>
          </ul>
        </div>
      </div>
    </div>
    <div class="container">
      <div class="row">{{outlet}}</div>
    </div>
  </div>
  <div class="container"><p>&copy;2013 Mike Wasson</p></div>
</script>
```

GdPicture.NET 10

- Document viewing, processing, printing and scanning (TWAIN & WIA).
- Reading, writing and converting vector and raster images in more than 90 formats, PDF included.
- OMR, OCR, barcode reading and writing (linear & 2D).
- Annotations for image and PDF within Windows and Web applications.
- Color detection engine for image and PDF compression.

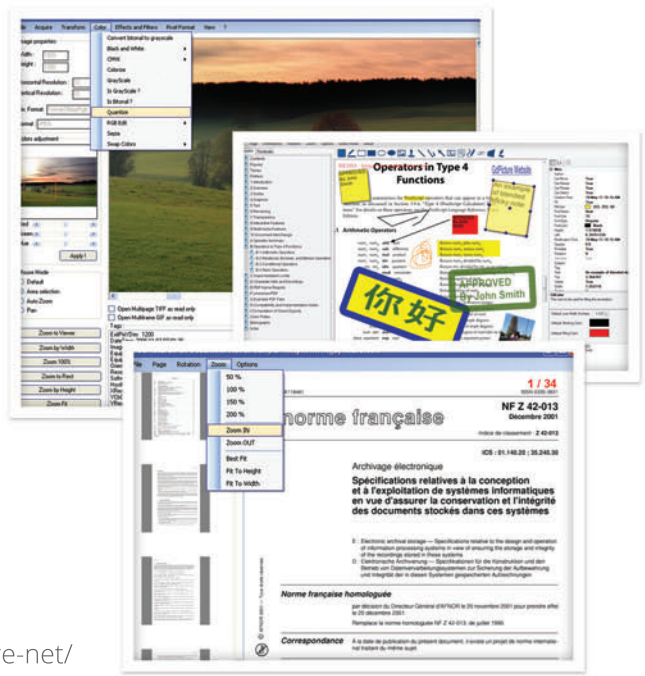
And much more ...

All-In-One Document Imaging SDK

Royalty-Free Document Imaging Toolkits for .NET and COM/ActiveX

GdPicture.NET 10 Plugins

- NEW Color detection
- Full managed PDF support
- Full annotations support for PDF and images
- OCR
- Forms processing
- JBIG2 encoding
- 1D and 2D barcode reading and writing



Try GdPicture.NET 10 FREE for 30 days

www.componentsource.com/products/gdpicture-net/

For every route, you create an HTML template using the Handlebars template library.

Ember has a top-level template for the entire app. This template gets rendered for every route. **Figure 15** shows the application template for my app. As you can see, the template is basically HTML, placed within a script tag with type="text/x-handlebars." The template contains special Handlebars markup inside double curly braces: {{ }}. This markup serves a similar purpose as the data-bind attribute in Knockout. For example, {{#linkTo}} creates a link to a route.

Now suppose the user navigates to /#/about. This invokes the "about" route. Ember first renders the top-level application template. Then it renders the about template inside the {{outlet}} of the application template. Here's the about template:

```
<script type="text/x-handlebars" data-template-name="about">
  <h2>Movies App</h2>
  <h3>About this app...</h3>
</script>
```

Figure 16 shows how the about template is rendered within the application template.

Because each route has its own URI, the browser history is preserved. The user can navigate with the Back button. The user can also refresh the page without losing the context, or bookmark and reload the same page.

Ember Controllers and Models

In Ember, each route has a model and a controller. The model contains the domain data. The controller acts as a proxy for the model and stores any application state data for the view. (This doesn't exactly match the classic definition of MVC. In some ways, the controller is more like a view model.)

Here's how I defined the movie model:

```
App.Movie = DS.Model.extend({
  Title: DS.attr(),
  Genre: DS.attr(),
  Year: DS.attr(),
  Rating: DS.attr(),
});
```

The controller derives from Ember.ObjectController, as shown in **Figure 17**.

There are some interesting things going on here. First, I didn't specify the model in the controller class. By default, the route automatically sets the model on the controller. Second, the save and cancel functions use the transaction features built into the DS.Model class. To revert edits, just call the rollback function on the model.

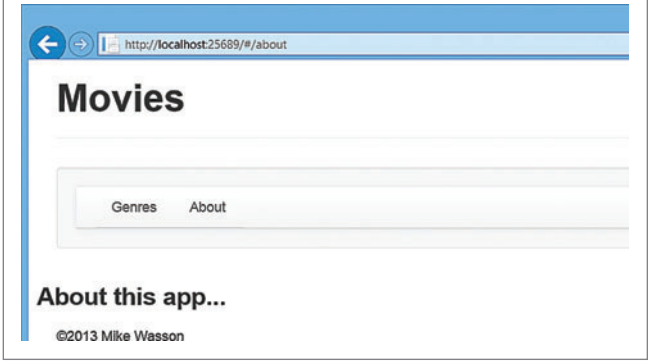


Figure 16 **Rendering the About Template**

Figure 17 **The Movie Controller Derives from Ember.ObjectController**

```
App.MovieController = Ember.ObjectController.extend({
  isEditing: false,
  actions: {
    edit: function () {
      this.set('isEditing', true);
    },
    save: function () {
      this.content.save();
      this.set('isEditing', false);
    },
    cancel: function () {
      this.set('isEditing', false);
      this.content.rollback();
    }
  }
});
```

Figure 18 **Several Routes Can Share the Same Controller**

```
App.GenresMoviesRoute = Ember.Route.extend({
  serialize: function (model) {
    return { genre_name: model.get('name') };
  },
  renderTemplate: function () {
    this.render({ controller: 'movies' });
  },
  afterModel: function (genre) {
    var controller = this.controllerFor('movies');
    var store = controller.store;
    return store.findQuery('movie', { genre: genre.get('name') })
    .then(function (data) {
      controller.set('model', data);
  });
  }
});
```

Ember uses a lot of naming conventions to connect different components. The genres route talks to the GenresController, which renders the genres template. In fact, Ember will automatically create a GenresController object if you don't define one. However, you can override the defaults.

In my app, I configured the genres/movies route to use a different controller by implementing the renderTemplate hook. This way, several routes can share the same controller (see **Figure 18**).

One nice thing about Ember is you can do things with very little code. My sample app is about 110 lines of JavaScript. That's shorter than the Knockout version, and I get browser history for free. On the other hand, Ember is also a highly "opinionated" framework. If you don't write your code the "Ember way," you're likely to hit some roadblocks. When choosing a framework, you should consider whether the feature set and the overall design of the framework match your needs and coding style.

Learn More

In this article, I showed how JavaScript frameworks make it easier to create SPAs. Along the way, I introduced some common features of these libraries, including data binding, routing, and the MVC and MVVM patterns. You can learn more about building SPAs with ASP.NET at asp.net/single-page-application. ■

Mike Wasson *is a programmer-writer at Microsoft. For many years he documented the Win32 multimedia APIs. He currently writes about ASP.NET, focusing on Web API. You can reach him at mwasson@microsoft.com.*

Thanks to the following technical expert for reviewing this article: *Xinyang Qiu (Microsoft)*

BEST SELLER

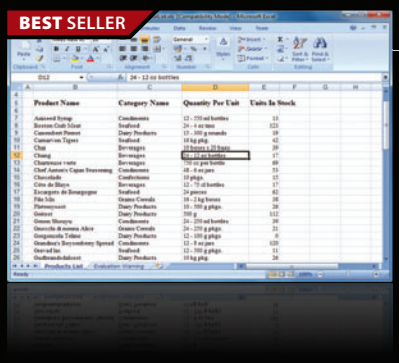
Help & Manual Professional | from $583.10

ec software

Easily create documentation for Windows, the Web and iPad.

- Powerful features in an easy accessible and intuitive user interface
- As easy to use as a word processor, but with all the power of a true WYSIWYG XML editor
- Single source, multi-channel publishing with conditional and customized output features
- Output to HTML, WebHelp, CHM, PDF, ePUB, RTF, e-book or print
- Styles and Templates give you full design control

BEST SELLER



Aspose.Total for .NET | from $2,449.02

ASPOSE
Your File Format Experts

Every Aspose .NET component in one package.

- Programmatically manage popular file formats including Word, Excel, PowerPoint and PDF
- Work with charts, diagrams, images, Project plans, emails, barcodes, OCR, and document management in .NET applications
- Common uses also include mail merging, adding barcodes to documents, building dynamic reports on the fly and extracting text from PDF files

NEW VERSION

GdPicture.NET | from $4,600.56

GdPicture
imaging technologies

All-in-one AnyCPU document-imaging and PDF toolkit for .NET and ActiveX.

- Document viewing, processing, printing, scanning, OMR, OCR, Barcode Recognition
- Annotate image and PDF within your Windows & Web applications
- Read, write and convert vector & raster images in more than 90 formats, including PDF
- Color detection engine for image and PDF compression
- 100% royalty-free and world leading Imaging SDK

BEST SELLER

ComponentOne Studio Enterprise 2013 v2 | from $1,315.60

ComponentOne
a division of GrapeCity

.NET Tools for the Professional Developer: Windows, HTML5/Web, and XAML.

- Hundreds of UI controls for all .NET platforms including grids, charts, reports and schedulers
- Visual Studio 2013 and Windows 8.1 support
- 40+ UI widgets built with HTML5, jQuery, CSS3, and SVG
- New TouchToolkit for touch enabling WinForms apps
- Royalty-free deployment and distribution

We accept purchase orders.
Contact us to apply for a credit account.

US Headquarters
ComponentSource
650 Claremore Prof Way
Suite 100
Woodstock
GA 30188-5188
USA

European Headquarters
ComponentSource
30 Greyfriars Road
Reading
Berkshire
RG1 1PE
United Kingdom

Asia / Pacific Headquarters
ComponentSource
3F Kojimachi Square Bldg
3-3 Kojimachi Chiyoda-ku
Tokyo
Japan
102-0083

Sales Hotline - US & Canada:
(888) 850-9911
www.componentsource.com

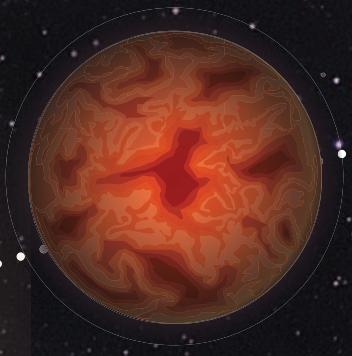
GSA Schedule Contract GS-35F-0188R

Visual Studio® LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

2014 Dates Announced

Much like outer space, the .NET development platform is ever-changing and constantly evolving. Visual Studio Live! exists to help guide you through this universe, featuring code-filled days, networking nights and independent education. Whether you are a .NET developer, software architect or a designer, Visual Studio Live!'s multi-track events include focused, cutting-edge education on the .NET platform.



LAS VEGAS

LIVE! 360 DEV

COMPREHENSIVE TRAINING FOR THE DEVELOPER WORLD

LIVE! Visual Studio | LIVE! SQL Server | LIVE! SharePoint | LIVE! WebDev | LIVE! Modern Apps

Visual Studio Live! Las Vegas

Part of Live! 360 Dev

March 10 – 14
Planet Hollywood
Las Vegas, NV

The Developer World is always expanding; new technologies emerge, current ones evolve and demands on your time grow. Live! 360 Dev offers comprehensive training on the most relevant technologies in your world today.

Visual Studio Live! Chicago

May 5 – 8
Chicago Hilton
Chicago, IL

Secure ASP.NET Web API with Windows Azure AD and Microsoft OWIN Components

Vittorio Bertocci

As the role of the Web API becomes more prominent, so does the need to ensure you can use it confidently in high-value scenarios, where sensitive data and operations might be exposed.

The industry is clearly converging on a solution for securing REST APIs that relies on the OAuth 2.0 standard. In practice, though, this doesn't offer detailed guidance on what should be done at the project level. Furthermore, the existing classes and tools in the Microsoft .NET Framework for securing communications were designed to work with particular application types (postback-based Web UX apps). They aren't a good fit for Web APIs and the multi-client scenarios they enable. As a result, securing a Web API has been a fairly artisanal activity. It isn't necessarily insecure, but varies greatly across solutions and requires far too much custom code.

This article uses a specific authentication flow offered by Windows Azure Active Directory (the code grant via native client) that's currently in Preview.

This article discusses:

- Authentication
- Creating the Web API project
- Registering a native client with the Windows Azure portal
- Creating a simple client project and testing the Web API

Technologies discussed:

Visual Studio 2013, ASP.NET, Windows Azure Active Directory, Microsoft Open Web Interface for .NET (OWIN)

With the release of Visual Studio 2013, you can leave all that behind. This edition introduces innovative ASP.NET tooling and security middleware from the Microsoft Open Web Interface for .NET (OWIN) components that make protecting your Web API straightforward. The new ASP.NET tools and templates let you configure a Web API project to outsource authentication to Windows Azure Active Directory (AD) directly, emitting the necessary code in the local project and in the corresponding entries in Windows Azure AD.

In this article, I'll show you how to take advantage of these new Visual Studio 2013 features to create a simple Web API protected by Windows Azure AD. I'll also show you how to create a test client so you can see the API in action. I'll also take a quick look at what happens behind the scenes, which can serve as a starting point if you want to dig deeper and explore more advanced aspects of this scenario.

Credentials, Please

Authentication boils down to asking a caller, when sending a message to a server, to include some kind of credential that can verify his identity or retrieve his attributes. The server then uses that information for authorization—determining whether access should be granted and in what terms.

Resources often offload most of the authentication functions to an external services provider, commonly known as an authority or an identity provider. These providers take care of heavy-duty tasks such as onboarding users, assigning credentials, handling lifecycle

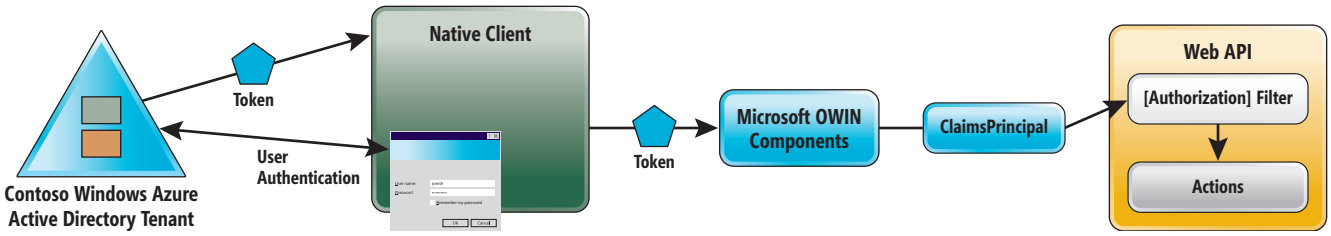


Figure 1 **The Architecture of the End-to-End Solution**

flows (such as password recovery), providing a UI for user authentication, credential verification over multiple protocols, multiple authentication factor management, fraud detection and more.

With those functions out of the way, the only authentication task left is to verify that authentication succeeded at the authority of choice. This typically involves examining a security token, a data fragment issued by an authority to a caller upon successful authentication.

Security tokens are usually crafted according to a specific format, digitally signed by a key that will unequivocally identify the issuing authority, and contain some data that uniquely ties the token to the target resource. When the resource receives a request, it looks for an accompanying token. If it finds one that meets the required validation attributes, the caller is authenticated.

At that level of detail, the pattern is so generic it can describe many different approaches to authentication. I'll apply it to this scenario by assigning the high-level roles to concrete entities.

The Resource The resource will be the ASP.NET Web API 2 project I need to secure. You can apply authentication requirements with finer granularity. For example, you can define a subset of actions to secure and leave others to accept anonymous callers.

The Authority I'll configure the Web API to offload its authentication needs to Windows Azure AD, a Platform as a Service (PaaS) available to every Windows Azure subscriber. Windows Azure AD is specifically designed to support cloud-based application workloads. The service stores information about users (attributes and credentials) and organizational structure. You can synchronize its data with your Windows Server Active Directory, if you so choose, or live exclusively in the cloud with no need for an on-premises infrastructure.

Nearly every online Microsoft service (Office 365, Intune and Windows Azure) takes advantage of Windows Azure AD for its authentication and directory needs. Thanks to open standards and support for common protocols, you can connect to Windows Azure AD from virtually any application (Web UX, Web API, native client, server to server, and so on) and platform. I'll demonstrate how to register your apps in Windows Azure AD and take advantage of its OAuth 2.0 endpoints.

Token Format and Validation The OAuth 2.0 specification doesn't mandate any particular token format, but the JSON Web Token (JWT) format (bit.ly/14EhlE8) for REST scenarios has become the de facto standard. Both Windows Azure AD and the Microsoft OWIN components support the JWT format in OAuth 2.0 flows. I mention this mostly to provide some context. The mechanics of JWT acquisition and validation are all taken care of by the middleware, and the token format is transparent to the application code.

The Client When a resource relies on an authority to handle authentication, it's effectively decoupled from the client. How the user (and the client application) obtains a token becomes a matter between the user and the authority. That's great for code maintainability, but if you want to see your API in action, you still need to set up a client. You'll learn how to register a native client for the Web API in Windows Azure AD, and how to use the Windows Azure AD Authentication Library (ADAL) to enable a .NET rich client application to authenticate users against Windows Azure AD and obtain tokens to secure calls to the Web API.

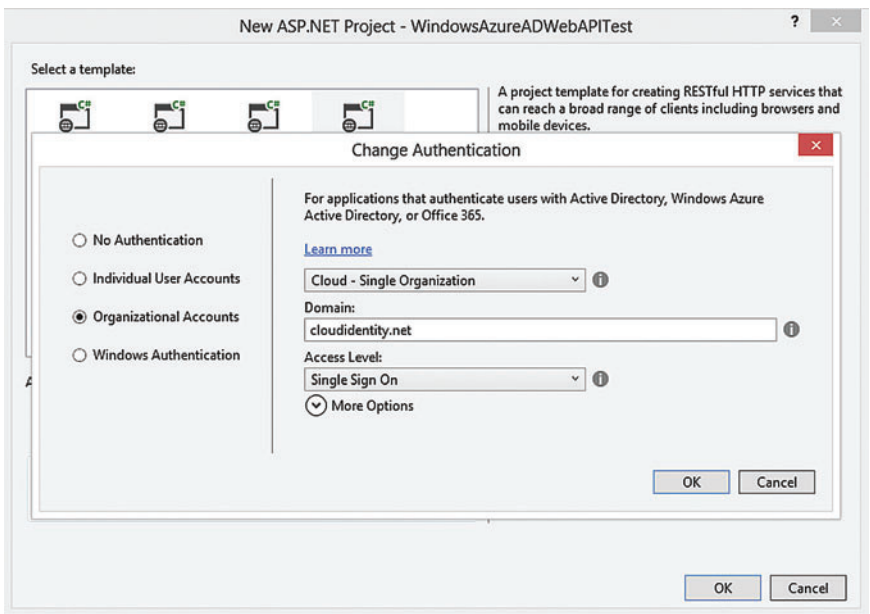


Figure 2 **The Organizational Accounts Authentication Dialog**

Figure 1 shows the various elements of the solution I'm going to build. Don't worry if you don't understand some of the labels at this point: Everything will be introduced as I walk through the development of the solution.

Creating the Web API Project

To create the Web API project, use the new ASP.NET tools and templates in Visual Studio 2013. Open Visual Studio and create a new ASP.NET Web Application project. In the new project dialog, select the Web API template. Click on the Change Authentication button.

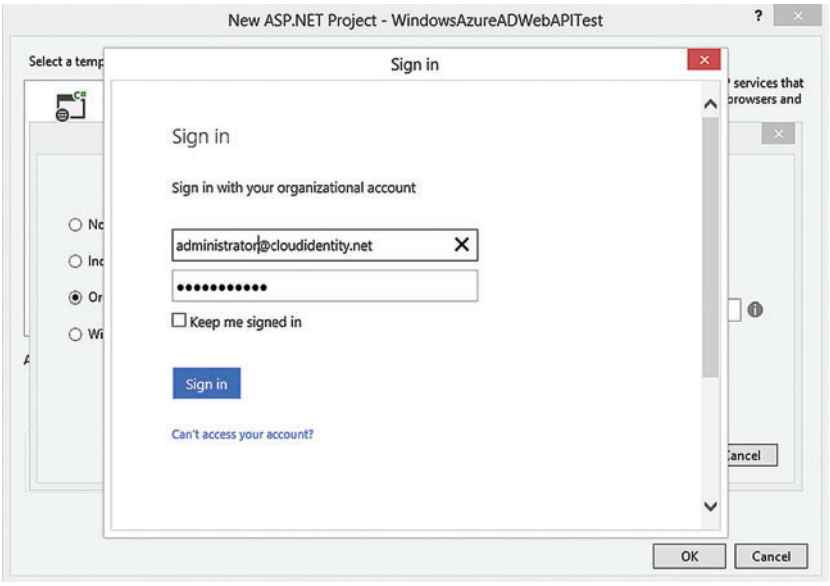


Figure 3 **Account Sign in Authentication Prompt**

You'll be presented with the authentication styles you can choose out of the box for a Web API, as shown in **Figure 2**. Choose Organizational Accounts, which lets you use Windows Azure AD as the authority. (For more information on all of the options, see bit.ly/1bhWngl.) The goal here is to gather information about the characteristics of your Web API that are important from an identity management perspective, and to determine which Windows Azure AD instance (commonly known as a "tenant") you should configure to handle authentication.

The first dropdown determines whether just one Windows Azure AD tenant (typical of a line-of-business application) or multiple Windows Azure AD tenants (as you'd expect from a Software as a Service [SaaS] application) should use the application. You'd normally select Single Organization if your app will be used by the employees in your company, or Multiple Organizations if the app will be accessed by users from many companies. That said, those two alternatives are currently available only for ASP.NET WebForms and ASP.NET MVC apps. Currently, the Web API project template supports only the Single Organization option.

The Domain textbox identifies which Windows Azure AD tenant should register your application. It's typically the enterprise directory associated with your Windows Azure subscription, but any directory for which you have administrative credentials will do (more on that later). At creation time, every Windows Azure AD tenant has one, associated three-level domain in the form yourorganization.onmicrosoft.com. Generally, you'll associate the tenant to one or more domains you already own. In the example in **Figure 2**, I use my own domain, cloudidentity.net.

The Access Level dropdown specifies which access rights the application should have against the directory. The default value, Single Sign On, lets the directory issue tokens for your application. This is a simple acknowledgement that the app is registered. The authority won't issue tokens for unregistered apps, even upon successful authentication.

Other access levels include "Read directory data" and "Read and write directory data," which respectively enable the application to query the directory and modify its content. It does this via the Graph API, a REST-based programmatic interface designed to let apps from any platform with an HTTP stack gain delegated access to the directory. I'll stay with the default Single Sign On access level and not demonstrate the Graph in the Web API project. That is, however, an extremely powerful feature of Windows Azure AD. For more information on the Windows Azure AD Graph API, see bit.ly/1aByRLS.

After you enter the domain associated to your Windows Azure AD tenant, click OK to generate the project. The tool will perform two tasks:

1. It will reach out to the selected Windows Azure AD tenant and add an entry describing the application being created.
2. It will emit code for a Web API project, add the necessary security middleware from the Microsoft OWIN components to handle Windows Azure AD authentication and generate the necessary initialization code to validate incoming tokens according to the Windows Azure AD tenant of choice.

The first task is performed via the Graph API (Visual Studio is itself a client of the Graph API). In order to add the entry describing the application you're creating to the directory, it needs to obtain a

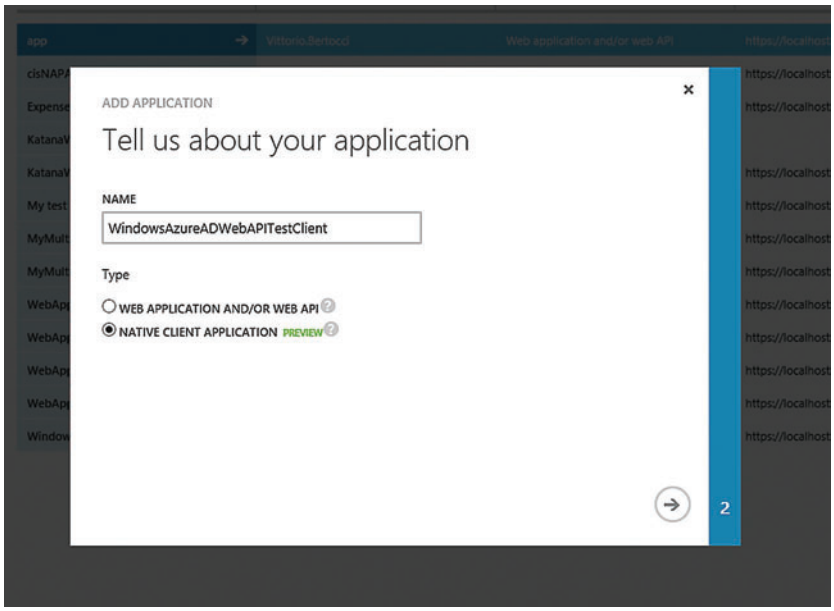


Figure 4 **The First Step of the Add Application Wizard on the Windows Azure Active Directory Portal**

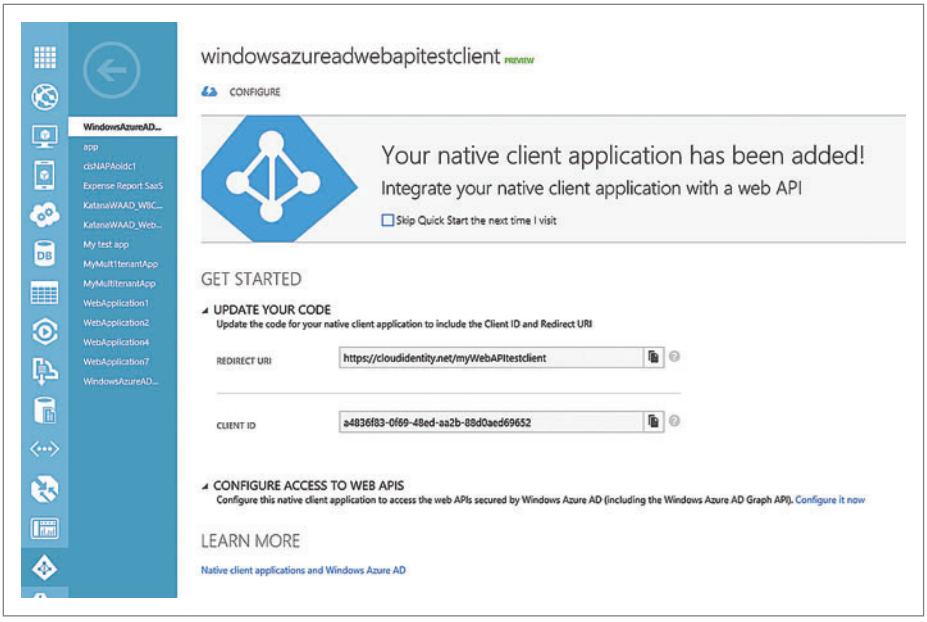


Figure 5 **The Quick Start Page**

token from the directory proving you have the necessary privileges to write in the directory. That's why, once you click OK, Visual Studio presents you with the authentication prompt you see in **Figure 3**.

All you need to do is enter administrator credentials for the directory. The tool verifies you have the necessary privileges to perform the operations required. When you click OK, Visual Studio contacts Windows Azure AD and creates the project files.

You now have a fully configured Web API, ready to enforce that only callers from the specified Windows Azure AD tenant gain access. Let's take a closer look.

Go to the Solution Explorer pane where you'll see in the root a file called Startup.cs. If you read Howard Dierking's "Getting Started with the Katana Project" from last month's issue (msdn.microsoft.com/magazine/dn451439), you already know the class in this file is called at application startup. In this case, implementation is simple:

```
public partial class Startup
{
  public void Configuration(IAppBuilder app)
  {
    ConfigureAuth(app);
  }
}
```

Because it's standard, you'd expect to find the ConfigureAuth method defined in some class under the App_Start solution folder. It's actually the Startup.Auth.cs file. Here's what it looks like:

```
public partial class Startup
{
  public void ConfigureAuth(IAppBuilder app)
  {
    app.UseWindowsAzureActiveDirectoryBearerAuthentication(
      new WindowsAzureActiveDirectoryBearerAuthenticationOptions
      {
        Audience = ConfigurationManager.AppSettings["ida:Audience"],
        Tenant = ConfigurationManager.AppSettings["ida:Tenant"]
      });
  }
}
```

The app.Use* naming convention suggests the method adds a middleware implementation to the OWIN pipeline. In this case, the added middleware inspects the incoming request to see if the HTTP header Authorization contains a security token. This is how you secure a request according to the OAuth 2.0 bearer token specification (see bit.ly/W4OqA3). If it finds a token, it's validated via a number of standard checks: Was the token issued by the intended authority? Has it been tampered in transit? Is it expired?

If the token looks good, the middleware projects its content in a principal, assigns the principal to the current user and cedes control to the next element in the pipeline. If the token doesn't pass the checks, the middleware sends back the appropriate error code.

If there's no token, the middleware simply lets the call go through without creating a principal. The authorization logic, typically in the form of authorization filters, uses the presence or absence of a principal (and its content) to decide whether the request should be served or access denied.

The single parameter passed to the middleware, WindowsAzureActiveDirectoryBearerAuthenticationOptions, supplies the settings for determining a token's validity. It captures the raw values during project creation and stores them in the web.config file. The Audience value is the identifier by which the Web API is known to Windows Azure AD. Any tokens carrying a different Audience are meant for another resource and should be rejected.

The Tenant property indicates the Windows Azure AD tenant used to outsource authentication. The middleware uses that information to access the tenant and read all the other properties (such as which key should be used to verify the token's signatures) that determine the validity of a token.

Those few auto-generated lines of code are all that's needed to authenticate callers with Windows Azure AD. The only thing left to do on the Web API project is decorate with [Authorize] the methods you want to protect. Add it to all the methods in ValuesController. (For the sake of simplicity, I'm working with the default controllers that come with the template.)

Now, how do you verify that the Web API behaves as you intended? The simplest way is to create a test client.

Register a Native Client

Just as Windows Azure AD won't issue tokens for Web APIs that haven't been registered, Windows Azure requires all clients requesting tokens be registered as well. In order to get a token for a particular Web API, a registered client must also have been explicitly given access to that Web API in Windows Azure AD. Such design-time settings must be in place even before any user attempts authentication. Now I'll show you how to use the Windows Azure portal to register a client application and create the permission that ties it to the Web API you created.

Start by accessing the Windows Azure portal. I'm going to authenticate to the same account I used earlier. To save time, I'll navigate

NEVRON
visualize your success

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

Nevron Data Visualization

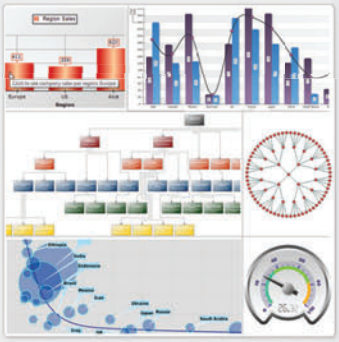
The leading data visualization components for a wide range of .NET platforms.14+ years of refinement, complete feature sets, highly customizable design and great support.

Developers

NEVRON Open VISION

Nevron Open Visions is a Cross - Platform Presentation Layer based on .NET. It aims to run on all major operating systems and integrate with already existing .NET based presentation layers.

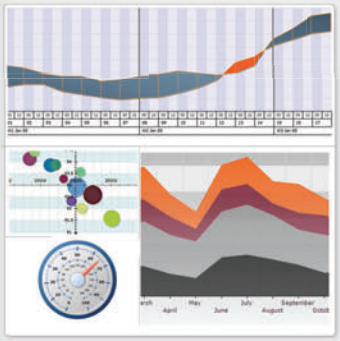
NEVRON VISION for .NET



Nevron Vision for .NET includes chart, gauge, diagram, map and user interfaces components that help you create enterprise-grade digital dashboards, scorecards, diagrams, maps, MMI interfaces and much more.

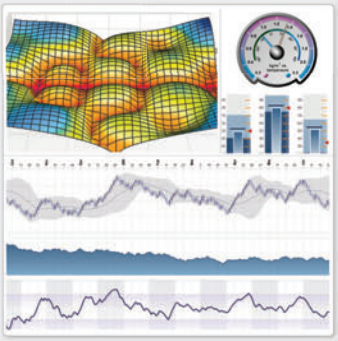
IT Professionals

NEVRON VISION for SharePoint



Nevron Vision for SharePoint combines the leading Chart and Gauge web parts for SharePoint 2007, 2010 and 2013. With this suite you can easily convert your SharePoint pages into interactive dashboards and reports.

NEVRON VISION for SSRS



Nevron Vision for SSRS presents the leading data visualization report items for SSRS 2005, 2008 and 2012. The Chart and Gauge help you deliver deep data insights with highly customizable engaging looks.

Nevron components integrate seamlessly in Web and Desktop .NET applications, SQL Server Reporting Services reports and SharePoint portals. All data visualization tools deliver an unmatched set of enterprise- grade features which makes Nevron the trusted vendor for many Fortune 500 companies.

Download you free evaluation copy from **www.nevron.com** today.

directly to the Windows Azure portal for my Windows Azure AD tenant. You obtain that URL by adding your tenant domain to the usual portal address, in my case http://manage.windowsazure.com/cloudidentity.net.

Navigating to the tenant-specific URL is quicker than hunting for the "Sign in with your organizational account" box from the general login page. Note that if the Windows Azure AD with which you're working isn't an admin or co-admin of your Windows Azure subscription, you'll want to sign in to the portal using your usual credentials (a Microsoft Account or other organizational entity).

Once the portal loads, select the Active Directory icon from the list of available services, click on your directory and then on the Applications tab. You'll see a list of all the applications you created, including the entry for the new Web API project. To create a new entry, click on the Add button from the command bar on the bottom of the page. You'll see the dialog shown in **Figure 4**. Technically, you could define just about any kind of app and that would be a valid client for your Web API. Choose a native client application and move to the next screen.

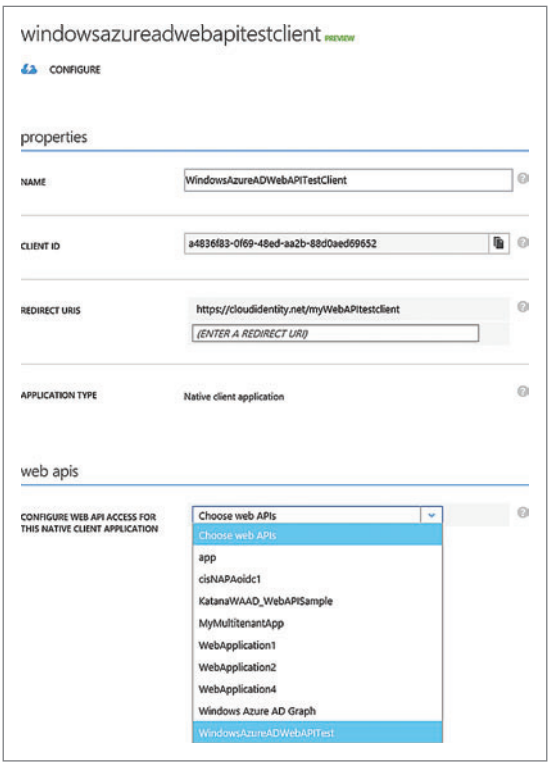


Figure 6 **The Native Client Application Properties Page on the Windows Azure Active Directory Portal**

The next and last screen asks you to enter a redirect URI for the application. This URI is simply an identifier used during the OAuth 2.0 token acquisition flow to signal to the caller the interactive portion of the authentication process is over. The rest of the token acquisition process will proceed without user input. Depending on the platform on which you develop the native client, you might have to deal with different constraints. For example, a Windows Store app would require you to use the ms-app:// protocol schema if you want to use certain features (see details at bit.ly/13KrM6i).

In this case, I'll write a classic .NET desktop app, which happens to be quite forgiving. Any valid URI will do. I used https://cloudidentity.net/myWebAPItestclient to ensure I'll remember what this client is for.

As soon as I finalize the app entry, I see the page shown in **Figure 5**.

The Update Your Code section provides information you'll need when you write the client application. This includes the settings for the redirect URI and the client ID (a simple identifier), which you need when crafting a token request to the authority.

The Configure Access to Web APIs section offers a link to the area of the portal where you can specify the API to which the client should have access. If you follow the link, you'll land on the application properties page shown in **Figure 6**. At the bottom you'll see a dropdown that lets you specify the Web API you want your client to access. Notice the dropdown lists both the apps you defined and built-in APIs, specifically the Windows Azure AD Graph API.

Choose the Web API project entry and hit Save. With that, everything is ready in Windows Azure AD to have a client obtain tokens for your service. Leave the browser open to that page, as you'll need some of the data it will display.

Creating a Simple Client Project and Testing the Web API

You're finally ready to create a test client and give the authenticated Web API a spin. You can create just about any client type, on any platform that offers an HTTP stack. To simplify the task of getting and maintaining tokens, Microsoft provides the ADAL, which makes it easy to authenticate against Active Directory (both Windows Azure and Windows Server) without having to become an expert in authentication protocols.

The library for the Microsoft .NET Framework has been released and is in developer preview for Windows Store. (For a tutorial on how to implement this scenario with a Windows Store app, see bit.ly/17YtYVg.) Eventually, there will be versions targeting all major client platforms.

Because the .NET version is already generally available, I'll use it here. Apart from syntactic differences across different stacks, what you'll learn here will be readily applicable to other platforms and application types. If you can't wait, keep in mind that all communications in the scenario described here follow open standards and are documented in detail. You can code the token request logic pretty easily. For an example using Windows Phone 8, see bit.ly/YatATk.

The project is for a new Windows Presentation Foundation (WPF) application within the same solution, but you can choose

Figure 7 **Code to Add to the Click Event Handler**

```
private async void btnCall_Click(object sender, RoutedEventArgs e)
{
  // Get token
  AuthenticationContext ac = new AuthenticationContext(
    "https://login.windows.net/cloudidentity.net");
  AuthenticationResult ar =
    ac.AcquireToken("https://cloudidentity.net/WindowsAzureADWebAPITest",
    "a4836f83-0f69-48ed-aa2b-88d0aed69652",
    new Uri("https://cloudidentity.net/myWebAPItestclient"));
  // Call Web API
  string authHeader = ar.CreateAuthorizationHeader();
  HttpClient client = new HttpClient();
  HttpRequestMessage request = new HttpRequestMessage(
    HttpMethod.Get, "https://localhost:44353/api/Values");
  request.Headers.TryAddWithoutValidation("Authorization", authHeader);
  HttpResponseMessage response = await client.SendAsync(request);
  string responseString = await response.Content.ReadAsStringAsync();
  MessageBox.Show(responseString);
}
```

ALEXSYS TEAM®

WORKFLOW APPLICATIONS | HELP DESK | BUG TRACKING

MADE EASY!

Alexsys Team® offers ***flexible task management*** software for Windows, Web, and Mobile Devices. Track whatever you need to get the job done - anywhere, anytime on practically any device!

Thousands are using Alexsys Team® to create workflow solutions and web apps - without coding!
Fortune 500 companies, state, local, DOD, and other federal agencies use Team to manage their tasks. Easily tailor Team to meet your exact requirements - even if they change daily, weekly, or even every minute.

Alexsys Team® Features Include:

- Form and Database customization
- Custom workflows
- Role-based security
- Adaptive Rules Engine
- DOD CAC card support
- Time recording
- Automated Escalations, Notifications, and SOAP Messaging
- Supports MS-SQL, MySQL, and Orcale databases

Our renowned tech support team is here to make you and your team a success. Don't have enough resources? Our professional services staff has helped companies large and small use Alexsys Team® at a fraction of the cost of those big consulting firms.

Find out yourself:
Free Trial and Single User FreePack™ available at Alexcorp.com

FIND OUT MORE!

1-888-880-ALEX (2539)
ALEXCORP.COM

any project type meant to run interactively with a user. After creating the project, I'll add a button and a click event handler to trigger the Web API invocation logic.

The ADAL is distributed as a NuGet package. To add it to the client project, simply open the Package Manager Console from the Tools menu in Visual Studio and type Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 1.0.0. Now add the code in **Figure 7** to the click event handler.

Don't worry if the code looks intimidating. It will all be clear in a minute.

The first line initializes a new AuthenticationContext. In the app's code, an AuthenticationContext instance represents the authority with which to work. In this case, the authority is my Windows Azure AD tenant represented by the URI https://login.windows.net/[mydomain]. You'd use the same logic to connect to an on-premises directory by passing the address of its Active Directory Federation Services (AD FS) server (see bit.ly/1d553F0 for details).

The second line asks the AuthenticationContext for a token. There are many ways of crafting a token request—every scenario and application type calls for different parameters. In this case, I want to specify the resource for which I want a token is my Web API. So I pass the identifier of its Windows Azure AD entry. This is the same value used as the audience in the Web API project. Then, because it's the native client requesting the token, I need to pass in the client ID and the redirect URI of my client from the app configuration page I left open in the browser.

That's all I need to do to obtain a token. The rest of the code puts that token in the correct HTTP header according to the OAuth 2.0 specification, and performs the actual call.

Now give the solution a spin. After changing the solution to start all projects at once, press F5. As soon as the execution hits the call to AcquireToken, you'll get the authentication dialog shown in **Figure 8**. ADAL takes care of contacting the right endpoint and rendering the authentication experience provided by the server in a pop-up dialog without requiring you to write any UI code.

When I provide the credentials of any valid user from my directory tenant, I get a token back. The subsequent code presents the token to the Web API in the request headers. The security middleware validates it. Because all validation parameters are a match, it sends back HTTP status code 200 with the results, successfully concluding the proof of concept for this scenario.

Just for kicks, click the button again. You'll see this time you get a token back right away, without being prompted. That's because ADAL has a built-in token cache that keeps track of the tokens. It even takes care of silently refreshing expired tokens whenever possible.

Where to Go from Here

I've barely scratched the surface of what you can accomplish with Web API, the Microsoft OWIN components, Windows Azure AD and ADAL. The tokens issued by Windows Azure AD aren't simply proofs of authentication. They carry rich user information you can easily access from the user's principal and use for sophisticated authorization logic. The authentication process can go far beyond the username and password shown here.

From multiple authentication factors to seamless single sign-on in federated scenarios, the possibilities are endless. Integrating with the Graph API can give you access to a treasure trove of features. You can query the directory for information that goes beyond the currently authenticated user, from simple people picker functionality to advanced organizational structure crawling.

Those features can add to the functionality of cloud-based Web APIs. Until now, you had to run them on-premises behind the corporate firewall. You can also use similar code with Windows Server Active Directory, in one of the clearest examples of cloud and on-premises symmetry of capabilities.

Of course, you can also publish the Web API to Windows Azure without changing a single line of code—the authentication features will keep working. The only thing you'd need to change is on the client project. The URL of the service will change according to the new application location. However, the logic for acquiring a token can remain exactly the same, because the resource identifier doesn't have to be tied to the physical address of the resource. ■

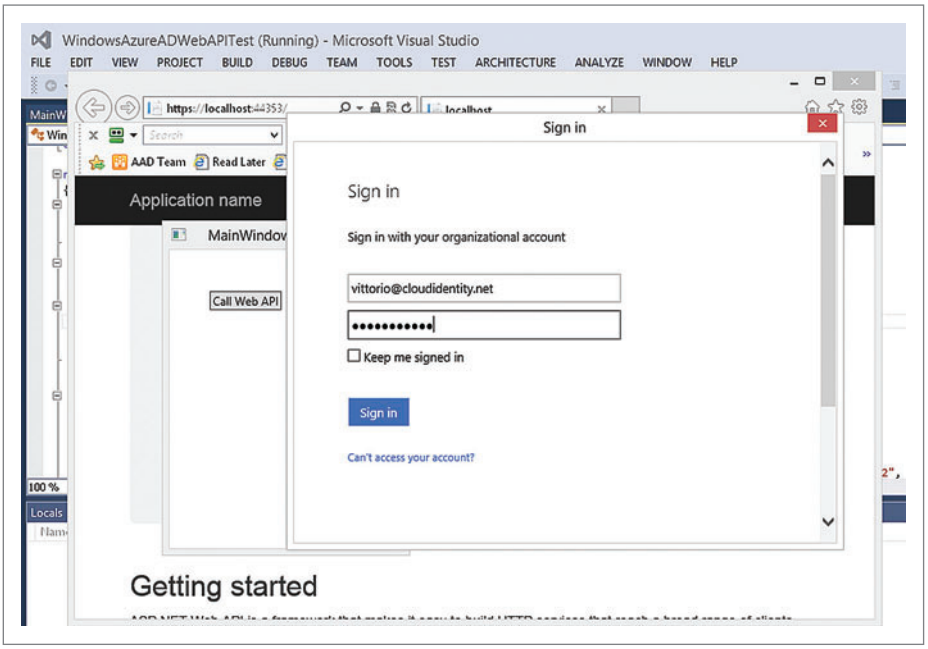


Figure 8 **The Active Directory Authentication Library Authentication Dialog**

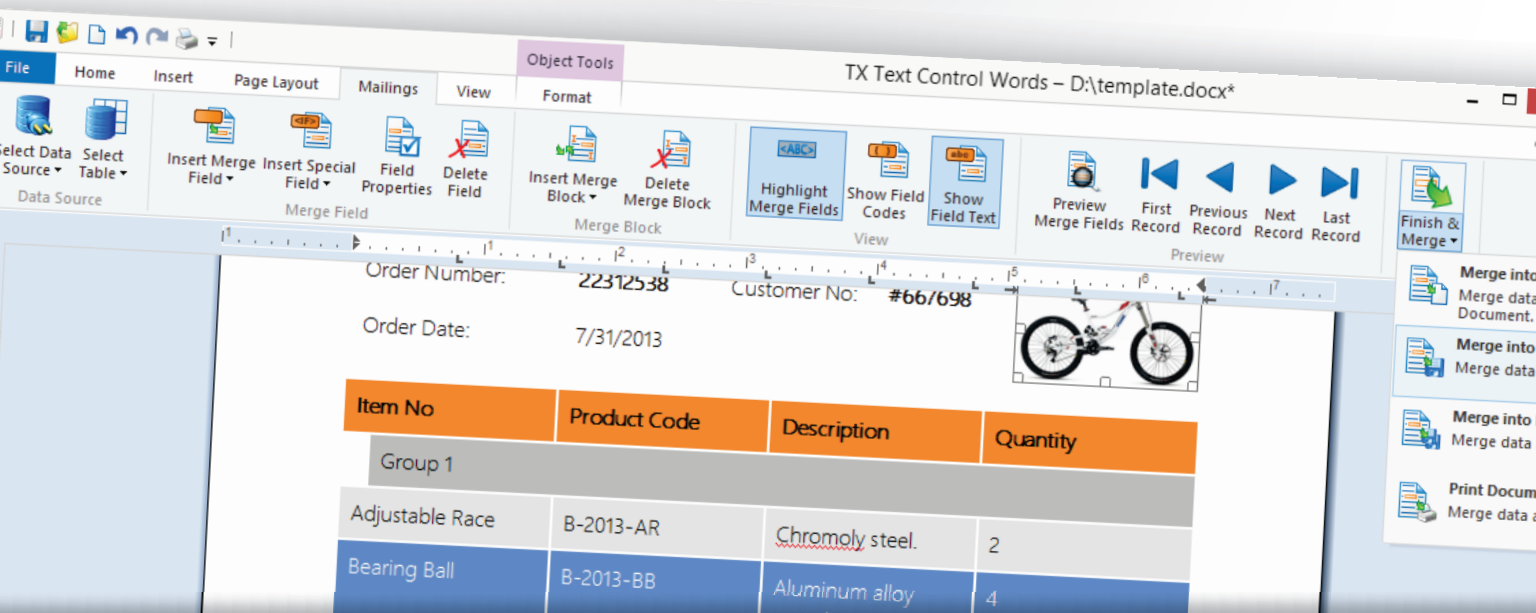
VITTORIO BERTOCCI *is a principal program manager on the Windows Azure AD team, where he looks after developer experience. Bertocci is well known in the developer community for his decade-long evangelism activities about identity and development via his books, his sessions at major conferences, his blog (cloudidentity.com) and Twitter feed (twitter.com/vibronet).*

THANKS to the following Microsoft technical experts for reviewing this article: *Howard Dierking and Daniel Roth*

TX TEXT CONTROL®
word processing components

Creating a report is as easy as writing a letter

- Reuse MS Word documents as your reporting templates
- Create encrypted and print-ready Adobe PDF and PDF/A
- Royalty-free WYSIWYG template designer
- Powerful and dynamic 2D/3D charting support
- Easy database connections and master-detail nested blocks
- 1D/2D barcode support including QRCode, IntelligentMail, EAN



www.textcontrol.com/reporting

txtextcontrol

US: +1 855-533-TEXT
EU: +49 421 427 067-10

Visual Studio Microsoft Partner

Reporting | Rich Text Editing | Spell Checking | Barcodes | PDF Reflow

Writing a Testable Presentation Layer with MVVM

Brent Edwards

With traditional applications from the Windows Forms days, the standard practice for testing was to lay out a view, write code in the view's codebehind, then run the app as a test. Fortunately, things have evolved a bit since then.

The advent of Windows Presentation Foundation (WPF) brought the concept of data binding to a whole new level. It has allowed a new design pattern called Model-View-ViewModel (MVVM) to evolve. MVVM lets you separate presentation logic from the actual presentation. Basically, it means for the most part you can avoid writing code in the view's codebehind.

This is a major improvement for those interested in developing testable applications. Now instead of having your presentation logic attached to the view's codebehind, which has its own lifecycle to complicate testing, you can use a plain old CLR object (POCO). View models don't have the lifecycle constraints that a view does. You can just instantiate a view model in a unit test and test away.

In this article, I'll take a look at how to approach writing a testable presentation layer for applications using MVVM. To help illustrate my approach, I'll include sample code from an open source framework I wrote, called Charmed, and an accompanying sample app, called Charmed Reader. The framework and sample apps are available on GitHub at github.com/brentedwards/Charmed.

I introduced the Charmed framework in my July 2013 article (msdn.microsoft.com/magazine/dn296512) as a Windows 8 framework and sample application. Then in my September 2013 article (msdn.microsoft.com/magazine/dn385706), I discussed making it cross-platform as a Windows 8 and Windows Phone 8 framework and sample app. In both of those articles, I talked about decisions I made to keep the app testable. Now, I'll revisit those decisions and show how I actually go about testing the app. This article uses Windows 8 and Windows Phone 8 code for the examples, but you can apply the concepts and techniques to any type of application.

This article discusses:

- The Charmed Reader sample app
- The three phases of unit testing
- Planning for testability
- Making navigation testable
- Creating testable secondary tiles

Technologies discussed:

Windows 8, Windows Phone 8, Visual Studio 2012, MSTest

Code download available at:

archive.msdn.microsoft.com/mag201311MVVM

About the Sample App

The sample app that illustrates how I approach writing a testable presentation layer is called Charmed Reader. Charmed Reader is a simple blog reader app that works on both Windows 8 and Windows Phone 8. It has the minimum functionality needed to illustrate the key points I want to cover. It's cross-platform and works mostly the same on both platforms, with the exception that the Windows 8 app leverages some Windows 8-specific functionality. While the app is basic, there's enough functionality for unit testing.

What Is Unit Testing?

The idea behind unit testing is to take discrete chunks of code (units) and write test methods that use the code in an expected way, then test to see they get the expected results. This test code is run using some sort of test harness framework. There are several test harness frameworks that work with Visual Studio 2012. In the sample code, I use MSTest, which is built into Visual Studio 2012 (and earlier). The goal is to have a single unit test method target a specific scenario. Sometimes it takes several unit test methods to cover all the scenarios you expect your method or property to accomodate.

Figure 1 **RssFeedServiceMock**

```
public class RssFeedServiceMock : IRssFeedService
{
  public Func<List<FeedData>> GetFeedsAsyncDelegate { get; set; }
  public Task<List<FeedData>> GetFeedsAsync()
  {
    if (this.GetFeedsAsyncDelegate != null)
    {
      return Task.FromResult<List<FeedData>>(this.GetFeedsAsyncDelegate());
    }
    else
    {
      return Task.FromResult<List<FeedData>>(null);
    }
  }
}
```

A unit test method should follow a consistent format to make it easier for other developers to understand. The following format is generally considered a best practice:

1. Arrange
2. Act
3. Assert

First, there may be some setup code you need to write to create an instance of the class under test as well as any dependencies it might have. This is the Arrange section of the unit test.

After the unit test is done setting the stage for the actual test, you can execute the method or property in question. This is the Act section of the test. You can execute the method or property in question with parameters (when applicable) set up during the Arrange section.

Finally, when you've executed the method or property in question, the test needs to verify the method or property did exactly what it was supposed to do. This is the Assert section of the test. During the assert phase, assert methods are called to compare actual results with expected results. If the actual results are as expected, the unit test passes. If not, the test fails.

Following this best practice format, my tests usually look something like the following:

```
[TestMethod]
public void SomeTestMethod()
{
  // Arrange
  // *Insert code to set up test

  // Act
  // *Insert code to call the method or property under test

  // Assert
  // *Insert code to verify the test completed as expected
}
```

Some people use this format without including comments to call out the different sections of the test (Arrange/Act/Assert). I prefer to have comments separating the three sections just to make sure I don't lose track of what a test is actually acting on or when I'm just setting up.

An added benefit of having a comprehensive suite of well-written unit tests is that they act as living documentation of the app. New developers viewing your code will be able to see how you expected the code to be used by looking at the different scenarios the unit tests cover.

Planning for Testability

If you want to write a testable application, it really helps to plan ahead. You'll want to design your application's architecture so it's conducive to unit testing. Static methods, sealed classes, database access, and Web service calls all can make your app difficult or impossible to unit test. However, with some planning, you can minimize the impact they have on your application.

The Charmed Reader app is all about reading blog posts. Downloading these blog posts involves Web access to RSS feeds, and it can be rather difficult to unit test that functionality. First, you should be able to run unit tests quickly and in a disconnected state. Relying on Web access in a unit test potentially violates these principles.

Moreover, a unit test should be repeatable. Because blogs are usually updated regularly, it might become impossible to get the same data downloaded over time. I knew in advance that unit testing the functionality that loads blog posts would be impossible if I didn't plan ahead.

Here's what I knew needed to happen:

1. The MainViewModel needed to load all the blog posts the user wants to read at once.
2. Those blog posts needed to be downloaded from the various RSS feeds the user has saved.
3. Once downloaded, the blog posts needed to be parsed into data transfer objects (DTOs) and made available to the view.

If I put the code to download the RSS feeds in MainViewModel, it would suddenly be responsible for more than just loading data and letting the view databind to it for display. MainViewModel would then be responsible for making Web requests and parsing XML data. What I really want is to have MainViewModel call out to a helper to make the Web request and parse the XML data. MainViewModel should then be given instances of objects that represent the blog posts to be displayed. These are called DTOs.

Knowing this, I can abstract the RSS feed loading and parsing into a helper object that MainViewModel can call. This isn't the end of the story, however. If I just create a helper class that does the RSS feed data work, any unit test I write for MainViewModel around this functionality would also end up calling that helper class to do the Web access. As I mentioned before, that goes against the goal of unit testing. So, I need to take it a step further.

If I create an interface for the RSS feed data-loading functionality, I can have my view model work with the interface instead of a concrete class. Then I can provide different implementations of the interface for when I'm running unit tests instead of running the app. This is the concept behind mocking. When I run the app for real, I want the real object that loads the real RSS feed data. When

Figure 2 **Testing Feed Loading Functionality**

```
[TestMethod]
public void FeedData()
{
  // Arrange
  var viewModel = GetViewModel();

  var expectedFeedData = new List<FeedData>();

  this.RssFeedService.GetFeedsAsyncDelegate = () =>
    {
      return expectedFeedData;
    };

  // Act
  var actualFeedData = viewModel.FeedData;

  // Assert
  Assert.AreSame(expectedFeedData, actualFeedData);
}
```

Figure 3 **Mock for INavigator**

```
public class NavigatorMock : INavigator
{
  public bool CanGoBack { get; set; }

  public Action GoBackDelegate { get; set; }
  public void GoBack()
  {
    if (this.GoBackDelegate != null)
    {
      this.GoBackDelegate();
    }
  }

  public Action<Type, object> NavigateToViewModelDelegate { get; set; }
  public void NavigateToViewModel<TViewModel>(object parameter = null)
  {
    if (this.NavigateToViewModelDelegate != null)
    {
      this.NavigateToViewModelDelegate(typeof(TViewModel), parameter);
    }
  }

#if WINDOWS_PHONE
  public Action RemoveBackEntryDelegate { get; set; }
  public void RemoveBackEntry()
  {
    if (this.RemoveBackEntryDelegate != null)
    {
      this.RemoveBackEntryDelegate();
    }
  }
#endif // WINDOWS_PHONE
}
```

Figure 4 **Testing Navigation Using the Mock Navigator**

```
[TestMethod]
public void ViewFeed()
{
  // Arrange
  var viewModel = this.GetViewModel();

  var expectedFeedItem = new FeedItem();

  Type actualViewModelType = null;
  FeedItem actualFeedItem = null;
  this.Navigator.NavigateToViewModelDelegate = (viewModelType, parameter) =>
    {
      actualViewModelType = viewModelType;
      actualFeedItem = parameter as FeedItem;
    };

  // Act
  viewModel.ViewFeed(expectedFeedItem);

  // Assert
  Assert.AreSame(expectedFeedItem, actualFeedItem, "FeedItem");
  Assert.AreEqual(typeof(FeedItemViewModel), actualViewModelType, "ViewModel Type");
}
```

I run the unit tests, I want a mock object that just pretends to load the RSS data, but never actually goes out to the Web. The mock object can create consistent data that is repeatable and never changes. Then, my unit tests can know exactly what to expect every time.

With this in mind, my interface for loading the blog posts looks like this:

```
public interface IRssFeedService
{
  Task<List<FeedData>> GetFeedsAsync();
}
```

There's only one method, GetFeedsAsync, which MainViewModel can use to load the blog post data. MainViewModel doesn't need to care how IRssFeedService loads the data or how it parses the data. All MainViewModel needs to care about is that calling GetFeedsAsync will asynchronously return blog post data. This is especially important given the cross-platform nature of the app.

Windows 8 and Windows Phone 8 have different ways of downloading and parsing the RSS feed data. By making the IRssFeedService interface and having MainViewModel interact with it, instead of directly downloading blog feeds, I avoid forcing MainViewModel to have multiple implementations of the same functionality.

Using dependency injection, I can make sure to give MainViewModel the correct instance of IRssFeedService at the right time. As I mentioned, I'll provide a mock instance of IRssFeedService during the unit tests. One interesting thing about using Windows 8 and Windows Phone 8 code as the foundation for a unit test discussion is there aren't any real dynamic mocking frameworks currently available for those platforms. Because mocking is a big part of how I unit test my code, I had to come up with my own simple way to create mocks. The resulting RssFeedServiceMock is shown in **Figure 1**.

Basically, I want to be able to provide a delegate that can set up how the data is loaded. If you aren't developing for Windows 8 or Windows Phone 8, there's a pretty good chance you can use a dynamic mocking framework such as Moq, Rhino Mocks or NSubstitute. Whether you roll your own mocks or use a dynamic mocking framework, the same principles apply.

Now that I have the IRssFeedService interface created and injected into the MainViewModel, the MainViewModel calling GetFeedsAsync on the IRssFeedService interface, and the RssFeedServiceMock created and ready to use, it's time to unit test the MainViewModel's interaction with IRssFeedService. The important aspects I want to test for in this interaction are that MainViewModel correctly calls GetFeedsAsync and the feed data that's returned is the same feed data that MainViewModel makes available via the FeedData property. The unit test in **Figure 2** verifies this for me.

Whenever I'm unit testing a view model (or any other object, for that matter), I like to have a helper method that gives me the actual instance of the view model to test. View models are likely to change over time, which may involve different things being injected into the view model, which means different constructor parameters. If I create a new instance of the view model in all my unit tests and then change the constructor's signature, I have to change a whole bunch of unit tests along with it. However, if I create a helper method to

Figure 5 **Using ISecondaryPinner**

```
public async Task Pin(Windows.UI.Xaml.FrameworkElement anchorElement)
{
  // Pin the feed item, then save it locally to make sure it's still available
  // when they return.
  var tileInfo = new TileInfo(
    this.FormatSecondaryTileId(),
    this.FeedItem.Title,
    this.FeedItem.Title,
    Windows.UI.StartScreen.TileOptions.ShowNameOnLogo |
      Windows.UI.StartScreen.TileOptions.ShowNameOnWideLogo,
    new Uri("ms-appx:///Assets/Logo.png"),
    new Uri("ms-appx:///Assets/WideLogo.png"),
    anchorElement,
    Windows.UI.Popups.Placement.Above,
    this.FeedItem.Id.ToString());

  this.IsFeedItemPinned = await this.secondaryPinner.Pin(tileInfo);

  if (this.IsFeedItemPinned)
  {
    await SavePinnedFeedItem();
  }
}
```

Extreme Performance & Linear Scalability

Remove data storage and database performance bottlenecks and scale your applications to extreme transaction processing (XTP). NCache lets you cache data in memory and reduce expensive database trips. It also scales linearly by letting you add inexpensive cache servers at runtime.

Enterprise Distributed Cache

- Extremely fast & linearly scalable with 100% uptime
- Mirrored, Replicated, Partitioned, and Client Cache
- NHibernate & Entity Framework Level-2 Cache

ASP.NET Optimization in Web Farms

- ASP.NET Session State storage
- ASP.NET View State cache
- ASP.NET Output Cache provider
- ASP.NET JavaScript & image merge/minify

Runtime Data Sharing

- Powerful event notifications for pub/sub data sharing

Download a 60-day FREE trial today!

FREE EXPRESS EDITION

NCache™

Distributed Cache for .NET & Java

www.alachisoft.com

1-800-253-8195

create that new instance of the view model, I only have to make the change in one place. In this case, GetViewModel is the helper method:

```
private MainViewModel GetViewModel()
{
  return new MainViewModel(this.RssFeedService, this.Navigator, this.MessageBus);
}
```

I also use the TestInitialize attribute to ensure the MainViewModel dependencies are created anew before running every test. Here's the TestInitialize method that makes this happen:

```
[TestInitialize]
public void Init()
{
  this.RssFeedService = new RssFeedServiceMock();
  this.Navigator = new NavigatorMock();
  this.MessageBus = new MessageBusMock();
}
```

With this, every unit test in this test class will have brand-new instances of all the mocks when they run.

Looking back at the test itself, the following code creates my expected feed data and sets up the mock RSS feed service to return it:

```
var expectedFeedData = new List<FeedData>();

this.RssFeedService.GetFeedsAsyncDelegate = () =>
  {
    return expectedFeedData;
  };
```

Notice I'm not adding any actual FeedData instances to the list of expectedFeedData because I don't need to. I only need to ensure the list itself is what MainViewModel ends up with. I don't care what happens when that list actually has FeedData instances in it, at least for this test.

The Act portion of the test has the following line:

```
var actualFeedData = viewModel.FeedData;
```

I can then assert the actualFeedData is the same instance of the expectedFeedData. If they aren't the same instance, then MainViewModel didn't do its job and the unit test should fail.

```
Assert.AreSame(expectedFeedData, actualFeedData);
```

Testable Navigation

Another important piece of the sample application I want to test is navigation. The Charmed Reader sample app uses view model-based navigation because I want to keep the views and view models separate. Charmed Reader is a cross-platform application and the view models I create are used on both platforms, although the views need to be different for Windows 8 and Windows Phone 8. There are a number of reasons why, but it boils down to the fact that each platform has slightly different XAML. Because of this, I didn't want my view models to know about my views, muddying up the waters.

Abstracting the navigation functionality behind an interface was the solution for several reasons. The first and foremost is that each platform has different classes involved in navigation, and I didn't want my view model to have to worry about these differences. Also, in both cases, the classes involved in navigation can't be mocked. So, I abstracted those issues away from the view model and created the INavigator interface:

```
public interface INavigator
{
  bool CanGoBack { get; }
  void GoBack();
  void NavigateToViewModel<TViewModel>(object parameter = null);

#if WINDOWS_PHONE
  void RemoveBackEntry();
#endif // WINDOWS_PHONE
}
```

I inject INavigator into the MainViewModel via the constructor, and MainViewModel uses INavigator in a method called ViewFeed:

```
public void ViewFeed(FeedItem feedItem)
{
  this.navigator.NavigateToViewModel<FeedItemViewModel>(feedItem);
}
```

When I look at how ViewFeed interacts with INavigator, I see two things I want to verify as I write the unit test:

1. The FeedItem that's passed into ViewFeed is the same FeedItem passed into NavigateToViewModel.
2. The view model type passed to NavigateToViewModel is FeedItemViewModel.

Before I actually write the test, I need to create another mock, this time for INavigator. **Figure 3** shows the mock for INavigator. I followed the same pattern as before with delegates for each method as a way to execute test code when the actual method gets called. Again, if you're working on a platform with mocking framework support, you don't need to create your own mock.

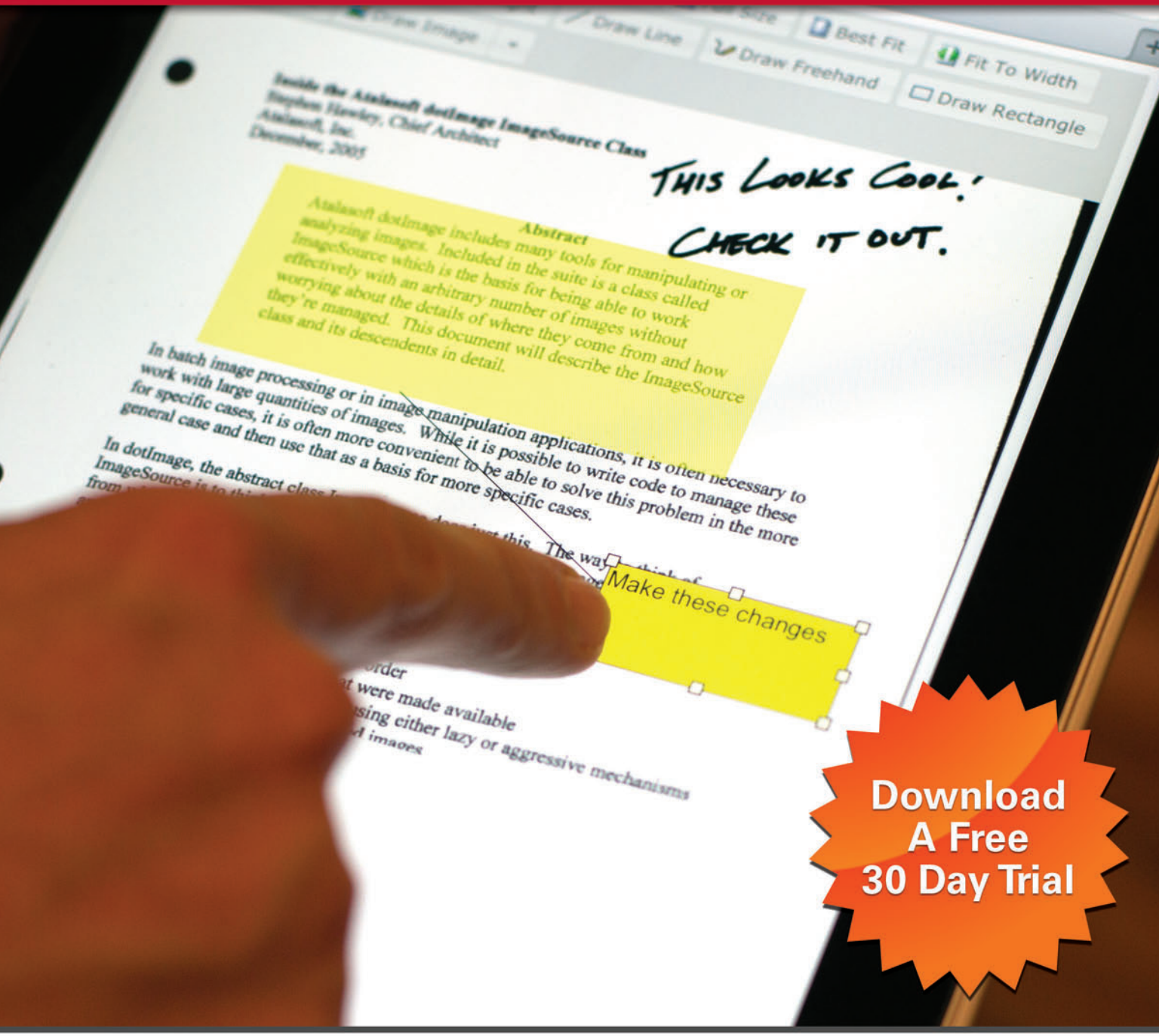
With my mock Navigator class in place, I can put it to use in a unit test, as shown in **Figure 4**.

What this test really cares about is that the FeedItem passed around is correct and the view model being navigated to is correct. When

Figure 6 **Testing for a Successful Pin**

```
[TestMethod]
public async Task Pin_PinSucceeded()
{
  // Arrange
  var viewModel = GetViewModel();

  var feedItem = new FeedItem
  {
    Title = Guid.NewGuid().ToString(),
    Author = Guid.NewGuid().ToString(),
    Link = new Uri("http://www.bing.com")
  };

  viewModel.LoadState(feedItem, null);

  Placement actualPlacement = Placement.Default;
  TileInfo actualTileInfo = null;
  SecondaryPinner.PinDelegate = (tileInfo) =>
    {
      actualPlacement = tileInfo.RequestPlacement;
      actualTileInfo = tileInfo;

      return true;
    };

  string actualKey = null;
  List<FeedItem> actualPinnedFeedItems = null;
  Storage.SaveAsyncDelegate = (key, value) =>
    {
      actualKey = key;
      actualPinnedFeedItems = (List<FeedItem>)value;
    };

  // Act
  await viewModel.Pin(null);

  // Assert
  Assert.AreEqual(Placement.Above, actualPlacement, "Placement");
  Assert.AreEqual(string.Format(Constants.SecondaryIdFormat,
    viewModel.FeedItem.Id), actualTileInfo.TileId, "Tile Info Tile Id");
  Assert.AreEqual(viewModel.FeedItem.Title,
    actualTileInfo.DisplayName, "Tile Info Display Name");
  Assert.AreEqual(viewModel.FeedItem.Title,
    actualTileInfo.ShortName, "Tile Info Short Name");
  Assert.AreEqual(viewModel.FeedItem.Id.ToString(),
    actualTileInfo.Arguments, "Tile Info Arguments");
  Assert.AreEqual(Constants.PinnedFeedItemsKey, actualKey, "Save Key");
  Assert.IsNotNull(actualPinnedFeedItems, "Pinned Feed Items");
}
```

Build A Mobile Document Viewer with Annotations, Touch Interfaces, Zooming, Pagination, and More



Download A Free 30 Day Trial

Atalasoft
A Kofax Company

.NET Web Scanning & Imaging SDKs

1.413.572.4443 ◆ www.atalasoft.com

working with mocks, it's important to keep in mind what you should care about for a particular test and what you aren't concerned with. For this test, because I have the INavigator interface that MainViewModel is working against, I don't need to care about whether the navigation actually takes place. That's handled by whatever implements INavigator for the runtime instance. I just need to care that INavigator is given the proper parameters when navigation occurs.

Testable Secondary Tiles

The final area for which I'm going to look at testing is secondary tiles. Secondary tiles are available in both Windows 8 and Windows Phone 8, and they let users pin elements of an app to their home screens, creating a deep link into a specific part of the app. However, secondary tiles are handled completely differently in the two platforms, which means I have to provide platform-specific implementations. Despite the difference, I'm able to provide a consistent interface for secondary tiles I can use on both platforms:

```
public interface ISecondaryPinner
{
  Task<bool> Pin(TileInfo tileInfo);
  Task<bool> Unpin(TileInfo tileInfo);
  bool IsPinned(string tileId);
}
```

The TileInfo class is a DTO with properties for both platforms combined to create a secondary tile. Because each platform uses a different combination of properties from TileInfo, each platform needs to be tested differently. I'll look specifically at the Windows 8 version. **Figure 5** shows how my view model uses ISecondaryPinner.

There are actually two things going on in the Pin method in **Figure 5**. The first is the actual pinning of the secondary tile. The second is saving the FeedItem into local storage. So, that's two things I need to test. Because this method changes the IsFeedItemPinned property on the view model based on the results of attempting to pin the FeedItem, I also need to test the two possible results of the Pin method on ISecondaryPinner: true and false. **Figure 6** shows the first test I implemented, which tests the success scenario.

There's a little more setup involved with this than with previous tests. First, after the controller, I set up a FeedItem instance. Notice that I call ToString on Guids for both Title and Author. That's because I don't care what the actual values are, I just care that they have values I can compare with in the assert section. Because Link is a Uri, I do need a valid Uri for this to work, so I provided one. Again, what the actual Uri is doesn't matter, just that it's valid. The remainder of the setup involves ensuring I capture the interactions for pinning and saving for comparison in the assert section. The key to ensuring this code actually tests the success scenario is that the PinDelegate returns true, indicating success.

Figure 7 shows pretty much the same test, but for the unsuccessful scenario. The fact that PinDelegate returns false is what ensures the test is focusing on the unsuccessful scenario. In the unsuccessful scenario, I also need to verify in the assert section that SaveAsync wasn't called.

Writing testable applications is a challenge. It's especially challenging to test the presentation layer where user interaction is involved. Knowing in advance that you're going to write a testable application lets you make decisions at every step in favor of testability. You can also look out for things that will make your app less testable and come up with ways to fix them.

Over the course of three articles, I've discussed writing testable apps with the MVVM pattern, specifically for Windows 8 and Windows Phone 8. In the first article, I looked at writing testable Windows 8 applications, while still leveraging Windows 8-specific features not easily testable by themselves. The second article evolved the discussion to include developing testable cross-platform apps with Windows 8 and Windows Phone 8. With this article, I showed how I approach testing the apps I worked so hard to make testable in the first place.

MVVM is a broad topic, with a number of different interpretations. I'm glad to have been able to share my interpretation of such an interesting topic. I find a lot of value in using MVVM, especially as it relates to testability. I also find exploring testability to be stimulating and useful, and I'm glad to share my approach to writing a testable application. ■

Figure 7 **Testing for an Unsuccessful Pin**

```
[TestMethod]
public async Task Pin_PinNotSucceeded()s
{
  // Arrange
  var viewModel = GetViewModel();

  var feedItem = new FeedItem
  {
    Title = Guid.NewGuid().ToString(),
    Author = Guid.NewGuid().ToString(),
    Link = new Uri("http://www.bing.com")
  };

  viewModel.LoadState(feedItem, null);

  Placement actualPlacement = Placement.Default;
  TileInfo actualTileInfo = null;
  SecondaryPinner.PinDelegate = (tileInfo) =>
  {
    actualPlacement = tileInfo.RequestPlacement;
    actualTileInfo = tileInfo;

    return false;
  };

  var wasSaveCalled = false;
  Storage.SaveAsyncDelegate = (key, value) =>
  {
    wasSaveCalled = true;
  };

  // Act
  await viewModel.Pin(null);

  // Assert
  Assert.AreEqual(Placement.Above, actualPlacement, "Placement");
  Assert.AreEqual(string.Format(Constants.SecondaryIdFormat,
    viewModel.FeedItem.Id), actualTileInfo.TileId, "Tile Info Tile Id");
  Assert.AreEqual(viewModel.FeedItem.Title, actualTileInfo.DisplayName,
    "Tile Info Display Name");
  Assert.AreEqual(viewModel.FeedItem.Title, actualTileInfo.ShortName,
    "Tile Info Short Name");
  Assert.AreEqual(viewModel.FeedItem.Id.ToString(),
    actualTileInfo.Arguments, "Tile Info Arguments");
  Assert.IsFalse(wasSaveCalled, "Was Save Called");
}
```

Brent Edwards *is a principal lead consultant for Magenic, a custom application development firm that focuses on the Microsoft stack and mobile application development. He is also a cofounder of the Twin Cities Windows 8 User Group in Minneapolis. Reach him at brente@magenic.com.*

THANKS to the following technical expert for reviewing this article: *Jason Bock (Magenic)*

SpreadsheetGear

Performance Spreadsheet Components

SpreadsheetGear 2012 Now Available

NEW!

WPF and Silverlight controls, multithreaded recalc, 64 new Excel compatible functions, save to XPS, improved efficiency and performance, Windows 8 support, Windows Server 2012 support, Visual Studio 2012 support and more.

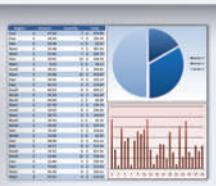
Excel Reporting for ASP.NET, WinForms, WPF and Silverlight

Easily create richly formatted Excel reports without Excel from any ASP.NET, Windows Forms, WPF or Silverlight application using spreadsheet technology built from the ground up for performance, scalability and reliability.

Excel Compatible Windows Forms, WPF and Silverlight Controls

Add powerful Excel compatible viewing, editing, formatting, calculating, filtering, charting, printing and more to your Windows Forms, WPF and Silverlight applications with the easy to use WorkbookView controls.

Excel Dashboards, Calculations, Charting and More



You and your users can design dashboards, reports, charts, and models in Excel or the SpreadsheetGear Workbook Designer rather than hard to learn developer tools and you can easily deploy them with one line of code.

Free 30 Day Trial

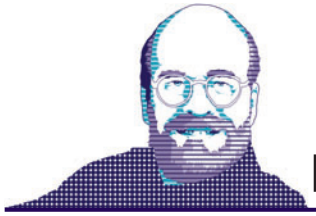
Download our fully functional 30-Day evaluation and bring Excel Reporting, Excel compatible charting, Excel compatible calculations and much more to your ASP.NET, Windows Forms, WPF, Silverlight and other Microsoft .NET Framework solutions.

www.SpreadsheetGear.com

Visual Studio Microsoft Partner

SpreadsheetGear

Toll Free USA (888) 774-3273 | Phone (913) 390-4797 | sales@spreadsheetgear.com



CHARLES PETZOLD

Who's Afraid of Glyph Runs?

I first encountered the concept of character glyphs in Windows Presentation Foundation (WPF) some years back. In typography, "glyph" refers to the graphical representation of a character rather than its function in a particular written language, which is indicated by the character's Unicode value. I knew the WPF Glyphs element and GlyphRun class represented an alternative way to display text, but they seemed to add a layer of unnecessary complexity. The biggest oddity involved referencing a desired font not by a font family name, but by the file path of the actual font file. I'd been coding for Windows since beta versions of 1.0, and I'd never before needed to reference a font by the filename. It simply wasn't how it's done.

I quickly concluded that Glyphs and GlyphRun were much too esoteric for mainstream programmers such as myself. Only years later when I began working with the Microsoft XML Paper Specification (XPS) did the rationale behind glyphs become apparent.

The Basic Distinction

Normally when a program requests a font from Windows, it does so using a font family name such as Century Schoolbook and attributes such as italic or bold. At runtime, Windows finds a font file on the user's system with a font that matches the family name. Italic and bold can either be intrinsic to this font or synthesized by the system. To lay out text, either Windows or the application accesses font metric information that describes the sizes of the font characters.

What happens if two different users have two different font files on their systems that both contain a font with the family name Century Schoolbook, but with somewhat different font metrics? That's not a real problem. Because text is formatted and laid out at runtime, the actual rendering of these two fonts could be slightly different, but it wouldn't really matter. Application programs are designed to be flexible in that way.

XPS documents are different, however. Much like PDF, XPS describes a fixed-page document. All the text on each page is already laid out and precisely positioned. If such a document is rendered with a font that is slightly different from the font used to design the page, it won't look right. This is why XPS documents often contain embedded font files, and why the XPS pages use the Glyphs element to reference these font files and render the text. Any ambiguity is entirely eliminated.

Code download available at archive.msdn.microsoft.com/mag201311DXF.

Normally we refer to text characters by character codes. We usually don't need to know anything about the precise glyph that happens to be displayed as a consequence of using that character code with a particular font. But sometimes it's important to have more control over the glyphs themselves.

You might assume there's a one-to-one correspondence between character codes and glyphs in a particular font, and that's usually the case. But there are very important exceptions: Some font files contain ligatures, which are single glyphs corresponding to character pairs such as fi or fl. Some non-Latin character sets require multiple glyphs to render a single character. In addition, some font files contain alternate glyphs for certain characters—for example, a zero with a slash through it, small capital letters used for lower case or letters with stylistic swashes. These stylistic alternatives are characteristic of a particular font file rather than a font. That's why getting the right font file is crucial.

> Exploring how to display glyphs in DirectX involves tracing a path through several interfaces, methods and structures.

If you're writing a Windows Store application for Windows 8, you can make use of these stylistic alternatives through the Windows Runtime API. Chapter 16 of "Programming Windows, 6th Edition" (O'Reilly Media, 2012) has a program called TypographyDemo that demonstrates how to do just that. But the underlying support for glyphs in Windows 8 is implemented in DirectX, and the power of glyphs themselves is revealed the most in that environment.

The DirectX Glyph Support

Exploring how to display glyphs in DirectX involves tracing a path through several interfaces, methods and structures.

Let's begin with the ID2D1RenderTarget interface, which contains the basic methods to display 2D graphics and text. This interface defines a DrawGlyphRun method. This method requires a structure of type DWRITE_GLYPH_RUN that describes the glyphs to be displayed, and also references an object of type IDWriteFontFace.

One way to obtain an IDWriteFontFace object is through the CreateFontFace method of IDWriteFactory. That method requires an IDWriteFontFile object, which is obtained through the CreateFontFileReference method of IDWriteFactory.

The CreateFontFileReference method references a font file using a path and filename. However, if you're using DirectX in a Windows Store application, your app probably won't have full and free access to the user's hard drive, and you can't usually access fonts in that manner. Any font file you reference with CreateFontFileReference will likely be defined as an application resource and be bound into the application's package.

However, you can't include just any font file in your application's package. If a font is part of your application, you must have a license to distribute that font. Fortunately, Microsoft has licensed several font files from Ascender Corp. for the express purpose of allowing you to distribute them with your application. These font files have been used primarily for XNA applications, but they're also interesting from a glyphs perspective.

Among the downloadable code for this column is a program called GlyphDump that I created based on the DirectX App (XAML) template in Visual Studio Express 2013 Preview. This template creates an application that renders DirectX graphics on a SwapChainPanel.

I created a new filter (and corresponding folder) in the GlyphDump project named Fonts, and added the 10 font files licensed from Ascender Corp. A ListBox in DirectXPage.xaml explicitly lists the family names of these fonts, with Tag properties referencing the filenames. When one is selected, the program constructs the path and name of the font file like so:

```
Package::Current->InstalledLocation->Path +
  "\\Fonts\\" + filename;
```

The GlyphDumpRenderer class then uses that path to create an IDWriteFontFile object and an IDWriteFontFace object.

The rest of the job occurs in the Render method, as shown in **Figure 1**.

This Render method displays all the glyphs in the selected font file in rows of 16 glyphs each, so it attempts to calculate a fontEmSize value sufficiently small for that display. (For some fonts with many glyphs, this won't work well.) Normally the glyphIndices field of the DWRITE_GLYPH_RUN structure is an array of glyph indices. Here I'm only displaying one glyph at a time.

The DrawGlyphRun method requires a coordinate point, the DWRITE_GLYPH_RUN structure and a brush. The coordinate indicates where the left edge of the glyph's baseline is to be positioned. Notice I said "baseline." This is different from most text display methods, which require specifying the top-left corner of the first character.

Figure 2 shows perhaps the most interesting font of this batch, which has a family name of Pescadero and a font filename of Pesca.ttf.

Before writing this program, I had never seen a display like this. It starts out looking something like a traditional ASCII table, but then there are a bunch of glyphs for numeric subscripting and superscripting, and a whole collection of ligatures of various sorts, plus capital letters with decorative swashes.

Obviously, the glyphIndices field of DWRITE_GLYPH_RUN isn't the same as a character code. These are indices that reference the actual glyphs within the font file, and these glyphs don't even need to be in any kind of rational order.

Text with Glyphs

You might already see some utility in being able to display text using a specific font file with glyph indices rather than character codes. You can specify exactly what glyphs you want.

The FancyTitle project demonstrates this. The idea here is that you have a program that's mostly XAML-based, but you want a fancy title using some ligatures and swashes from the Pescadero font. The project includes the Pesca.ttf file, and the XAML file defines the SwapChainPanel to have a width of 778 and a height of 54, values I chose empirically based on the size of the rendered text.

Because the display requirements of this project are simple, I removed the FancyTitleMain class and the rendering classes, leaving the DirectXPage class to render to the SwapChainPanel. (However, I had to modify DeviceResources slightly to make IDeviceNotify a ref class interface so that DirectXPage could implement IDeviceNotify and be notified when the output device is lost and recreated.)

The text output shown in **Figure 3** has 24 characters, but only 22 glyphs. You'll recognize the ligatures and the swashes from **Figure 2**.

Figure 1 **The Render Method in GlyphDump**

```
bool GlyphDumpRenderer::Render()
{
  if (!m_needsRedraw)
      return false;

  ID2D1DeviceContext* context = m_deviceResources->GetD2DDeviceContext();
  context->SaveDrawingState(m_stateBlock.Get());
  context->BeginDraw();
  context->Clear(ColorF(ColorF::White));
  context->SetTransform(m_deviceResources->GetOrientationTransform2D());

  if (m_fontFace != nullptr)
  {
    Windows::Foundation::Size outputBounds = m_deviceResources->GetOutputBounds();
    uint16 glyphCount = m_fontFace->GetGlyphCount();
    int rows = (glyphCount + 16) / 16;
    float boxHeight = outputBounds.Height / (rows + 1);
    float boxWidth = outputBounds.Width / 17;

    // Define entire structure except glyphIndices
    DWRITE_GLYPH_RUN glyphRun;
    glyphRun.fontFace = m_fontFace.Get();
    glyphRun.fontEmSize = 0.75f * min(boxHeight, boxWidth);
    glyphRun.glyphCount = 1;
    glyphRun.glyphAdvances = nullptr;
    glyphRun.glyphOffsets = nullptr;
    glyphRun.isSideways = false;
    glyphRun.bidiLevel = 0;

    for (uint16 index = 0; index < glyphCount; index++)
    {
      glyphRun.glyphIndices = &index;

      context->DrawGlyphRun(Point2F(((index) % 16 + 0.5f) * boxWidth,
                                    ((index) / 16 + 1) * boxHeight),
                                    &glyphRun,
                                    m_blackBrush.Get());
    }
  }

  // We ignore D2DERR_RECREATE_TARGET here. This error indicates
  // that the device is lost. It will be handled during
  // the next call to Present.
  HRESULT hr = context->EndDraw();
  if (hr != D2DERR_RECREATE_TARGET)
  {
    DX::ThrowIfFailed(hr);
  }

  context->RestoreDrawingState(m_stateBlock.Get());
  m_needsRedraw = false;
  return true;
}
```

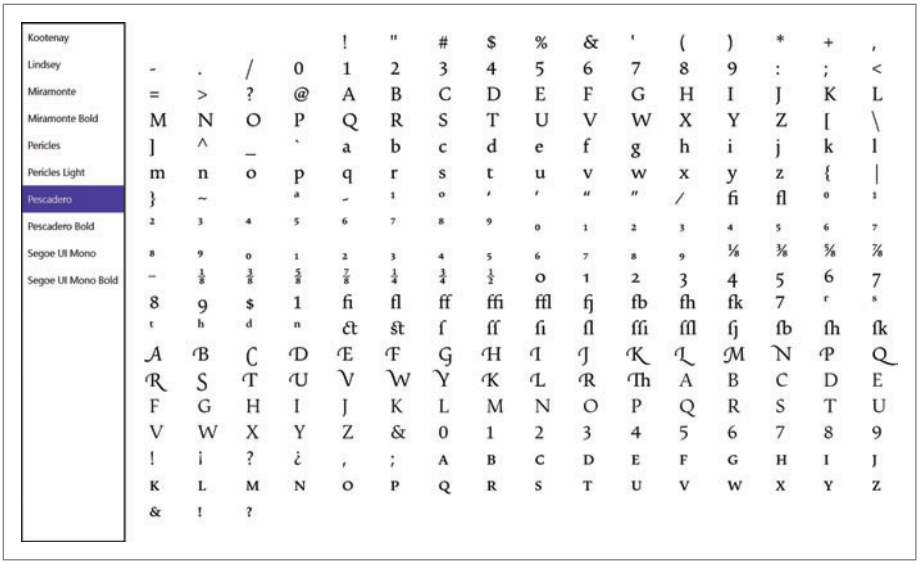


Figure 2 **The GlyphDump Program Showing the Pescadero Font**

I've made the DirectX background Alice Blue so you can see that the SwapChainPanel is barely larger than the text. Of course, it's possible to anticipate the size of the output exactly because the font file is part of the application and the glyphs are being accessed directly.

You can get ligatures and alternative glyphs without using DrawGlyphRun. You can also get them with less precision using the TextBlock element in the Windows Runtime, as the TypographyDemo program from my Windows 8 book demonstrates. In DirectWrite, you can use the SetTypography method of IDWriteTextLayout with an IDWriteTypography object, which ultimately references a member of the extensive DWRITE_FONT_FEATURE_TAG enumeration. But these techniques aren't quite as certain as specifying the glyphs precisely.

How do you get italics and boldface with DrawGlyphRun? In many cases, different font files will contain italic and bold variations of a font. Among the fonts included by the GlyphDump program are a couple of Bold fonts and a Light font. However, you also have the option to simulate oblique and bold characters using DWRITE_FONT_SIMULATIONS flags in CreateFontFace.

Advances and Offsets

Both GlyphDump and FancyTitle set two of the fields in DWRITE_GLYPH_RUN to nullptr. These two fields are named glyphAdvances and glyphOffsets.

When you display an array of glyphs, you specify the origin of the first character's left baseline. For each successive character, the horizontal coordinate of the origin is automatically increased based on the width of the character. (A similar process occurs when displaying sideways text.) This increase is known as an "advance."

You can obtain the advances that DirectX uses for spacing characters by calling the GetDesignGlyphMetrics method of IDWriteFontFace. The result is an array of DWRITE_GLYPH_METRICS structures, one for each glyph index in which you're interested . The advanceWidth field of that structure indicates the advance relative to the designUnitsPerEm field of the DWRITE_FONT_METRICS structure obtained from the GetMetrics method of IDWriteFontFace, and which also includes vertical metrics applicable to all glyphs within a particular font.

Or, you can call the GetDesignGlyphAdvances method of IDWriteFontFace1, which also provides advances relative to the designUnitsPerEm value. Divide the values by designUnitsPerEm (which is often a nice round value such as 2,048) and then multiply by the em size specified in DWRITE_GLYPH_RUN.

The glyphAdvances array is commonly used to space characters closer together or further apart than the design metrics indicate. If you decide to use it, you'll need to set glyphAdvances to an array of values that's at least the size of the array of glyph indices, minus one. An advance isn't needed on the final glyph because nothing is displayed beyond that.

The glyphOffsets field is an array of DWRITE_GLYPH_OFFSET structures, one for each character. The two fields are advanceOffset and ascenderOffset, which indicate a desired offset (to the right and up, respectively) relative to the character's normal location. This facility is often used in XPS files to position multiple glyphs of a particular font all over the page.

The CenteredGlyphDump program demonstrates how to use the glyphOffsets field to display the entire array of glyphs from a particular font file with a single DrawGlyphRun call:

```
context->DrawGlyphRun(Point2F(), &m_glyphRun, m_blackBrush.Get());
```

The coordinate passed to DrawGlyphRun is (0, 0), and glyphAdvances is set to an array of zero values to inhibit advancing successive glyphs. The position of each glyph is entirely governed by glyphOffsets. This position is based on glyph metrics to center each glyph in its column. **Figure 4** shows the result.

Providing Your Own Font Loader

If a font file is part of an application package, it's fairly easy to derive a file path you can use with CreateFontReference. But what if you have a font that's located in an XPS or EPUB package, or perhaps in isolated storage or in the cloud?

As long as you can write code that accesses the font file, DirectX can access it. You'll need to provide two classes, one that implements IDWriteFontFileLoader and another that implements IDWriteFontFileStream. Generally, the class that implements IDWriteFontFileLoader is a singleton that accesses all the font files your application will need, and assigns each of them a key. The CreateStreamFromKey method in your IDWriteFontFileLoader implementation returns an instance of IDWriteFontStream for each font file.

Figure 3 **The FancyTitle Output**

Work With Relational Data, Not Complex APIs or Services

Whether you are a developer using ADO.NET, JDBC, OData, or MySQL, or a systems integrator working with SQL Server or Biztalk, or even an information worker familiar with ODBC or Excel – our products give you bi-directional access to live data through easy-to-use technologies that you are already familiar with. If you can connect to a database, then you will already know how to connect to Salesforce, SAP, SharePoint, Dynamics CRM, Google Apps, QuickBooks, and much more!

Give RSSBus a try today and see what mean:

visit us online at www.rssbus.com to learn more or download a free trial.

rssbus

INTEGRATION YOUR WAY

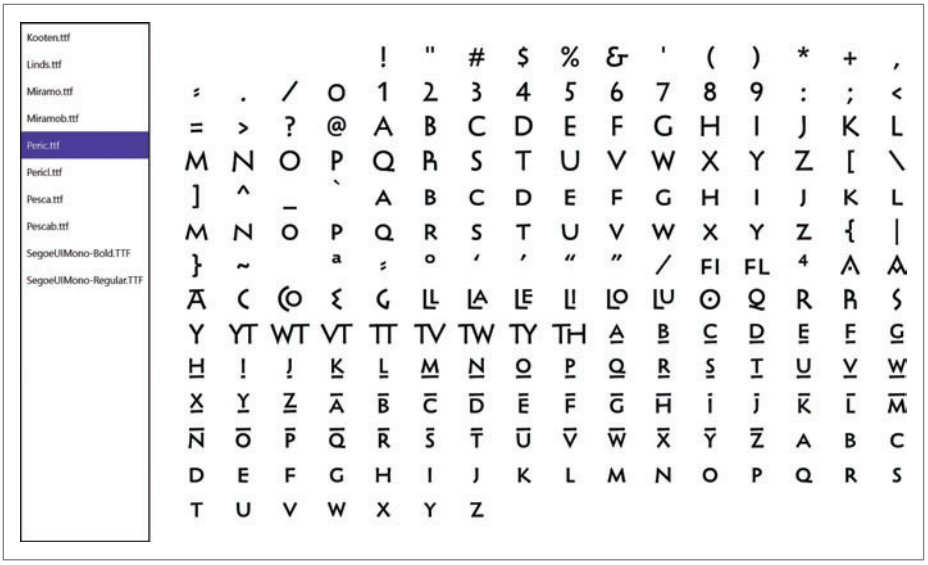


Figure 4 **The CenteredGlyphDump Program**

To use these two classes, you first instantiate a single instance of the class that implements IDWriteFontFileLoader and pass it to the RegisterFontFileLoader of your IDWriteFactory object. Then, instead of calling CreateFontFileReference to obtain an IDWriteFontFile object, call CreateCustomFontFileReference with this IDWriteFontFileLoader instance and a key identifying the particular font file you want.

This technique is demonstrated in the CenteredGlyphDump program. The project includes two classes—PrivateFontFileLoader and PrivateFontFileStream—that implement the two interfaces. The classes access font files in the application package, but they could be adapted for other purposes.

It's likely your IDWriteFontFileLoader implementation will need to make file I/O calls, and in a Windows Store application, these calls must be asynchronous. It makes a lot of sense for the IDWriteFontFileLoader class to define an asynchronous method that loads all the fonts into memory, and for the IDWriteFontFileStream to simply return pointers to these memory blocks. This is the approach I took with PrivateFontFileLoader and PrivateFontFileStream, after closely examining the Direct2D Magazine App Sample among the Microsoft sample code for Windows 8.1.

For the key to identify each file, I used the filename. After the asynchronous loading method in PrivateFontFileLoader is finished, the CenteredGlyphDump program obtains these filenames for the ListBox. That's why only the filenames are displayed in **Figure 4**. The program is ignorant of the font family names associated with each file.

From Characters to Glyphs

Of course, displaying text by referencing glyphs is fine if you know exactly what glyphs you need, but can you display normal Unicode characters using DrawGlyphRun?

You can, because IDWriteFontFace has a GetGlyphIndices method that converts character codes to glyph indices for the particular font file. In doing this, it picks the default glyph for each character code, so you won't get any of the fancy alternatives.

But the character codes you pass to GetGlyphIndices are in a form perhaps unique in all of Windows. Instead of 16-bit character codes (as is the case when you're normally working with Unicode character strings), you need to create an array of 32-bit character codes. As you may know, Unicode is actually a 21-bit character set, but characters are commonly stored as 16-bit values (called UTF-16), which means that some characters comprise two 16-bit values. But GetGlyphIndices wants 32-bit values. Unless your character string has characters with codes above 0xFFFF, you can simply transfer values from one array to another.

If you're dealing with normal characters from the Latin alphabet, you can also assume that there's a one-to-one correspondence between characters and glyphs. Otherwise, you might need to create a larger output array to receive the indices.

This simple technique is demonstrated by the HelloGlyphRun project. **Figure 5** shows the code that loads a font file and sets up a DWRITE_GLYPH_RUN structure. The Update method adjusts the glyph offsets to achieve a kind of rippling effect of the text.

Although you'll probably be using DrawGlyphRun only with font files you're intimately familiar with, it's possible to determine what type of glyphs are contained in a particular font file at runtime. IDWriteFontFace defines a TryGetFontTable method that accesses tables in the OpenFont file. Use a tag of "cmap" for the character-to-glyph table and "GSUB" for the glyph substitution table, but be prepared to spend many excruciating hours with the OpenType specification to read these tables successfully.

Figure 5 **Defining a DWRITE_GLYPH_RUN from a Character String**

```
// Convert string to glyph indices
std::wstring str = L"Hello, Glyph Run!";
uint32 glyphCount = str.length();
std::vector<uint32> str32(glyphCount);

for (uint32 i = 0; i < glyphCount; i++)
     str32[i] = str[i];

m_glyphIndices = std::vector<uint16>(glyphCount);
m_fontFace->GetGlyphIndices(str32.data(), glyphCount, m_glyphIndices.data());

// Allocate array for offsets (set during Update)
m_glyphOffsets = std::vector<DWRITE_GLYPH_OFFSET>(glyphCount);

// Get output bounds
Windows::Foundation::Size outputBounds = m_deviceResources-
>GetOutputBounds();

// Define fields of DWRITE_GLYPH_RUN structure
m_glyphRun.fontFace = m_fontFace.Get();
m_glyphRun.fontEmSize = outputBounds.Width / 8; // Empirical
m_glyphRun.glyphCount = glyphCount;
m_glyphRun.glyphIndices = m_glyphIndices.data();
m_glyphRun.glyphAdvances = nullptr;
m_glyphRun.glyphOffsets = m_glyphOffsets.data();
m_glyphRun.isSideways = false;
m_glyphRun.bidiLevel = 0;
```

Glyph Runs Using System Fonts

Is DrawGlyphRun only good for font files you supply yourself? At first it would seem so, but you can use it with system fonts as well. Here's the process: Use the GetSystemFontCollection method of IDWriteFactory to obtain an IDWriteFontCollection object. That object allows you to find all the family names associated with the fonts installed on the system. The GetFontFamily method of IDWriteFontCollection returns an object of type IDWriteFontFamily. From that, you can call GetMatchingFonts or GetFirstMatchingFont to combine the font family with italic, bold and stretch attributes to obtain an IDWriteFont.

Figure 6 **The OutlinedCharacters Program**

Once you have an IDWriteFont object, call CreateFontFace to get an IDWriteFontFace object. That's the same type of object obtained from CreateFontFace in the earlier programs! From that object you can begin setting up a DWRITE_GLYPH_RUN structure for DrawGlyphRun.

This is demonstrated in the SystemFontGlyphs project. The project uses the GetSystemFontCollection in in its DirectXPage class to fill a ListBox with the system font family names. When an item is selected, an IDWriteFontFace object is derived that's passed to the renderer.

The SystemFontGlyphsRenderer class builds a DWRITE_GLYPH_RUN structure based on the text "Annoying vibrating text effect." The Update method is then obliged to set the values of the glyph offsets array to random numbers between -3 and 3, making it seem as if all the characters of the text string are independently vibrating.

There doesn't seem to be much of a conceptual difference between IDWriteFont and IDWriteFontFace except that IDWriteFont always represents a font that's part of a font collection (such as the system font collection) while IDWriteFontFace need not. IDWriteFontCollection has a GetFontFromFontFace method that accepts an IDWriteFontFace and returns the corresponding IDWriteFont, but it works only if the font file associated with the IDWriteFontFace is part of the font collection.

Custom Font Collections

Now you've seen how you can use DrawGlyphRun with font files that your application loads as well as with system fonts. Is it possible to use the regular text-output methods (DrawText and DrawTextLayout) with your own font files?

Yes, it is. You'll recall that both DrawText and DrawTextLayout require an IDWriteTextFormat object. You create this using the CreateTextFormat method of IDWriteFactory by specifying both a font family name and a font collection.

Normally you set that font collection argument to nullptr to indicate system fonts. But you can also create a custom font collection by calling the CreateCustomFontCollection method of IDWriteFactory. You'll need to supply your own class that implements the IDWriteFontCollectionLoader interface, and another class that implements IDWriteFontFileEnumerator.

This is demonstrated in the ResourceFontLayout program. The program also includes the implementations of the IDWriteFontFileLoader and IDWriteFontFileStream interfaces from CenteredGlyphDump. When you obtain the list of font families from this custom font collection, you'll notice that multiple font files are sometimes consolidated into a single font family.

The Prize at the Bottom of the Box

At this point, the importance of IDWriteFontFace should be quite evident. You can create an IDWriteFontFace object from two directions: either directly from a font file, or by choosing a particular font from a font collection. You then reference this IDWriteFontFace in a DWRITE_GLYPH_RUN structure, or use it to obtain glyph metrics or to access tables in the font file.

IDWriteFontFace also defines a method named GetGlyphRunOutline. The arguments to this method are very much the same as the fields of the DWRITE_GLYPH_RUN structure, but an additional argument is an IDWriteGeometrySink, which is the same as an ID2D1SimplifiedGeometrySink.

This means you can convert a text string to an ID2D1PathGeometry, and then render and manipulate that geometry in whatever way you want. **Figure 6** is a screenshot from the OutlinedCharacters program that shows a character geometry that has been stroked and filled; the same geometry rendered with a dotted style (which is animated to make the dots travel around the characters); and the geometry widened and outlined, in effect, outlining the outlines.

And I'm just getting started. ■

Charles Petzold *is a longtime contributor to* MSDN Magazine *and the author of "Programming Windows, 6th Edition" (O'Reilly Media, 2012), a book about writing applications for Windows 8. His Web site is charlespetzold.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Jim Galasyn and Justin Panian*

DAVID S. PLATT

Advice to the New CEO

In August, Steve Ballmer announced his impending retirement as CEO of Microsoft. He promises to choose a successor within a year.

I suspect it won't be me. I told Microsoft, as I told Harvard during its recent presidential search, I'd only take the job if the company gave me a completely free hand. (Neither institution asked me; I'm just saying.)

Nevertheless, you can depend on me to continue as the diligent court jester, speaking truth to power for as long as they'll listen or at least not try to shut me up. I call 'em as I see 'em. And I have some advice for that brave soul who takes the reins at Microsoft, whoever it turns out to be.

1. First and foremost, understand that tablets and phones are not PCs. My Surface keeps calling itself a PC. It isn't—I don't want it to be and neither did all the customers you hoped would buy one but didn't. PCs have keyboards and precise pointing devices. They devote equal resources to producing and consuming content. Tablets and phones lack these input devices and therefore are optimized for consuming content. As I wrote in my May and June 2011 columns, DEC died because it couldn't stop thinking of itself as a minicomputer company. Microsoft has to stop thinking of itself as a PC company. Ballmer said, "We're now the devices and services company." That didn't ring true coming from the ultimate PC warhorse. It's your job now to make it so.
2. With your current share of the smartphone (3 percent) and tablet (4.5 percent) markets, you have a horrendous chicken-and-egg problem. Who's going to buy your hardware until it has good apps, and who's going to write apps until there's a hardware base?

   The only way anyone has ever solved a chicken-and-egg problem like that is by giving away one-half to sell the other, classically giving away razors to sell blades. Spend less on advertising and more on making good devices cheap. Subsidize them if necessary. Microsoft charges $930 today for a Surface Pro with a keyboard. For that price, you could buy a 17-inch Dell laptop, a full-size iPad (with Retina Display!) *and* a case of decent beer. You won't solve your chicken-and-egg problem that way. Discontinued HP TouchPads flew off the shelf at $99, even with no apps and no future. Buy yourself some market share quickly to start the virtuous cycle.
3. Your biggest asset is your army of developers. This is an army of which your humble correspondent has created a tiny part, and helped shape a larger one. They would love to use their current skills to develop apps for this brave new device-and-tablet world. They have trouble seeing the return on investment at your current market share. But you can solve that. The worst problem any developer has in the device space is covering the multiplicity of OSes and platforms. Developers want to support iOS and Android. They'd support Windows 8, too, if it didn't take so much extra effort. Microsoft should develop a toolkit that covers all three from the same codebase.

   Wasn't that the original intent of the Microsoft .NET Framework, anyway, when the world was new and I still had hair? Developers, even non-Microsoft ones, would flock to you. It'll get you more client apps for Windows platforms, and also guide more devs toward using your cloud platform as the back end of whatever client apps they write. Either do it yourself, or buy Xamarin as a jump-start. Do it quickly, though, before the army gets hungry and switches sides.

> DEC died because it couldn't stop thinking of itself as a minicomputer company. Microsoft has to stop thinking of itself as a PC company.

4. Finally, listen to outsiders. It's an ongoing problem for the person atop any large organization. People beneath you only tell you what you want to hear. Everyone at Microsoft says to each other, "Sure, PCs are great, I use PCs every day. Yep, the Windows Phone OS is cool and so is the Surface." You get locked into this self-reinforcing positive feedback cycle where no one dares tell you that you forgot your pants today.

   Develop your own channels of independent information. Start by reading my column every month. If you really want the lowdown, I'll privately send you the early drafts before the editors sanitize it.

So, my incoming friend, I've mapped out your global strategy. All you have to do is execute. Good luck from your faithful court jester. You'll certainly need it. ■

DAVID S. PLATT *teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.*

.NET TOOLS FOR DEV PROS

Whether you're building the most modern touch-enabled apps or maintaining and updating legacy applications, our flagship product, Studio Enterprise, helps to deliver rich, responsive, desktop and web apps on time and under budget.

DOWNLOAD YOUR FREE TRIAL

componentone.com/se

ComponentOne
a division of GrapeCity