THE MICROSOFT JOURNAL FOR DEVELOPERS

AUGUST 2013 VOL 28 NO 8

msdn magazine

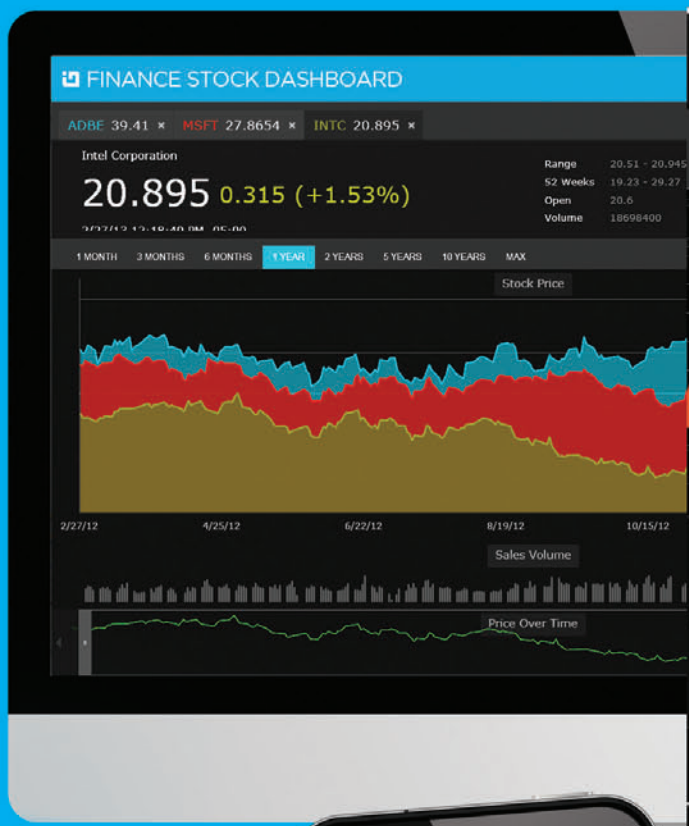
Using the C++ REST SDK.....26

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

AUGUST 2013 VOLUME 28 NUMBER 8

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.

LEADTOOLS V18
THE WORLD LEADER IN IMAGING SDKs

MULTIMEDIA SDKs

DirectShow and Media Foundation SDK for .NET, Win32/64

MPEG-2 TRANSPORT STREAM, DVR, KLV, RTSP

MPEG4, MPEG2, H.264, H.263, AAC, AC3 AND 100+CODECS

MULTIPLEXERS AND DEMULTIPLEXERS

MKV, MXF, MP4, AVI, WMV AND MPG

IIS SMOOTH STREAMING, UDP, RTP, ENCRYPTION, WMV

CAPTURE, PLAYBACK, CONVERSION, COMPRESSION

CLOUD SDK FOR DISTRIBUTED PROCESSING

PLAY, CONVERT, AUTHOR AND BURN DVDs AND CDs

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

MICHAEL DESMOND

A Better Windows 8

Microsoft has always been a patient company. Whether it was the long road Windows took from a graphical operating environment to a business-capable OS, or the years-long campaign to evolve SQL Server into an enterprise database, or the extended effort it took to mature Visual Studio and the .NET Framework, Microsoft has shown a willingness to invest in the technologies it believes are at the core of the company—even in the face of early struggles.

Which is why the Windows 8.1 Preview release at the Build 2013 conference in San Francisco in June was so important. Windows 8 has, to a large extent, labored under the weight of its own vision. Users puzzled at the innovative new UI even as developers scrambled to come to terms with the native programming model posed by the Windows Runtime. The combination provoked questions about the future of Microsoft's flagship OS and its prospects in an increasingly diverse computing landscape.

But Microsoft has courted this kind of sea change before. The move to 32-bit development, first with the launch of Windows 95 and later Windows NT and its successor OSes, demanded wholesale changes to the underpinnings of the Windows environment and its development infrastructure. The launch of the .NET Framework in 2001 was no less of a high-stakes transition, as Microsoft ushered its developer ecosystem into a managed code framework.

Neither of these transitions happened overnight. Windows 95 suffered from a dearth of optimized 32-bit applications in the months after its release, while efforts to evangelize the .NET Framework were hampered by immature tooling, confused messaging and a rough language transition (Visual Basic, anyone?).

With Windows 8 and the modern UI, Microsoft is taking on an even steeper challenge, presenting both a new programming framework—the Windows Runtime—and a radically changed UI. And while Windows 8 presented a case for modern apps designed around the new Microsoft design language, the OS itself was, frankly, incomplete.

Windows 8.1 doesn't address every question and criticism. Enterprise developers and IT pros may still be waiting on a viable, side-loading deployment model, for example. But the latest version of the OS, which will be available as a free update to all Windows 8 users, significantly improves on the initial release. The preview I've been running on my Dell XPS 12 convertible laptop since Build has been more stable, more responsive and more impressive than the shipping OS in virtually all aspects.

Just as important, Microsoft is filling in the gaps. There's a host of new business-oriented features in Windows 8.1 that improve security and management for IT operators. And IT managers and end users alike will welcome the option to desktop-ify Windows 8, restoring the Start button and booting directly to the traditional Windows 7-style desktop. Microsoft has also worked to fix things that were, admittedly, broken in Windows 8—like the Windows Store experience, which is suddenly much more enjoyable and informative under Windows 8.1.

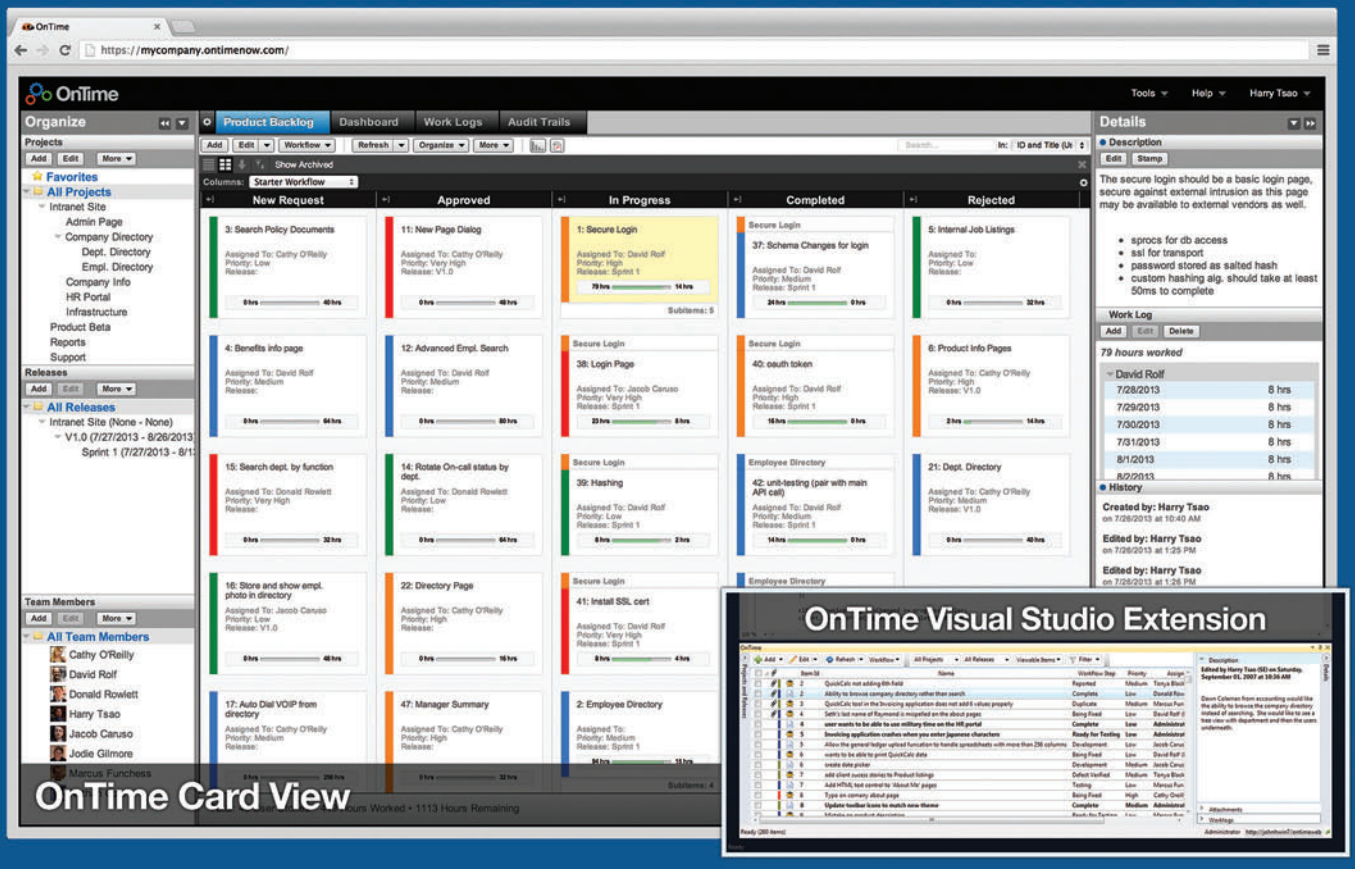
For developers, Windows 8.1 marches under the familiar banner of code compatibility. Win32- and .NET Framework-based applications will run (and are fully supported) in Windows 8.1, with a new version of the .NET Framework (4.5.1) previewed at the Build conference. Developers who are content with or must align to the traditional desktop presented by Windows 7 are fully empowered to do so.

No question, hard choices remain. Moving applications to the Windows Runtime and the new modern UI will impose costs. But the reality is that traditional desktop applications risk limited reach and effectiveness as users increasingly rely on tablets and other devices rather than traditional client PCs.

What do *you* think of Windows 8.1? E-mail me at mmeditor@microsoft.com.

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 to ship your software 24% faster? Think about it: that's **3 free months every year**, and it's the average time saved by users of OnTime Scrum.

How are they doing it? Because OnTime Scrum is fast, extremely customizable, and easy to use. Features like **Card View** make managing your user stories highly visible and effortless. And with our **Visual Studio integration**, your dev team can manage their OnTime items without leaving their development environment.

So 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

Creating Mobile-Optimized Views in ASP.NET MVC 4, Part 2: Using WURFL

Old memories of the browser wars of a decade ago still scare developers today. One of the reasons for the universal success of jQuery can be found in its ability to hide subtle and less-subtle differences in the Document Object Model (DOM) and JavaScript implementations across browsers. Today, developers supposedly can focus on features and forget about specific browser capabilities. Really? This is true for desktop browsers—but much less so for mobile browsers.

Mobile browsers form a space that's much more fragmented than desktop browsers were a decade ago. The number of different desktop browsers to take into account can be measured in tens; the number of different mobile browsers has an order of magnitude of thousands. Device form factors likely will evolve in both directions—getting smaller as with smartphones and mini-tablets, but also larger as with smart TVs. The key to success today and tomorrow is in serving users of each class of devices an appropriate experience. For that, you need to detect the device first and then its capabilities.

The Good Old Browser Capabilities Database

Since version 1.0, ASP.NET has offered a browser capabilities database. The Browser property exposed by the HttpRequest object could be queried for a small set of capabilities of the requesting browser. The mechanics of the browser capabilities infrastructure are fairly simple and effective. When building the HttpRequest object, the ASP.NET runtime grabs the user agent string and runs it by the ASP.NET browser database. Using the user agent string as the key, it retrieves a list of known capabilities and stores values into an HttpBrowserCapabilities object that developers access via Request.Browser. As far as I can see, this feature is unique to ASP.NET, and no other Web development platform has anything similar.

Figure 1 **Instructing ASP.NET MVC 4 to Support Up to Three Display Modes of the Same Razor View**

```
public class DisplayConfig
{
  public static void RegisterDisplayModes(
    IList<IDisplayMode> displayModes)
  {
    var modeSmartphone = new DefaultDisplayMode("smart")
    {
      ContextCondition = (c => c.Request.IsSmartphone())
    };
    var modeTablet = new DefaultDisplayMode("tablet")
    {
      ContextCondition = (c => c.Request.IsTablet())
    };
    var modeDesktop = new DefaultDisplayMode("")
    {
      ContextCondition = (c => c.Request.IsDesktop())
    };

    displayModes.Clear();
    displayModes.Add(modeSmartphone);
    displayModes.Add(modeTablet);
    displayModes.Add(modeDesktop);
  }
}
```

> Mobile browsers form a space that's much more fragmented than desktop browsers were a decade ago.

As you look into the underpinnings of browser capabilities it should become clear that this framework suffers from a relevant shortcoming: To be effective, the database that stores browser capabilities must be constantly updated as new devices and browser versions hit the market.

Originally, Microsoft designed the browser capabilities framework to help distinguish among a few desktop browsers. The advent of mobile devices completely changed the magnitude of the problem. For a while, Microsoft supported the Mobile Device Browser File (MDBF) project (mdbf.codeplex.com), aimed at creating a rich database of capability definitions for individual mobile devices and browsers. The idea was that by simply replacing the standard .browser files you get with ASP.NET with the MDBF database, you could access detailed browser and mobile device information through Request.Browser. A couple of years ago, however, the project was discontinued. But the underlying idea remains quite valid and probably the most effective way to serve tailor-made and highly optimized content to a few classes of devices. I'll explore how to do that using ASP.NET MVC 4. In the upcoming example, any information about devices is provided by the Wireless Universal Resource FiLe (WURFL) at wurfl.sourceforge.net. Among other things, WURFL is the Device Description Repository (DDR) used by many large organizations, including Google and Facebook. You can read more about WURFL and DDRs in my previous series of columns about mobile site development, starting with "Mobile Site Development: Markup" (msdn.microsoft.com/magazine/jj133814).

telerik

deliver more than expected

Telerik DevCraft Q2

RELEASE: JUNE 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 Q2 2013 Release & download your 30 day free trial at

www.telerik.com

The Mobile Face of ASP.NET MVC 4

In ASP.NET MVC 4, by invoking the code in **Figure 1** from within global.asax, you prepare the ground for having up to three different representations of the same Razor view.

RegisterDisplayModes instructs the ASP.NET MVC 4 runtime to consider three distinct display modes for each view. This means that every time a controller invokes a view, the actual name of the view (say, "index") will be examined by the display mode provider and will be changed to index.smart or index.tablet if the context condition defined for smartphones or tablets is verified. The order in which display mode objects are inserted in the provider is key. The search, in fact, stops at the first match. Suppose, for example, that the HTTP request results in the following code:

```
public ActionResult Index()
{
  return View();
}
```

While resolving the view name, ASP.NET MVC 4 will go through the display modes collection and first check the smartphone display mode. The context condition for smartphones returns either true or false depending on the implementation of IsSmartphone. If true, then a view with the "smart" suffix is selected if any exists. Otherwise, the search continues with the next available display modes.

Defining the number of display modes is up to you, as is deciding the conditions that determine which requests belong to which display modes.

To map a request to a display mode you need to do some device detection. However, you aren't going to have one view for each possible device, but rather one view for each class of devices you intend to support in your application.

> To map a request to a display mode you need to do some device detection.

Making Device Detection Smart

At the end of the day, device detection is about sniffing the user agent string. User agent strings aren't an exact science and might require a lot of parsing and normalization work to be digestible and easily mapped to a list of capabilities. Maintaining such a database is expensive, because you should look at every new browser version and device release and OS customization done by OEMs. In addition, for each unique device identified, you should figure out capabilities and store them efficiently in a database.

A few companies are active in this industry and sell their products according to various models, including fairly inexpensive cloud-based solutions. The most popular framework for device detection is the aforementioned WURFL—a cross-platform library available for a variety of languages that's also open source according to the Affero General Public License (AGPL). For ASP.NET MVC 4 developers, WURFL is also available as a NuGet package. In the Microsoft .NET Framework space, another choice is provided by the 51Degrees database (51degrees.mobi). In the next section, I'll examine how a device-detection framework can be used to route display modes in an ASP.NET MVC 4 application.

Figure 2 **Defining a Smartphone as a Virtual Capability**

```
public static class HttpRequestBaseExtensions
{
  public static Boolean IsSmartphone(this HttpRequestBase request)
  {
    var device =
      WURFLManagerBuilder.Instance.GetDeviceForRequest(userAgent);
    return device.IsWireless() && !device.IsTablet() &&
      device.IsTouch() &&
      device.Width() > 320 &&
      (device.HasOs("android", new Version(2, 2)) ||
      device.HasOs("iphone os", new Version(3, 2)) ||
      device.HasOs("windows phone os", new Version(7, 1)) ||
      device.HasOs("rim os", new Version(6, 0)));
  }
}
```

Adding WURFL to Display Modes

The code in **Figure 1** is essentially centered on the Context-Condition delegate:

```
Boolean ContextCondition(HttpContextBase)
```

The signature is self-explanatory: It passes the HTTP context and expects a Boolean answer. The logic in the delegate should consume any information in the HTTP context and determine whether the request can be served a view as intended by the display mode. As an example, I'll start from where I left off in my last column, "Creating Mobile-Optimized Views in ASP.NET MVC 4" (msdn.microsoft.com/magazine/dn296507). The IsTablet method used in **Figure 1** is an extension method added to the HttpRequestBase class. Here's its code:

```
public static class HttpRequestBaseExtensions
{
  public static Boolean IsTablet(this HttpRequestBase request)
  {
    var ua = userAgent.ToLower();
    return ua.Contains("ipad") || ua.Contains("gt-");
  }
}
```

The quick evolution of mobile devices is making it more difficult to find and maintain a static definition of what a tablet or smartphone is. Being a tablet or a smartphone can hardly be considered as a physical characteristic such as whether the device supports Flash video (.flv) streaming or inline images. The class of device is more of a virtual capability whose definition is entirely up to the development team. Virtual capabilities typically are implemented as the logical combination of multiple physical capabilities.

To add WURFL to your ASP.NET MVC 4 code, just invoke NuGet and get the official WURFL API. The package installs the WURFL database—a zipped file—in the App_Data folder. The database is a relatively recent snapshot of device information; to get updates on a weekly basis, you need to buy a commercial license from ScientiaMobile (scientiamobile.com).

Once WURFL is in place, you add the following line to Application_Start:

```
WURFLManagerBuilder.Build(new ApplicationConfigurer());
```

This will load the database into memory and ensure that all the data is cached for the quickest possible access. To run a WURFL query, you need the following code that must run in each and every request for an HTML view:

```
var userAgent = ...; // Typically read from Request object
var device = WURFLManagerBuilder.Instance.GetDeviceForRequest(userAgent);
```

Again, the code is self-describing. The WURFL framework gets the user agent and returns (in a matter of milliseconds) all the information it knows about the device. The information is expressed in the form of a name/value collection where both the name of the capability and its returned value are strings. Turning strings into strongly typed data (for example, integers or Booleans) is your responsibility. WURFL offers more than 500 capabilities per device catalogued in several categories. Clearly, you're not interested in all of them. I'd say that a more realistic number is one-tenth of that, which is also in line with the number of capabilities supported by the now-dead MDBF project. Anyway, that number amounts to a lot more capabilities than you can check with CSS3 media queries. Media queries have five total properties, only one or two of which (width and orientation) are often used. Here's how to reliably check for tablets using WURFL:

```
public static class HttpRequestBaseExtensions
{
  public static Boolean IsTablet(this HttpRequestBase request)
  {
    var device =
      WURFLManagerBuilder.Instance.GetDeviceForRequest(userAgent);
   return device.IsTablet();
  }
}
```

IsTablet is an extension method to the WURFL IDevice type that I just created to overcome the weakly typed nature of WURFL:

```
public static Boolean IsTablet(this IDevice device)
{
  return device.GetCapability("is_tablet").ToBool();
}
```

Note that ToBool is yet another extension method that just wraps up a call to Boolean.TryParse.

The usefulness of extension methods to keep the code clear and elegant is even more apparent in the sample code that detects smartphones, shown in **Figure 2**. There's probably no common definition of what a smartphone is that's acceptable to everybody. By combining multiple WURFL properties, you can create your own definition of a smartphone, as shown in **Figure 2**.

> Server-side detection and client-side responsive design aren't an either/or choice.

The code in **Figure 2** defines a smartphone as a wireless device that isn't a tablet, is touch-enabled, is at least 320 pixels wide and runs any of the specified OSes. All of the methods used on the device variable are extension methods built on top of WURFL native capabilities.

Server-Side Detection and Client-Side Responsive Design

The focus in some segments of the industry on client-side detection of features is justified by the goal of leveraging advanced HTML5 and CSS3 capabilities in Web sites. However, this has little to do with making sites mobile-friendly. Client-side feature detection is limited to what the browser allows detection of—at best the five properties to which you gain access through media queries and whatever can be programmatically tested. To distinguish an Android device from an iPhone, you still need user agent sniffing—not to mention that running Android 2.2, for example, doesn't say much about the actual device capabilities.

If you really need to serve ad hoc markup to devices (small and large), then server-side device and feature detection is the only way to go. WURFL makes it quick and easy.

Server-side detection and client-side responsive design aren't an either/or choice. Server-side detection is only about identifying the device and the related display mode. The markup you serve can easily contain media queries, liquid layouts and whatever else helps make the result better. If you like acronyms, this is what RESS (standing for Responsive Design + Server-Side Components) is all about. Server-side detection just adds one extra level of abstraction over the process of building tailor-made views. ■

Dino Esposito *is the author of "Architecting Mobile Solutions for the Enterprise" (Microsoft Press, 2012) and the forthcoming "Programming ASP.NET MVC 5" from 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: *Mani Subramanian (Microsoft)*

Aspose.Total

Every Aspose component combined in one powerful suite.

Powerful File Format Components and Controls

Aspose.Words
DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.

Aspose.Cells
XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.

Aspose.BarCode
JPG, PNG, BMP, GIF, TIF, WMF, ICON & other image formats.

Aspose.Pdf
PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.

Aspose.Email
MSG, EML, PST, EMLX & other formats.

Aspose.Slides
PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.

Aspose.Diagram
VSD, VSDX, VSS, VST, VSX & other formats.

... and many others!

Try Free Today!

- Aspose.Total for .NET
- Aspose.Total for Java
- Aspose.Total for SharePoint
- Aspose.Total for SSRS
- Aspose.Total for JasperReports
- Aspose.Total for Cloud

KENNY KERR

The Windows Runtime Application Model

Our lives are replete with abstractions. As developers, we're often left struggling when we use abstractions without understanding them for what they are. Abstractions are sometimes broken and fail to completely hide underlying complexity. Don't get me wrong, abstractions are great. They help users and they help developers, but you'll do yourself a world of good if you dig into the abstractions that you rely on regularly to understand how they operate. Moreover, libraries that acknowledge this reality are often more successful than ones that don't, in part because they allow you to step around the abstraction if and when you feel the need.

The Windows Runtime (WinRT) is one such abstraction, and in this month's column I'm going to illustrate this by examining the WinRT core application model. It revolves around the CoreWindow class, of which there's an instance that lives inside each and every "modern" Windows Store and Windows Phone app. Yet relatively few developers even know it exists, let alone how it works. Perhaps this is a testament to the success of the abstraction.

Since the Windows 8 API was first announced in 2011, a lot has been spoken and written about the various language projections that offer an abstraction over the Windows Runtime. However, the best way to understand the Windows Runtime is to eschew the various language projections, including C++/CX, and embrace standard C++ and classic COM. Only C++ lets you pull the curtain aside and see what's really going on (technically, so does C, but that would be needlessly painful). You might still choose to use some or other language projection (hopefully C++/CX), as you probably should, but at least you'll have a much clearer understanding of what's really going on.

To begin, open Visual Studio 2012 and create a new Visual C++ project for a Windows Store or Windows Phone app. It doesn't matter which template you use. Once it's loaded, go over to the Solution Explorer and delete everything that's nonessential. If you picked a XAML-based template, delete all of the XAML files. You can also delete all of the C++ sources files. You might want to hold on to the precompiled header, but be sure to delete everything inside of it. All that should remain are the package assets required to deploy the app, images, certificate and XML manifest.

Next, open the project's property pages and select the compiler properties—the C/C++ node in the tree on the left. Find the line for the /ZW compiler option that's called Consume Windows Runtime Extension and select No to disable the C++/CX language extensions. That way you can be sure there's nothing mysterious going on beyond the wonderful mysteries of the standard C++ compiler. While you're there, you might as well set the compiler's warning level to /W4.

If you try to compile the project, you should be greeted with a linker error informing you that the project's WinMain entry point function can't be found. Add a new C++ source file to the project, and the first thing you'll do is add the missing WinMain function:

```
int __stdcall wWinMain(HINSTANCE, HINSTANCE, PWSTR, int)
{
}
```

As you can see, this is the age-old WinMain function for a C Runtime Libraries (CRT)-based Windows application. Of course, HINSTANCE and PWSTR aren't fundamental C++ types, so you'll need to include the Windows header:

```
#include <windows.h>
```

If you kept the project's precompiled header, you can include it in there. I'll also be using ComPtr from the Windows Runtime C++ Template Library (WRL), so now would be a good time to include that as well:

```
#include <wrl.h>
```

I'll cover WRL in more detail over the next few columns. For now, I'll just make use of the ComPtr class template for maintaining a COM interface pointer. All you need to keep in mind at this stage is that the WRL ComPtr is simply a COM interface smart pointer. Although it provides certain features that are unique to the Windows Runtime, I won't be using them in this month's column. You could just as easily use the Active Template Library (ATL) CComPtr instead, or any COM interface smart pointer of your choice. The WRL ComPtr is defined in the Microsoft::WRL namespace:

```
using namespace Microsoft::WRL;
```

I'm also going to use an ASSERT macro as well as the HR function for error handling. I've discussed these previously, so I won't go over them again here. If you're unsure about these steps, check out my May 2013 column, "Introducing Direct2D 1.1" (msdn.microsoft.com/magazine/dn198239).

Finally, to use any of the WinRT functions mentioned in this column, you need to give the linker the name of the .lib file:

```
#pragma comment(lib, "RuntimeObject.lib")
```

The first thing the application model expects is a multithreaded apartment (MTA). That's right—the COM apartment model lives on. The Windows Runtime provides the RoInitialize function, which is a thin wrapper around CoInitializeEx:

```
HR(RoInitialize(RO_INIT_MULTITHREADED));
```

Despite the fact that CoInitializeEx is usually sufficient, I suggest you use RoInitialize. This function allows future improvements to be made to the Windows Runtime without potentially breaking classic COM. It's analogous to OleInitialize, which also calls CoInitializeEx and then some. The point is that there's nothing mysterious about your application's main thread. The only thing that might be slightly surprising is that it's not a single-threaded

apartment (STA). Don't worry, your application's window will still run from within an STA thread, but the Windows Runtime will be the one creating it. This STA is actually an Application STA (ASTA), which is slightly different, but more on that later.

The next bit is a little tricky. The Windows Runtime forsakes the traditional COM activation model that uses GUID-based class identifiers in favor of a model that activates classes based on textual class identifiers. The textual names are based on namespace-scoped class names popularized by Java and the Microsoft .NET Framework, but before you scoff and say good riddance to the registry, keep in mind that these new class identifiers are still stored in the registry. Technically only first-party types are registered in the registry, whereas third-party types are only registered in a per-application manifest. There are pros and cons to this change. One of the cons is that it's slightly harder to describe the class identifier when calling a WinRT function. The Windows Runtime defines a new remotable string type to replace the traditional BSTR string type, and any class identifiers need to be provided using this new medium. The HSTRING, as it's called, is far less error-prone than BSTR, chiefly because it's immutable. The simplest way to create an HSTRING is with the WindowsCreateString function:

```
wchar_t buffer[] = L"Poultry.Hatchery";
HSTRING string;

HR(WindowsCreateString(buffer,
                       _countof(buffer) - 1,
                       &string));
```

WindowsCreateString allocates enough memory to store a copy of the source string as well as a terminating null character and then copies the source string into this buffer. To release the backing buffer you must remember to call WindowsDeleteString, unless ownership of the string is returned to a calling function:

```
HR(WindowsDeleteString(string));
```

Given an HSTRING, you can get a pointer to its backing buffer with the WindowsGetStringRawBuffer function:

```
wchar_t const * raw = WindowsGetStringRawBuffer(string, nullptr);
```

The optional second parameter returns the length of the string. The length is stored with the string, saving you from having to scan the string to determine its length. Before you run off and write a C++ wrapper class, it's worth noting that the C++/CX compiler doesn't bother with WindowsCreateString and WindowsDeleteString when generating code for a string literal or const array.

Figure 1 **GetActivationFactory Function Template**

```
template <typename T, unsigned Count>
auto GetActivationFactory(WCHAR const (&classId)[Count]) -> ComPtr<T>
{
  HSTRING_HEADER header;
  HSTRING string;

  HR(WindowsCreateStringReference(classId,
                                  Count - 1,
                                  &header,
                                  &string));

  ComPtr<T> result;

  HR(RoGetActivationFactory(string,
                            __uuidof(T),
                            reinterpret_cast<void **>(result.GetAddressOf())));

  return result;
}
```

Instead, it uses what's known as a fast-pass string that avoids the extra memory allocation and copy that I previously mentioned. This also avoids the risk of a memory leak. The WindowsCreateStringReference function creates a fast-pass string:

```
HSTRING_HEADER header;
HSTRING string;

HR(WindowsCreateStringReference(buffer,
                                _countof(buffer) - 1,
                                &header,
                                &string));
```

This function uses the caller-provided HSTRING_HEADER to avoid a heap allocation. In this case, the backing buffer for the HSTRING is the source string itself, so you must ensure that the source string (as well as the header) remains unchanged for the life of the HSTRING. This approach obviously isn't of any use when you need to return a string to a calling function, but it's a worthy optimization when you need to pass a string as an input to another function whose lifetime is scoped by the stack (not asynchronous functions). WRL also provides wrappers for HSTRING and fast-pass strings.

RoGetActivationFactory is just such a function and is used to get the activation factory or static interface for a given class. This is analogous to the COM CoGetClassObject function. Given that this function is usually used with a const array generated by the MIDL compiler, it makes sense to write a simple function template to provide a fast-pass string wrapper. **Figure 1** illustrates what this might look like.

The GetActivationFactory function template infers the string length automatically, eliminating an error-prone buffer length argument or costly runtime scan. It then prepares a fast-pass string before calling the actual RoGetActivationFactory function. Here, again, the function template infers the interface identifier and safely returns the resulting COM interface pointer wrapped in a WRL ComPtr.

You can now use this helper function to get the ICoreApplication interface:

```
using namespace ABI::Windows::ApplicationModel::Core;
auto app = GetActivationFactory<ICoreApplication>(
  RuntimeClass_Windows_ApplicationModel_Core_CoreApplication);
```

Figure 2 **The SampleWindow QueryInterface Method**

```
auto __stdcall QueryInterface(IID const & id,
                              void ** result) -> HRESULT
{
  ASSERT(result);

  if (id == __uuidof(IFrameworkViewSource) ||
      id == __uuidof(IInspectable) ||
      id == __uuidof(IUnknown))
  {
    *result = static_cast<IFrameworkViewSource *>(this);
  }
  else if (id == __uuidof(IFrameworkView))
  {
    *result = static_cast<IFrameworkView *>(this);
  }
  else if (id == __uuidof(IActivatedEventHandler))
  {
    *result = static_cast<IActivatedEventHandler *>(this);
  }
  else
  {
    *result = nullptr;
    return E_NOINTERFACE;
  }

  // static_cast<IUnknown *>(*result)->AddRef();
  return S_OK;
}
```

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 VERSION 7

Spread Studio for .NET

Download your free trial @

www.componentone.com

ComponentOne®
a division of GrapeCity®

The ICoreApplication interface is what gets the ball rolling for your application. To use this COM interface, you'll need to include the application model header:

```
#include <Windows.ApplicationModel.Core.h>
```

This header defines ICoreApplication, inside the ABI::Windows::ApplicationModel::Core namespace, as well as the CoreApplication's textual class identifier. The only interface method you really need to think about is the Run method.

Before I go on, it's helpful to appreciate how the Windows Runtime brings your application to life. As I've mentioned before, the Windows Runtime considers you only a guest inside your own process. This is analogous to how Windows services have worked for years. In the case of a Windows service, the Windows Service Control Manager (SCM) starts the service using the CreateProcess function, or one of its variants. It then waits for the process to call the StartServiceCtrlDispatcher function. This function establishes a connection back to the SCM whereby the service and the SCM can communicate. If, for example, the service fails to call StartServiceCtrlDispatcher in a timely fashion, the SCM will assume something went wrong and tear down the process. The StartServiceCtrlDispatcher function only returns when the service has ended, so the SCM needs to create a secondary thread for the service to receive callback notifications. The service merely responds to events and is at the mercy of the SCM. As you'll discover, this is remarkably similar to the WinRT application model.

The Windows Runtime waits for the process to get the ICoreApplication interface and call its Run method. Like the SCM, if the process fails to do so in a timely manner, the Windows Runtime assumes that something went wrong and tears down the process. Thankfully, if a debugger is attached, the Windows Runtime notices and disables the timeout, unlike the SCM. However, the model is the same. The Windows Runtime is in charge and calls the application on a runtime-created thread when events occur. Of course, the Windows Runtime is COM-based, so instead of a callback function (as is the case with the SCM) the Windows Runtime—which relies on the Process Lifetime Manager (PLM) for this—expects the application to provide the Run method with a COM interface that it can use to call the application.

Your app must provide an implementation of IFrameworkViewSource, which also hails from the ABI::Windows::ApplicationModel::Core namespace, and CoreApplication will call its solitary CreateView method once it has created your app's UI thread. IFrameworkViewSource doesn't really just have CreateView as a method. It derives from IInspectable, the WinRT base interface. IInspectable in turn derives from IUnknown, the COM base interface.

Figure 3 **The SampleWindow IInspectable Methods**

```
auto __stdcall GetIids(ULONG *,
                       IID **) -> HRESULT
{
  return E_NOTIMPL;
}

auto __stdcall GetRuntimeClassName(HSTRING *) -> HRESULT
{
  return E_NOTIMPL;
}

auto __stdcall GetTrustLevel(TrustLevel *) -> HRESULT
{
  return E_NOTIMPL;
}
```

WRL provides extensive support for implementing COM classes, but I'll save that for an upcoming column. For now, I want to underline how the Windows Runtime really is rooted in COM, and what better way to show that than by implementing IUnknown? For my purposes, it's useful to note that the C++ class that will implement IFrameworkViewSource—and a few others—has its lifetime defined by the stack. Essentially, the application's WinMain function boils down to this:

```
HR(RoInitialize(RO_INIT_MULTITHREADED));
auto app = GetActivationFactory<ICoreApplication>( ...
SampleWindow window;
HR(app->Run(&window));
```

All that remains is to write the SampleWindow class so that it properly implements IFrameworkViewSource. Although CoreApplication doesn't care where they're implemented, at a minimum your app will need to implement not only IFrameworkViewSource but also the IFrameworkView and IActivatedEventHandler interfaces. In this case, the SampleWindow class can just implement them all:

```
struct SampleWindow :
  IFrameworkViewSource,
  IFrameworkView,
  IActivatedEventHandler
{};
```

The IFrameworkView interface is also defined in the ABI::Windows::ApplicationModel::Core namespace, but IActivatedEventHandler is a little harder to pin down. I've defined it myself as follows:

```
using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::ApplicationModel::Activation;

typedef ITypedEventHandler<CoreApplicationView *, IActivatedEventArgs *>
  IActivatedEventHandler;
```

If you have some experience with COM, you might think this looks rather unorthodox—and you'd be right. As you'd expect, ITypedEventHandler is just a class template, and that's a rather odd way to define a COM interface—the most obvious problem being that you couldn't possibly know what interface identifier to attribute it with. Fortunately, all of these interfaces are generated by the MIDL compiler, which takes care to specialize each one, and it's on these specializations that it attaches the GUID representing the interface identifier. As complicated as the previous typedef might appear, it defines a COM interface that derives directly from IUnknown and provides a single method called Invoke.

I have a few interface methods to implement, so let's get started. First up is IUnknown and the mighty QueryInterface method. I don't want to spend too much time on IUnknown and IInspectable here, as I'll be covering them in detail in an upcoming column. **Figure 2** provides a simple implementation of QueryInterface for a stack-based class such as this.

A few things are worth noting about this implementation. First, the method asserts that its arguments are valid. A more politically correct implementation might return E_POINTER, but it's assumed that such errors are bugs that can be resolved during development, so there's no need to waste extra cycles at run time. This gives the best possible behavior by immediately causing an access violation and a crash dump that's pretty easy to analyze. If you return E_POINTER, the broken caller will probably just ignore it. The best policy is to fail early. This is in fact the position taken by many

implementations, including DirectX and the Windows Runtime. A lot goes into implementing QueryInterface correctly. The COM specification is quite specific so that COM classes will always provide certain object identity guarantees correctly and consistently. Don't worry if the chain of *if* statements looks intimidating. I'll cover it in due course.

The final point worth mentioning about this implementation is that it doesn't bother to call AddRef. Ordinarily, QueryInterface must call AddRef on the resulting IUnknown interface pointer before returning. However, because the SampleWindow class resides on the stack, there's no point in reference counting. For the same reason, implementing the IUnknown AddRef and Release methods is straightforward:

```
auto __stdcall AddRef()  -> ULONG { return 2; }
auto __stdcall Release() -> ULONG { return 1; }
```

The results of these methods are only advisory, so you can take advantage of this fact, and any non-zero value will do. A word of caution here: You might want to override operators *new* and *delete* to make it explicit that the class is only designed to work on the stack. Alternatively, you could simply implement reference counting, just in case.

Next, I need to implement IInspectable, but because it won't be used for this simple application, I'll cheat and leave its methods not implemented, as shown in **Figure 3**. This isn't a conforming implementation and isn't guaranteed to work. Again, I'll cover IInspectable in an upcoming column, but this is sufficient to get the SampleWindow IInspectable-derived interfaces up and running.

Next, I need to implement IFrameworkViewSource and its CreateView method. Because the SampleWindow class is also implementing IFrameworkView, the implementation is simple. Again, note that ordinarily you'd need to call AddRef on the resulting IUnknown-derived interface pointer before returning. You might want to call AddRef in the body of this function, just in case:

```
auto __stdcall CreateView(IFrameworkView ** result) -> HRESULT
{
  ASSERT(result);
  *result = this;
  // (*result)->AddRef();
  return S_OK;
}
```

The IFrameworkView interface is where things finally get interesting for the application. After calling CreateView to retrieve the interface pointer from the application, the Windows Runtime calls most of its methods in quick succession. It's important that you respond to these calls quickly, as they all count as time spent by the user waiting for your application to start. The first is called Initialize, and this is where the application must register for the Activated event. The Activated event signals that the application has been activated, but it's up to the application to activate its CoreWindow. The Initialize method is quite simple:

```
auto __stdcall Initialize(ICoreApplicationView * view) -> HRESULT
{
  EventRegistrationToken token;
  HR(view->add_Activated(this, &token));
  return S_OK;
}
```

The SetWindow method is then called, providing the application with the actual ICoreWindow implementation. An ICoreWindow just models a regular desktop HWND inside the Windows Runtime. Unlike the previous application model interfaces, ICoreWindow is defined in the ABI::Windows::UI::Core namespace. Inside the SetWindow method you should just make a copy of the interface pointer, as you'll need it soon enough:

```
using namespace ABI::Windows::UI::Core;

ComPtr<ICoreWindow> m_window;

auto __stdcall SetWindow(ICoreWindow * window) -> HRESULT
{
  m_window = window;
  return S_OK;
}
```

The Load method is next and is where you should stick any and all code to prepare your application for initial presentation:

```
auto __stdcall Load(HSTRING) -> HRESULT
{
  return S_OK;
}
```

At a minimum, you should register for events related to window size and visibility changes, as well as changes to DPI scaling. You might also take the opportunity to create the various DirectX factory objects, load device-independent resources and so on. The reason this is a good spot for all of this is that it's at this point that the user is presented with your application's splash screen.

When the Load method returns, the Windows Runtime assumes your application is ready to be activated and fires the Activated event, which I'll handle by implementing the IActivatedEventHandler Invoke method, like so:

```
auto __stdcall Invoke(ICoreApplicationView *,
                      IActivatedEventArgs *) -> HRESULT
{
  HR(m_window->Activate());
  return S_OK;
}
```

With the window activated, the application is finally ready to run:

```
auto __stdcall Run() -> HRESULT
{
  ComPtr<ICoreDispatcher> dispatcher;
  HR(m_window->get_Dispatcher(dispatcher.GetAddressOf()));

  HR(dispatcher->ProcessEvents(CoreProcessEventsOption_ProcessUntilQuit));
  return S_OK;
}
```

There are many ways to implement this. Here I'm just retrieving the window's ICoreDispatcher interface, which represents the message pump for the window. Finally, there's the Uninitialize method, which might be called occasionally but is otherwise useless and can safely be ignored:

```
auto __stdcall Uninitialize() -> HRESULT
{
  return S_OK;
}
```

And that's it. You can now compile and run the application. Of course, you're not actually painting anything here. You can grab a copy of dx.h from dx.codeplex.com and start adding some Direct2D rendering code (see my June 2013 column, "A Modern Library for DirectX Programming," at msdn.microsoft.com/magazine/dn201741 for more about that), or wait till my next column, where I'll show you how best to integrate Direct2D with the WinRT core application model. ■

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: *James McNellis (Microsoft)*

NEW OPPORTUNITIES WITH NEW DOMAINS

New domains. New opportunities. Create your perfect web address with **over 500 new top-level domains** from 1&1! Short, memorable domains like fashion.blog and auto.shop are ideal for getting your website found easily. Pre-reserve your preferred domain **for free, with no obligation!** With regular updates from 1&1 you don´t miss the chance to get the domains you want. Find out more on our website **1and1.com**

PRE-RESERVE FREE WITH NO OBLIGATION!*

1&1®

MEMBER OF united internet

DOMAINS | E-MAIL | WEB HOSTING | eCOMMERCE | SERVERS

1and1.com

* Pre-reserving a domain name is not a guarantee that you will be able to register that domain. Other terms and conditions may apply. Visit www.1and1.com for full promotional offer details. Program and pricing specifications and availability subject to change without notice. 1&1 and the 1&1 logo are trademarks of 1&1 Internet, all other trademarks are the property of their respective owners.

JULIE LERMAN

Coding for Domain-Driven Design: Tips for Data-Focused Devs

This year, Eric Evans' groundbreaking software design book, "Domain-Driven Design: Tackling Complexity in the Heart of Software" (Addison-Wesley Professional, 2003, amzn.to/ffL1k), celebrates its 10th anniversary. Evans brought to this book many years of experience guiding large businesses through the process of building software. He then spent more years thinking about how to encapsulate the patterns that lead these projects to success—interacting with the client, analyzing the business problems being solved, building teams and architecting the software. The focus of these patterns is the business's domain, and together they comprise Domain-Driven Design (DDD). With DDD, you model the domain in question. The patterns result from this abstraction of your knowledge about the domain. Rereading Martin Fowler's foreword and Evans' preface even today, continues to provide a rich overview of the essence of DDD.

> I'm intrigued and inspired by DDD, but struggle with my data-driven perspective to comprehend some of the technical patterns that make it successful.

In this column and the next two as well, I'll share some pointers that have helped my data-focused, Entity Framework brain gain clarity as I work on getting my code to benefit from some DDD technical patterns.

Why Do I Care About DDD?

My introduction to DDD came from a short video interview on InfoQ.com with Jimmy Nilsson, a respected architect in the .NET community (and elsewhere), who was talking about LINQ to SQL and the Entity Framework (bit.ly/11DdZue). At the end, Nilsson is asked to name his favorite tech book. His reply: "My favorite computer book is the book by Eric Evans, "Domain-Driven Design." It's like poetry, I think. It's not just great content, but you can read it many times and it reads like poetry." Poetry! I was writing my first tech book, "Programming Entity Framework" (O'Reilly Media, 2009), at the time, and I was intrigued by this description. So I went and read a little bit of Evans' book to see what it was like. Evans is a beautiful, fluid writer. And that, combined with his perceptive, naturalistic view of software development, does make the book a joy to read. But I was also surprised by what I was reading. Not only was the writing wonderful, what he was writing about intrigued me. He talked about building relationships with clients and truly understanding their businesses and their business problems (related to the software in question), not just slogging code. This is something that's been important to me in my 25 years of software development. I wanted more.

I tiptoed around the edge of DDD for a few more years, then started learning more—meeting Evans at a conference and then attending his four-day immersion workshop. While I'm far from an expert in DDD, I found that the Bounded Context pattern was something I could leverage right away as I worked to shift my own software creation process toward a more organized, manageable structure. You can read about that topic in my January 2013 column, "Shrink EF Models with DDD Bounded Contexts" (msdn.microsoft.com/magazine/jj883952).

Since then I've explored further. I'm intrigued and inspired by DDD, but struggle with my data-driven perspective to comprehend some of the technical patterns that make it successful. It seems likely that many developers go through the same struggle, so I'm going to share some of the lessons I've been learning with the help, interest, and generosity of Evans and a number of other DDD practitioners and teachers, including Paul Rayner, Vaughn Vernon, Greg Young, Cesar de la Torre, and Yves Reynhout.

When Modeling the Domain, Forget About Persistence

Modeling the domain is all about focusing on the tasks of the business. When designing types and their properties and behaviors, I'm sorely tempted to think about how a relationship will work out in the database and how my object relational mapping (ORM) framework of choice—Entity Framework—will treat the properties, relationships and inheritance hierarchies that I'm building. Unless you're building software for a company whose business is data storage and

Code download available at archive.msdn.microsoft.com/mag201308DataPoints.

retrieval—something like Dropbox—data persistence only plays a supporting role in your application. It's much like making a call out to a weather source's API in order to display the current temperature to a user. Or sending data from your app to an external service, perhaps a registration on Meetup.com. Of course, your data may be more complicated, but with a DDD approach to bounding contexts, focusing on behaviors and following DDD guidance when building types, the persistence can be much less complex than the systems you may be building today.

And if you've studied up on your ORM, such as learning how to configure database mappings using the Entity Framework Fluent API, you should be able to make the persistence work as needed. In the worst case, you may need to make some tweaks to your classes. In an extreme case, such as with a legacy database, you could even add in a persistence model designed for database mapping, then use something such as AutoMapper to resolve things between your domain model and your persistence model.

But these concerns are unrelated to the business problem your software is aimed at solving, so persistence should not interfere with the domain design. This is a challenge for me because as I'm designing my entities, I can't help but consider how EF will infer their database mappings. And so I try to block out that noise.

Private Setters and Public Methods

Another rule of thumb is to make property setters private. Instead of allowing calling code to randomly set various properties, you should control interaction with DDD objects and their related data using methods that modify the properties. And, no, I don't mean methods like SetFirstName and SetLastName. For example, instead of instantiating a new Customer type and then setting each of its properties, you might have some rules to consider when creating a new customer. You can build those rules into the Customer's constructor, use a Factory Pattern method or even have a Create method in the Customer type. **Figure 1** shows a Customer type that's defined following the DDD pattern of an aggregate root (that is, the "parent" of a graph of objects, also referred to as a "root entity" in DDD). Customer properties have private setters so that only other members of the Customer class can directly affect those properties. The class exposes a constructor to control how it's instantiated and hides the parameter-less constructor (required by Entity Framework) as internal.

Figure 1 **Properties and Methods of a Type That Acts As an Aggregate Root**

```
public class Customer : Contact
{
  public Customer(string firstName,string lastName, string email)
  { ... }

  internal Customer(){ ... }

  public void CopyBillingAddressToShippingAddress(){ ... }

  public void CreateNewShippingAddress(
   string street, string city, string zip) { ... }

  public void CreateBillingInformation(
   string street, string city, string zip,
   string creditcardNumber, string bankName){ ... }

  public void SetCustomerContactDetails(
   string email, string phone, string companyName){ ... }

  public string SalesPersonId { get; private set; }
  public CustomerStatus Status{get;private set;}
  public Address ShippingAddress { get; private set; }
  public Address BillingAddress { get;private set; }
  public CustomerCreditCard CreditCard { get; private set; }
}
```

The Customer type controls and protects the other entities in the aggregate—some addresses and a credit-card type—by exposing specific methods (such as CopyBillingAddressToShippingAddress) with which those objects will be created and manipulated. The aggregate root must make sure the rules that define each entity within the aggregate are applied using domain logic and behavior implemented in these methods. Most important, the aggregate root is in charge of invariant logic and consistency throughout the aggregate. I'll talk more about invariants in my next column, but for the meantime, I recommend reading Jimmy Bogard's blog post, "Strengthening Your Domain: Aggregate Construction," at bit.ly/ewNZ52, which provides an excellent explanation of invariants in aggregates.

> Not everything in your app needs to be created using DDD.

In the end, what's exposed by Customer is behavior rather than properties: CopyBillingAddressToShippingAddress, CreateNewShippingAddress, CreateBillingInformation and SetCustomerContactDetails.

Note that the Contact type, from which Customer derives, lives in a different assembly named "Common" because it may be needed by other classes. I need to hide the properties of Contact, but they can't be private or Customer wouldn't be able to access them. Instead, they're scoped as Protected:

```
public class Contact: Identity
{
  public string CompanyName { get; protected set; }
  public string EmailAddress { get; protected set; }
  public string Phone { get; protected set; }
}
```

A side note about Identities: Customer and Contact may look like DDD value objects because they have no key value. However, in my solution, the key value is provided by the Identity class from which Contact derives. And neither of these types are immutable, so they can't be considered value objects anyway.

Because Customer inherits from Contact, it will have access to those protected properties and is able to set them, as in this SetCustomerContactDetails method:

```
public void SetCustomerContactDetails
  (string email, string phone, string companyName)
{
  EmailAddress = email;
  Phone = phone;
  CompanyName = companyName;
}
```

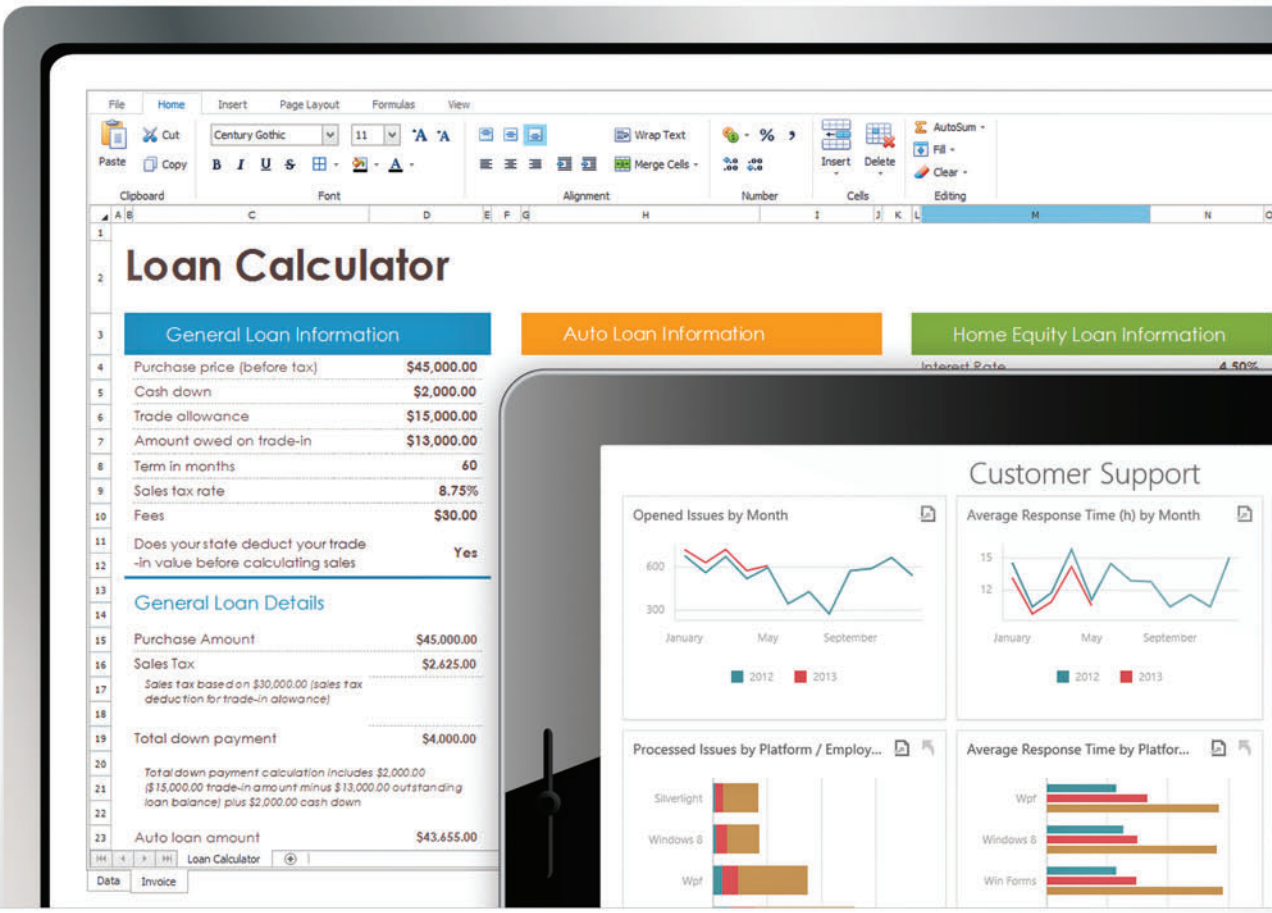
Sometimes All You Need Is CRUD

Not everything in your app needs to be created using DDD. DDD is there to help handle complex behaviors. If you just need to do some raw, random editing or querying, then a simple class (or set

DevExpress™

When only the best will do.

High-Performance, Elegant and Feature-Complete .NET Controls and Libraries



Download your FREE 30-day trial today. **DevExpress.com/try**

WIN WinForms · ASP ASP.NET · WPF WPF · SL Silverlight · Windows 8 XAML · HTML JS

of classes), defined just as you'd typically do with EF Code First (using properties and relationships) and combined with insert, update and delete methods (via a repository or just DbContext), is all you need. So, to accomplish something like creating an order and its line items, you might want DDD to help work through special business rules and behaviors. For example, is this a Gold Star customer placing the order? In that case, you need to get some customer details to determine if the answer is yes, and, if so, apply a 10 percent discount to each item being added to the order. Has the user provided their credit-card information? Then you might need to call out to a verification service to ensure it's a valid card.

The key in DDD is to include the domain logic as methods within the domain's entity classes, taking advantage of OOP instead of implementing "transactional scripts" within stateless business objects, which is what a typical demo-ware Code First class looks like.

> I'm learning to really consider where I need to share data, and then pick my battles.

But sometimes all you're doing is something very standard, like creating a contact record: name, address, referred by, and so forth, and saving it. That's just create, read, update and delete (CRUD). You don't need to create aggregates and roots and behaviors to satisfy that.

Most likely your application will contain a combination of complex behaviors and simple CRUD. Take the time to clarify the behaviors and don't waste time, energy and money over-architecting the pieces of your app that are really just simple. In these cases, it's important to identify boundaries between different subsystems or bounded contexts. One bounded context could be very much data-driven (just CRUD), while a critical core-domain-bounded context should, on the other hand, be designed following DDD approaches.

Shared Data Can Be a Curse in Complex Systems

Another issue I banged my head on, then ranted and whined about as people kindly tried to explain further, concerned sharing types and data across subsystems. It became clear that I couldn't "have my cake and eat it too," so I was forced to think again about my assumption that I absolutely positively must share types across systems and have those types all interact with the same table in the same database.

I'm learning to really consider where I need to share data, and then pick my battles. Some things just may not be worth trying, like mapping from different contexts to a single table or even a single database. The most common example is sharing a Contact that's trying to satisfy everyone's needs across systems. How do you reconcile and leverage source control for a Contact type that might be needed in numerous systems? What if one system needs to modify the definition of that Contact type? With respect to an ORM, how do you map a Contact that's used across systems to a single table in a single database?

DDD guides you away from sharing domain models and data by explaining that you don't always need to point to the same person table in a single database.

My biggest push back with this is based on 25 years of focusing on the benefits of reuse—reusing code and reusing data. So, I have a hard time with the following idea but I'm warming up to it: *It isn't a crime to duplicate data.* Not all data will fit into this new (to me) paradigm, of course. But what about something lightweight like a person's name? So what if you duplicate a person's first and last name in multiple tables or even multiple databases that are dedicated to different sub-systems of your software solution? In the long run, by letting go of the complexity of sharing data, you make the job of building your system much simpler. In any case, you must always minimize data and attribute duplication in different bounded contexts. Sometimes you just need the customer's ID and status in order to calculate discounts in a Pricing-bounded context. That same customer's first and last name might be needed only in the Contact Management-bounded context.

But there's still so much information that needs to be shared between systems. You can leverage what DDD refers to as an "anti-corruption layer" (which can be something as simple as a service or a message queue) to ensure that, for example, if someone creates a new contact in one system, you either recognize that the person already exists elsewhere, or ensure that the person, along with a common identity key, is created in another subsystem.

Plenty to Chew on Until Next Month

As I make my way through learning and comprehending the technical side of Domain-Driven Design, struggling to reconcile old habits with new ideas, and arriving at innumerable "aha!" moments, the pointers I discussed here are truly ones that helped me see more light than darkness. Sometimes it's just a matter of perspective, and the way I have expressed them here reflects the perspective that helped make things clearer to me.

I'll share some more of my "aha!" moments in my next column, where I'll talk about that condescending term you may have heard: "anemic domain model," along with its DDD cousin, the "rich domain model." I'll also discuss unidirectional relationships and what to expect when it's time to add in data persistence if you're using Entity Framework. I'll also touch on some more DDD topics that caused me plenty of grief in an effort to shorten your own learning curve.

Until then, why not take a closer look at your own classes and see how to be more of a control freak, hiding those property setters and exposing more descriptive and explicit methods. And, remember: no "SetLastName" methods allowed. That's cheating! ■

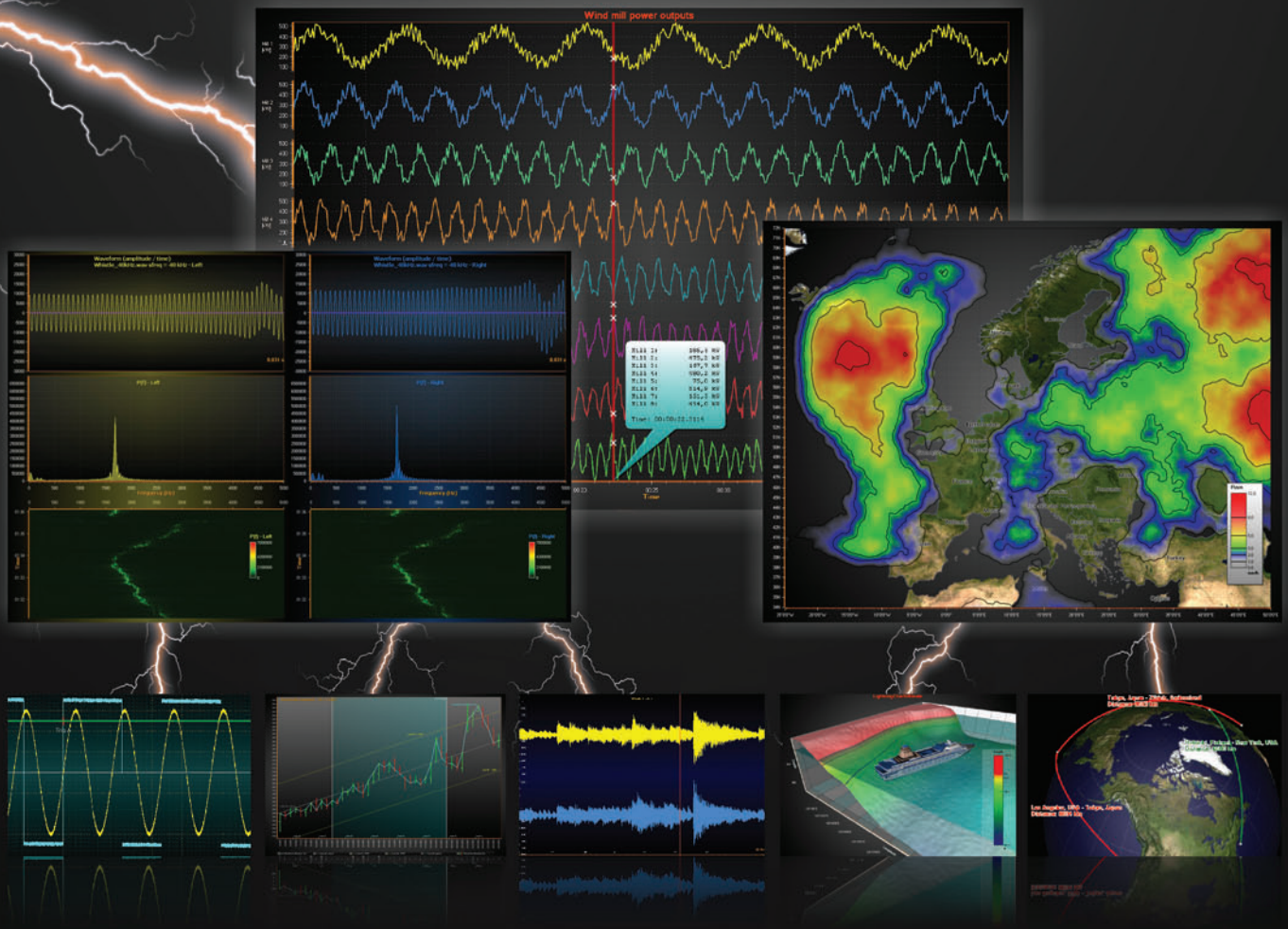
Julie Lerman *is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other Microsoft .NET topics at user groups and conferences around the world. She blogs at thedatafarm.com/blog and is the author of "Programming Entity Framework" (2010) as well as a Code First edition (2011) and a DbContext edition (2012), all from O'Reilly Media. Follow her on Twitter at twitter.com/julielerman and see her Pluralsight courses at juliel.me/PS-Videos*

Thanks to the following technical expert for reviewing this article: *Cesar de la Torre (Microsoft)*

ULTIMATE CHARTING POWER

LightningChart

The fastest rendering data visualization components for WPF and WinForms



- **Fully DirectX accelerated**
- **Superior 2D and 3D rendering performance**
- **Very extensive property sets**
- **Optimized for real-time data monitoring**
- **Supports gigantic data sets**
- **Professional, friendly and fast customer support**
- **Compatible with Visual Studio 2005...2012**

WPF charts performance comparison

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

Winforms charts performance comparison

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

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

FREE TRIAL

Download a free 30-day evaluation from

www.LightningChart.com

Arction

Pioneers of high-performance data visualization

Bringing RESTful Services to C++ Developers

Sridhar Poduri

In this article I'll show how to use the C++ REST SDK to build a simple Windows-based client application that uploads a file to Dropbox, along with a standard C++ class that supports OAuth.

The world of connected devices is increasingly moving into a heterogeneous mixture of native apps, Web apps and Web sites all connecting to a mix of cloud- and non-cloud-based services. These apps share and consume data through this mishmash of devices and services and provide rich, immersive experiences for end users. These users are increasingly using apps installed on their personal devices (smartphones, tablets, PCs and other similar devices). Developers are tasked with creating seamless experiences in those apps to facilitate sharing, organizing and viewing data—and to delight users.

Apps are written in a variety of programming languages. Some, such as PHP, C#, and Java, are by default Web-enabled and might meet the needs of some of the common scenarios for sharing and consuming data. In other cases, specialized libraries might be utilized to perform server-side computing, processing, and analysis, and return a response to the client app.

In his paper, "Welcome to the Jungle" (bit.ly/uhfrzH), Herb Sutter unambiguously describes the mainstream move to elastic, distributed computing as a natural evolution of the same scale and performance trends that have driven multicore and heterogeneous computing into the mainstream.

This article discusses:

- The advantages of using the C++ REST SDK
- Client classes in the C++ REST SDK
- Using the Dropbox API
- Comparing the code with C-style APIs

Technologies discussed:

C++ REST SDK

Code download available at:

archive.msdn.microsoft.com/mag201308CPP

C++ is the preferred programming language when cross-platform code sharing and performance are design considerations. When it comes to cloud computing, though, C++ seemingly doesn't deliver on its full potential. One of the primary reasons cited for this apparent failure is the lack of a cross-platform, high-performance, efficiently scalable library that would enable a C++ developer to seamlessly integrate existing C and C++ code into the cloud environment.

The C++ REST SDK (bit.ly/VI67I5) is Microsoft's first foray into enabling native code to move to the cloud environment. It aims to provide developers with tools and APIs that solve day-to-day problems of moving native code to the cloud. The initial release is a client-side, cross-platform library to access REST services. The goal is to make the C++ REST SDK a truly cross-platform library that can unlock the true potential of native code in the cloud. The C++ REST SDK can be the basis of building more specialized libraries such as those to support Windows Azure services authoring, and it will enable lightweight, high-performance Web services to be written in portable C++ code without the need for additional frameworks or runtimes to be deployed on cloud-based virtual machines (VMs).

Why Choose the C++ REST SDK?

For accessing REST services, a C++ developer can build abstractions over the C-style APIs such as WinINet or WinHTTP on Windows and similar APIs on other platforms. Given such options, one question that calls for an immediate answer is: Why should a developer choose the C++ REST SDK?

The C++ REST SDK is designed and written from the ground up using modern C++. Features include:

- Support for accessing REST-based services from native code on Windows Vista, Windows 7, Windows 8, Windows Store apps, and Linux by providing asynchronous bindings to HTTP, JSON, XML, URIs, and so on.
- A Visual Studio Extension SDK that allows the consumption of the SDK in Windows Store apps.
- A consistent and powerful programming model for composing asynchronous operations based on standard C++11 features.

- An implementation of asynchronous streams and stream buffers that can be used to read and write to file/device streams.

The C++ REST Client Classes

C++ REST is built on the premise of using modern C++ and asynchronous programming patterns. For my experiments with Dropbox, I've used the http_client class, the Task classes and the asynchronous streams class. I'll discuss each.

The http_client Class As the name implies, the http_client class from the web::http namespace is used to set up and maintain a connection to an HTTP Web service. If you create an instance of the http_client and a URI to the service endpoint, the object instance can be used to make requests on behalf of the client. Asynchrony is built in to the C++ REST library, so responses are returned by the library as tasks.

The Task Class A task represents an operation that could potentially finish when the function producing said task has already returned. Not all tasks run to completion, and there's no guarantee in the order of completion, either. Each task object has a member function, is_done, which returns a bool. When the task has run to completion, is_done returns true; otherwise, it returns false. Once a task has run to completion—as indicated by the Boolean value returned by the is_done member function—calling the *get* function on the task returns the value out of the task. Beware of calling the get function when the is_done function returns false. Doing so will block the thread and defeat the whole purpose of building asynchronous patterns in code.

Instead of continuously checking for is_done, it's better to use the *then* function. It relies on attaching a handler function to the task, similar to promises in JavaScript extensions for Windows Store apps. It should be easily identifiable for developers using the PPL tasks used for programming Windows Runtime (WinRT) asynchronous operations. The handler function passed to the then function should take an argument either of type T or task<T>. Using the argument of type task<T> bestows an additional benefit: It's the only way to catch exceptions raised by the task operation itself!

Because the handler for the then function is invoked only after the task is completed, calling the get function inside the handler is safe and doesn't block the thread.

The Asynchronous Streams The C++ REST library includes a set of helper classes to read and write to objects encapsulated as streams and stream buffers. Following the pattern and precedence set in the standard C++ library, streams and buffers in C++ REST separate the concern of formatting data for input and output from the concern of writing and reading bytes or collections of bytes to and from some underlying medium such as a TCP socket, a disk file or even a memory buffer. In a way, streams are disconnected from the underlying medium used to read and write data. The big difference with streams in C++ REST is that they support asynchronous read and write operations, unlike standard C++ classes, which are blocking. As is the design with other C++ REST objects, the asynchronous methods in the stream classes return a task<T> instead of a value.

With this primer on C++ REST, it's now time to think about the Dropbox REST API. In the rest of this article, I'll discuss accessing the Dropbox REST API using C++ REST to upload a file from the local machine running Windows to the user's Dropbox folder.

The Dropbox REST API

Dropbox uses OAuth version 1 to authenticate all requests to its API (bit.ly/ZJLP4o) and requires that all requests are made over SSL. An Internet search for a standard C++ library supporting OAuth returns only the OAuth library, liboauth, and it requires either OpenSSL (bit.ly/BpfcH) or Mozilla Network Security Services (NSS) (mzl.la/abU77o). I wanted a lightweight, cross-platform class that supports OAuth, so I set out to build one to support Dropbox authentication.

For curious folks, Brook Miles has a great series of posts on accessing Twitter OAuth in C++ from Win32 (bit.ly/137Ms6y). I've built upon his basic ideas but refactored the code to work with the Dropbox APIs, using the standard C++ types supported by C++ REST as much as possible. In addition, I didn't intend to use either WinINet or WinHTTP to perform any Web requests, as that would tie me to the Windows platform only and force me to use a C-style API.

Now I'll discuss building a simple C++ class that supports OAuth and a Win32 application using C++ REST that makes calls to Dropbox and uploads a file to Dropbox. In a follow-up article, I'll show how to use C++ REST from a Windows Store app and upload files to Dropbox.

Before I begin writing an application that can access Dropbox, I need to register it with Dropbox. This is done at the Dropbox Apps console portal (bit.ly/R14tjq). I signed up for a free Dropbox

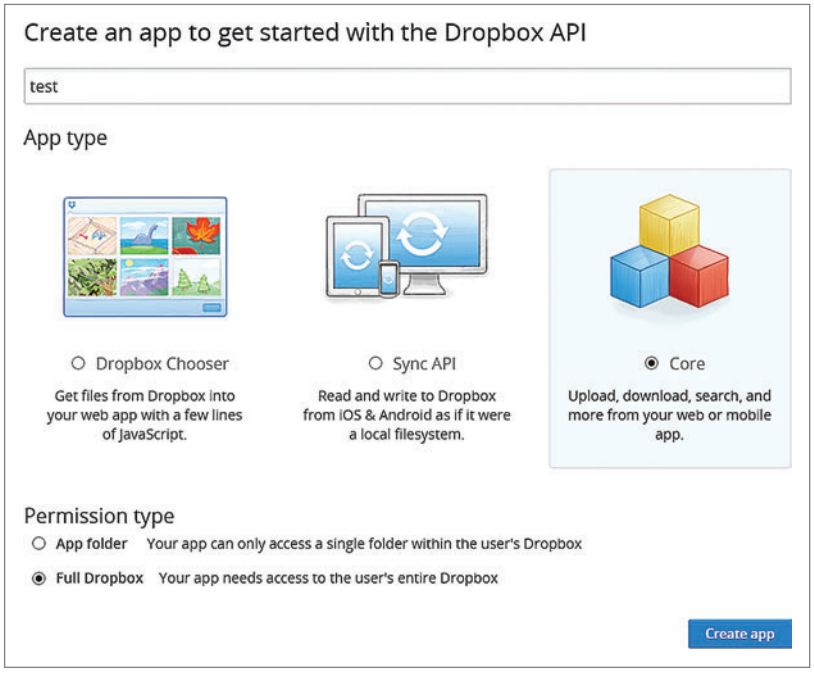


Figure 1 **Choices for Creating a Dropbox App**

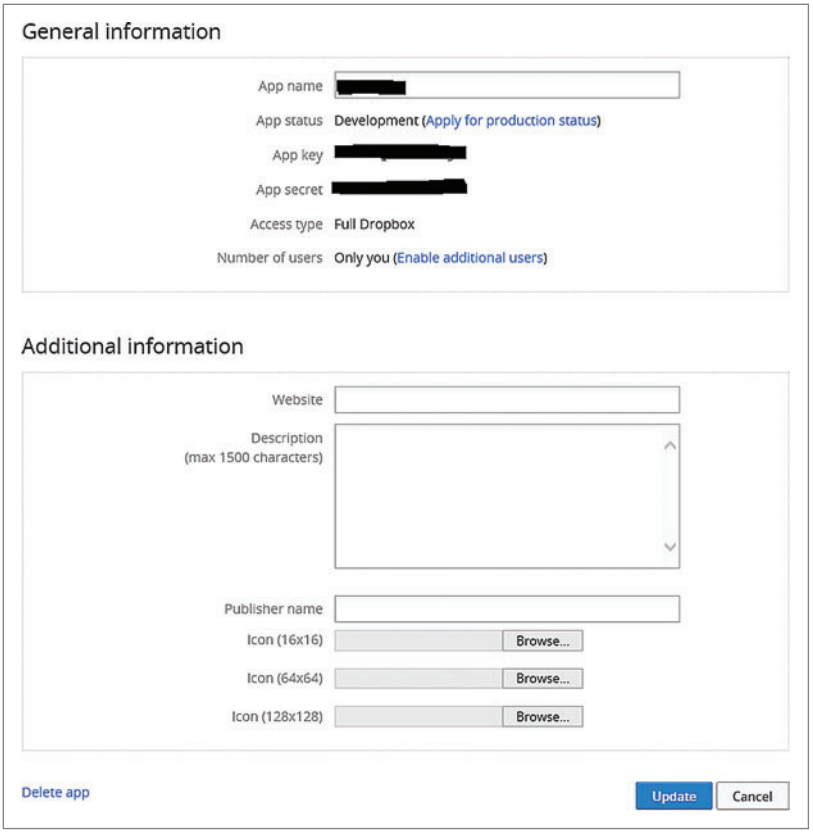


Figure 2 **Choosing Dropbox App Details**

account, logged into the Apps console portal and created a new app by clicking on the "Create app" button.

The process asks you to choose an app name, app type and a permission type. I entered "test" and chose the app type Core (see **Figure 1**). The other app types are Dropbox Chooser (useful for JavaScript developers) and Sync API (best for iOS and Android developers). Finally, I chose the permission type as Full Dropbox, as this gives me the flexibility of reading, writing and syncing to any folder on Dropbox versus a specific folder, which is provided by the "sandbox" App folder permission type.

Upon clicking the "Create app" button, Dropbox creates the app and provides details needed for access, such as the app key, app secret and so on, as shown in **Figure 2**. I made note of these because I need them to perform programmatic authentication and authorization requests using OAuth.

Figure 3 **Building the oAuth Class**

```
// Dropbox consumer key and secret
const std::wstring consumerKey = L"Your app key";
const std::wstring consumerSecret = L"Your app secret";

// List of Dropbox authenticate, authorize, request and file upload URIs
const std::wstring DropBoxRequestTokenURI =
  L"https://api.dropbox.com/1/oauth/request_token";
const std::wstring DropBoxAuthorizeURI =
  L"https://www.dropbox.com/1/oauth/authorize";
const std::wstring DropBoxAccessTokenURI =
  L"https://api.dropbox.com/1/oauth/access_token";
const std::wstring DropBoxFileUploadURI =
  L"https://api-content.dropbox.com/1/files_put/dropbox/<your file name here>";
const std::wstring LocalFiletoUpload = L"Your local file goes here";
```

A Simple Cross-Platform C++ Class for OAuth

Now for the real action! As I said earlier, I wanted to write a cross-platform class that could be used across both Windows and Linux. The current release of the C++ REST SDK doesn't support making HTTPS requests from a Linux box. This is a disappointment, and I've heard that full HTTPS support for Linux is coming soon. The C++ class I discuss here should work across both Windows and Linux without major modifications. In order to support OAuth-based authentication for Dropbox, I had to meet the following main requirements (see the Dropbox site for full requirements):

- All requests should be made over SSL. This means using only HTTPS, versus HTTP.
- OAuth requires that request URIs and parameters be signed using either HMAC-SHA1 or RSA-SHA1 encryption, or plain text if the request is performed over SSL.

For the purpose of my experiments, I settled on using plain-text signature transport and making requests via HTTPS. Building an encryption API that works across both Windows and Linux is complex and time-consuming, and is worth a detailed exploration later.

Once I settled on the requirements, it was time to build the class. I declared a high-level namespace, conveniently called Authentication, and a class inside named oAuth. At the namespace level, I had a few const strings declared for the URI endpoints, the app key and app secret for the app obtained in the app registration process, and a few helper methods, as shown in **Figure 3**.

The entire OAuth protocol support is implemented in the BuildSignedOAuthParameters method of the oAuth class. This method accepts the endpoint URI, the HTTP method type (GET, POST, PUT and so on), the app key, app secret, request token, and token secret, and builds a signature that should be sent across to Dropbox with each request. Dropbox attempts to build the exact signature on its end using the parameters passed with the HTTPS request and match the signatures generated. If the signatures don't match, it returns an HTTP error code.

A signature is built using a random number—called a nonce in OAuth parlance—including a time stamp for the request, the version of OAuth protocol supported, the signature type and more. The method returns a list of all the required parameters, sorted by name and with the signature URL encoded (see **Figure 4**).

With support for OAuth out of the way, I wrote the client code to access files on Dropbox. I created four methods:

1. **oAuthLoginAsync:** Performs the login to Dropbox using the app key and secret.
2. **AuthorizeDropBoxAccess:** Launches Internet Explorer and authorizes app access to Dropbox. This method is specific to Windows and launches Internet Explorer irrespective of whether it's the default browser.

3. **oAuthAcquireTokenAsync:** Performs the action to acquire the actual Dropbox access token.
4. **UploadFileToDropBoxAsync:** Uploads a file from the local system to Dropbox cloud storage.

Each of these operations is made extremely easy and seamless using the C++ REST client classes.

The Client Code

So how does client code written using modern C++ with asynchronous tasks match up against using a C-style API? It's time to find out.

With a C-style API such as WinINet, I would've had to make the following WinINet API calls to get my app working:

- Manually build the HTTP request header.
- Call InternetCrackUrl to resolve the REST endpoint URL.
- Call InternetOpen and obtain a handle to the Internet connection. Typically this is returned as an HINTERNET instance.
- Once I had a valid HINTERNET handle, make a call to HttpOpenRequest, which returns another instance of HINTERNET.
- Make the next call to HttpAddRequestHeaders, which returns a Boolean value indicating if the header information has been added successfully to the HTTP request.
- Once I successfully completed all the preceding steps with the appropriate error handling put in place, make the call to HttpSendRequest, which sends the actual request.
- Upon receiving a response to the previous request, make another call to InternetReadFile to read the response stream.

Please note that all of the previous APIs are C-style APIs with no support for modern C++ programming idioms such as shared pointers, lambdas and asynchronous patterns built in.

Now for the actual code using the C++ REST SDK. **Figure 5** shows the oAuthLoginAsync function, which performs the login

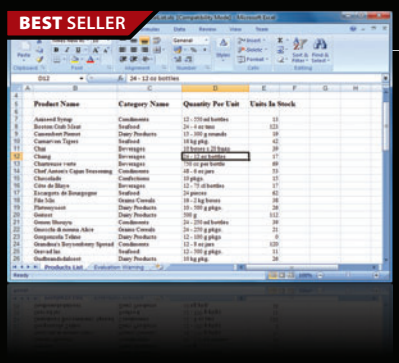
Figure 4 **Building a Signature**

```
HTTPParameters BuildSignedOAuthParameters(
  const HTTPParameters& requestParameters,
  const std::wstring& url,
  const std::wstring& httpMethod,
  const HTTPParameters* postParameters,
  const std::wstring& consumerKey,
  const std::wstring& consumerSecret,
  const std::wstring& requestToken = L"",
  const std::wstring& requestTokenSecret = L""
  )
{
  std::wstring timestamp = OAuthCreateTimestamp();
  std::wstring nonce = OAuthCreateNonce();

  m_oauthParameters[L"oauth_timestamp"] = timestamp;
  m_oauthParameters[L"oauth_nonce"] = nonce;
  m_oauthParameters[L"oauth_version"] = L"1.0";
  m_oauthParameters[L"oauth_signature_method"] = L"PLAINTEXT";
  m_oauthParameters[L"oauth_consumer_key"] = consumerKey;

  // Add the request token if found
  if (!requestToken.empty())
  {
    m_oauthParameters[L"oauth_token"] = requestToken;
  }


  // Create a parameter list containing both oauth and original
  // parameters; this will be used to create the parameter signature
  HTTPParameters allParameters = requestParameters;
  if(Compare(httpMethod, L"POST", false) && postParameters)
  {
    allParameters.insert(postParameters->begin(), postParameters->end());
  }
  allParameters.insert(m_oauthParameters.begin(), m_oauthParameters.end());

  // Prepare a signature base, a carefully formatted string containing
  // all of the necessary information needed to generate a valid signature
  std::wstring normalUrl = OAuthNormalizeUrl(url);
  std::wstring normalizedParameters =
    OAuthNormalizeRequestParameters(allParameters);

  std::wstring signatureBase =
    OAuthConcatenateRequestElements(httpMethod,
    normalUrl,
    normalizedParameters);

  // Obtain a signature and add it to header requestParameters
  std::wstring signature = OAuthCreateSignature(signatureBase,
    consumerSecret,
    requestTokenSecret);

  m_oauthParameters[L"oauth_signature"] = UrlEncode(signature);

  return m_oauthParameters;
}
```

Figure 5 **The oAuthLoginAsync Function**

```
task<void> oAuthLoginAsync(std::shared_ptr<app_credentials>& creds)
{
  uri url(DropBoxRequestTokenURI);

  std::shared_ptr<oAuth> oAuthObj = std::make_shared<oAuth>();

  auto signatureParams =
   oAuthObj->CreateOAuthSignedParameters(url.to_string(),
   L"GET",
   NULL,
   consumerKey,
   consumerSecret
   );

  std::wstring sb = oAuthObj->OAuthBuildSignedHeaders(url);

  http_client client(sb);

  // Make the request and asynchronously process the response
  return client.request(methods::GET)
    .then([&creds](http_response response)
  {
    if(response.status_code() != status_codes::OK)
    {
      // Handle error cases ...
      return pplx::task_from_result();
    }

    // Perform actions here reading from the response stream ...
    // in this example, parse the response body and
    // extract the token and token secret
    istream bodyStream = response.body();
    container_buffer<std::string> inStringBuffer;
    return bodyStream.read_to_end(inStringBuffer)
      .then([inStringBuffer, &creds](pplx::task<size_t> previousTask)
    {
      const std::string &text = inStringBuffer.collection();

      // Convert the response text to a wide-character
      // string and then extract the tokens
      std::wstring_convert
        <std::codecvt_utf8_utf16<wchar_t>, wchar_t> utf16conv;
      std::wostringstream ss;
      std::vector<std::wstring> parts;
      ss << utf16conv.from_bytes(text.c_str()) << std::endl;
      Split(ss.str(), parts, '&', false);
      unsigned pos = parts[1].find('=');
      std::wstring token = parts[1].substr(pos + 1, 16);

      pos = parts[0].find('=');
      std::wstring tokenSecret = parts[0].substr(pos + 1);
      creds->set_Token(token);
      creds->set_TokenSecret(tokenSecret);
    });
  });
}
```

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

BEST SELLER

GdPicture.NET | from $3,919.47

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
- Includes sample code for: .NET, VB6, Delphi, VC++, C++ Builder, VFP, HTML, Access...
- 100% royalty-free and world leading Imaging SDK

BEST SELLER

ComponentOne Studio Enterprise | 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
- Supports Visual Studio 2012 and Windows 8
- Now includes MVC 4 Scaffolding and new MVC 4 project templates (C# & VB)
- New Tile controls for WinForms, WPF, Silverlight, WinRT and Windows Phone
- Royalty-free deployment and distribution

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

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

Figure 6 **The UploadFileToDropBoxAsync Function**

```
task<void> UploadFileToDropBoxAsync(std::shared_ptr<app_credentials>& creds)
{
  using concurrency::streams::file_stream;
  using concurrency::streams::basic_istream;


  uri url(DropBoxFileUploadURI);

  std::shared_ptr<oAuth> oAuthObj = std::make_shared<oAuth>();

  auto signatureParams =
    oAuthObj->CreateOAuthSignedParameters(url.to_string(),
    L"PUT",
    NULL,
    consumerKey,
    consumerSecret,
    creds->Token(),
    creds->TokenSecret()
    );

  std::wstring sb = oAuthObj->OAuthBuildSignedHeaders(url);

  return file_stream<unsigned char>::open_istream(LocalFiletoUpload)
    .then([sb, url](pplx::task<basic_istream<unsigned char>> previousTask)
  {
    try
    {
      auto fileStream = previousTask.get();
      // Get the content length, which is used to set the
      // Content-Length property
      fileStream.seek(0, std::ios::end);
      auto length = static_cast<size_t>(fileStream.tell());
      fileStream.seek(0, 0);

      // Make HTTP request with the file stream as the body
      http_request req;
      http_client client(sb);
      req.set_body(fileStream, length);
      req.set_method(methods::PUT);

      return client.request(req)
        .then([fileStream](pplx::task<http_response> previousTask)
      {
        fileStream.close();

        std::wostringstream ss;
        try
        {
          auto response = previousTask.get();
          auto body = response.body();
          ss << L"Server returned returned status code "
            << response.status_code() << L"."
            << std::endl;
          std::wcout << ss.str();
        }
        catch (const http_exception& e)
        {
          ss << e.what() << std::endl;
          }
        std::wcout << ss.str();
      });

      }
      catch (const std::system_error& e)
      {
        std::wostringstream ss;
        ss << e.what() << std::endl;
        std::wcout << ss.str();

        // Return an empty task
        return pplx::task_from_result();
      }
    });
  }
```

operation to Dropbox and the UploadFileToDropBoxAsync function that uploads a file from the local system to Dropbox.

In the oAuthLoginAsync function, I first construct a URI instance from a string representation of the Login URI endpoint. Next, I create an instance of the oAuth class and call the member function CreateOAuthSignedParameters, which builds a map containing all the necessary OAuth request parameters. Finally, I sign the headers by calling the OAuthBuildSignedHeaders member function. Signing of headers is mandatory as per the OAuth specification. The HTTP communication begins now. I need only create an instance of an http_client and pass it the signed request string. Dropbox will use the request string and header information and attempt to build the same request string on the server side and match it against what I send as part of the HTTP request. If the strings match, I get a success return code; otherwise, I get an error.

I begin the communication process by creating an instance of http_client class and then call the request member function. I specify the HTTP method as GET. When the request method finishes execution, it returns an http_response object that I use to parse and extract the token and token secret, which are stored in the app_credentials class instance. The token should be sent along with all subsequent API requests to Dropbox.

The UploadFileToDropBoxAsync function is shown in **Figure 6**. It follows a similar pattern to the oAuthLoginAsync function until I build the signed OAuth headers. Once I build the header information, I create a task that reads a file from the local file system into a file_stream object and sets that file_stream object as the HTTP request body. I can then create an instance of an http_client class and set the request instance, which contains the file_stream contents as the body, and place the PUT request. Upon completion, I get a task that contains an http_response that can be parsed for success or failure. It really is this simple.

When compared to using a C-style API for Web communication or going down the route of building support using platform-specific APIs such as WinINet, the code written using modern C++ is more succinct, readable and elegant. As a matter of fact, all of the low-level implementation details are abstracted away from the public interface of the library. The C++ REST SDK builds on the promise of modern C++ and applies the same design principles for RESTful communication. The end result is an extremely well-designed and patterned library using modern C++ that makes the process of building connected apps easy and seamless.

Next: A Windows Store App

In this article I've explored how to build a simple Windows-based client application using the C++ REST SDK that uploads a file to Dropbox. Along the way, I've also discussed building a standard C++ class that supports OAuth. In a follow-up article, I'll show how to build a Windows Store app using the C++ REST SDK. Stay tuned! ■

Sridhar Poduri *is a program manager in the Windows team at Microsoft. A C++ aficionado and author of the book, "Modern C++ and Windows Store Apps" (Sridhar Poduri, 2013), he blogs regularly about C++ and the Windows Runtime at sridharpoduri.com.*

Thanks to the following technical expert for reviewing this article: *Artur Laksberg (Microsoft)*

Real-Time, Realistic Page Curling with DirectX, C++ and XAML

Eric Brumer

During the development of Windows 8 and Visual Studio 2012, the Microsoft C++ team created some open source apps to showcase the various C++ technologies available to software developers. One of these apps is Project "Austin," a digital note-taking app written in C++, using DirectX and XAML on the Windows Runtime (WinRT).

In this app, a user can create a notebook and jot down some notes or scribble diagrams. There's support for adding and deleting pages, different ink colors, and adding image files from a PC or from SkyDrive. **Figure 1** shows some screenshots of the app in action.

Users can view their notebooks in three ways: a single row of pages (as in **Figure 1**), a grid of pages or as if the pages were stacked on top of one another. In this stacked view, the user can flip through pages by swiping his finger across the page, as if he were flipping through pages in a real book. The digital pages are curled in real time based on the position of the user's finger as he flips the page. **Figure 2** shows the page curling in action.

The page-curling feature also handles page uncurling. When the user lets go of a page while curling, the page acts like a real piece of paper: if the page is below a certain threshold, it uncurls back to a lay-flat position; if the page is above the threshold, it uncurls but finishes turning.

This article describes in depth the geometry, technologies and code used to perform real-time page curling and uncurling.

This article discusses:

- The geometry of page curling
- Page curling architecture
- Ensuring smooth animation
- Handling user input
- Achieving necessary performance

Technologies discussed:

Windows 8, C++, DirectX, XAML

The Geometry of Page Curling

Before exploring the overall design, I'll get the geometry and math out of the way. This information is largely taken from my MSDN blog post, "Project Austin Part 2 of 6: Page Curling" (bit.ly/THF40f).

The 2006 paper, "Turning Pages of 3D Electronic Books" (L. Hong, S.K. Card and J. Chen), describes how page curling can be simulated by deforming the paper around an imaginary cone, as shown in **Figure 3**. By changing the shape and position of the cone you can simulate more (or less) curling.

Similarly, page curling can also be simulated by deforming the paper around an imaginary cylinder, as shown in **Figure 4**.

My method for page curling is as follows:

- If the user is curling from the top-right of the page, deform the page around a *cone with angle θ and apex at coordinates (0, Ay, 0)*.

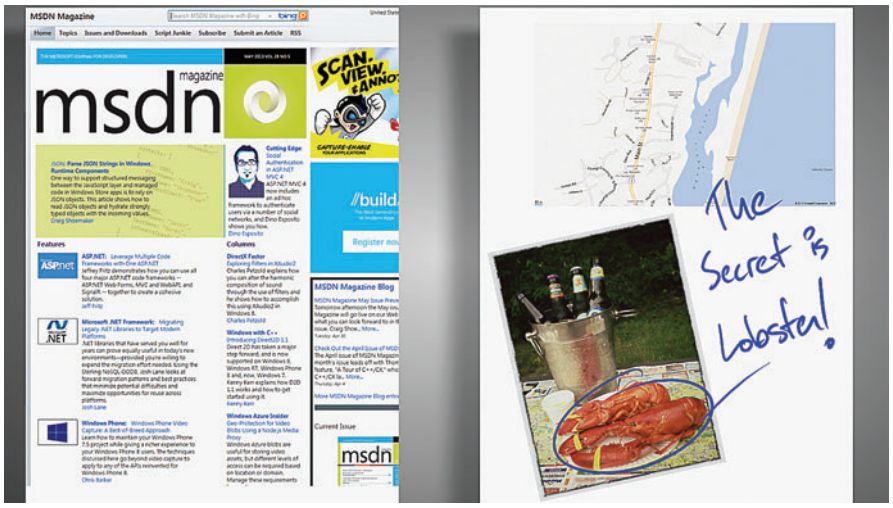


Figure 1 **Project "Austin"**

- If the user is curling from the center-right of the page, deform the page around a *cylinder with radius r*.
- If the user is curling from the bottom-right of the page, deform the page around an *inverted cone*.
- If the user is curling anywhere in between, deform the page *around the linear combination of a cone and a cylinder, based on the y-coordinate of the input*.
- After deforming, *rotate the paper around the y axis*.

Here are the details to transform a page around a cylinder. (The Hong article describes similar geometry to transform a page around a cone.) Given the point P_{flat} with coordinates $\{x_1, y_1, z_1 = 0\}$ of a flat page, the goal is to transform it into P_{curl} with coordinates $\{x_2, y_2, z_2\}$, the point on a cylinder with radius r that's lying on the "spine" of the book. Now take a look at **Figure 5**, which shows the end of the cylinder. You can see the x and z axes (the y axis runs in and out of the page). Note that I'm representing the flat paper and the cylinder using the same colors as in the previous figures.

The key insight is that the distance from the origin to P_{flat} (x_1) is the same as the arc distance from the origin to P_{curl} along the cylinder. So, from simple geometry, I can say that the angle $\beta = x_1 / r$. Now, to get P_{curl}, I take the origin, move it down by r on the z axis, rotate around β, then move it up by r on the z axis. The curlPage method in **Figure 6** shows the code to deform the vertex buffer for a page. The vertex buffer and page coordinate information are abstracted away.

The variable conicContribution, ranging from -1 to +1, captures the position on the y axis the user has touched. A value of -1 represents the user touching the bottom of the page, and +1 represents the top of the page.

The complete set of deformation parameters is captured in curl_parameters:

```
struct curl_parameters
{
  curl_parameters() {}
  curl_parameters(float t, float a, float ang, float c) :
    theta(t), ay(a), angle(ang), conicContribution(c) {}
  float theta;  // Angle of right-cone
  float ay;     // Location on y axis of cone apex
  float alpha;  // Rotation about y axis
  float conicContribution;  // South tip cone == -1, cylinder == 0,
    north tip cone == 1
};
```

Note that the cylinder radius is missing from this struct; I'm taking a shortcut by computing it based on the cone parameters, as in **Figure 6**.

Architecture

With the geometry out of the way, I can focus on the design and page-curling architecture. The goal of the design is to allow for realistic page curling and uncurling without losing fluidity. For instance, the user should be able to partially curl a page, let go so the page uncurls somewhat, then continue curling the page, while the animation remains fluid and realistic.

The Project Austin page-curling architecture is implemented in the page_curl class, shown in **Figure 7**.

Here are the methods that matter:

void page_curl::attachPage(const std::shared_ptr<paper_sheet_node> &pageNode) is called by the Project Austin code whenever a page is curled. The paper_sheet_node data structure captures all the pertinent information about the page coordinate system, as well as the DirectX vertex buffer used

Figure 2 **Page Curling**

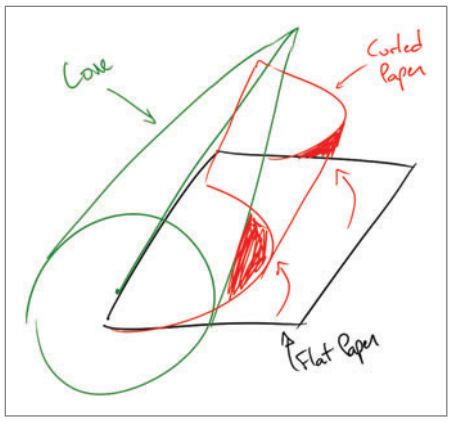


Figure 3 **Flat Paper (Black) Curling Around a Cone (Green) to Become the Curled Paper (Red)**

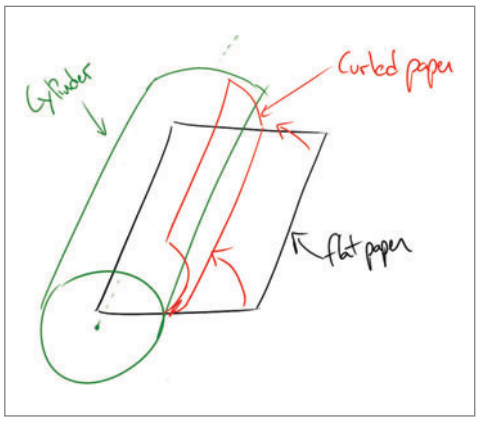


Figure 4 **Flat Paper (Black) Curling Around a Cylinder (Green) to Become the Curled Paper (Red)**

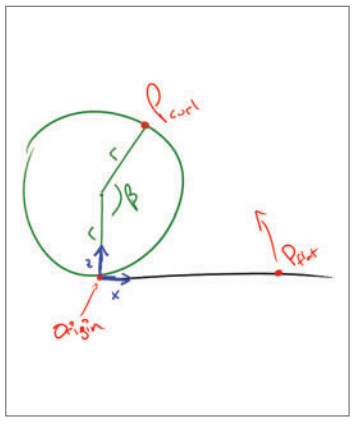


Figure 5 **Transforming P_{flat} to P_{curl}**

to render this particular page. The implementation isn't discussed in this article.

void page_curl::startUserCurl(float x, float y) is called by the Project Austin user input handler code to indicate the user has pressed his finger down and is curling at the location (x, y). This code does the following:

- sets the _userCurl state bit to indicate the user is curling the page
- unsets the _autoCurl state bit to stop any uncurling if it's in progress
- sets _nextCurlParams to the deformation parameters based on the user's position (x, y)

void page_curl::startAutoCurl is called by the Project Austin user input handler to indicate the user has let go of the screen. This code does the following:

Figure 6 **Deforming the Vertex Buffer**

```
void page_curl::curlPage(curl_parameters curlParams)
{
  float theta = curlParams.theta;
  float Ay = curlParams.ay;
  float alpha = curlParams.alpha;
  float conicContribution = curlParams.conicContribution;

  // As the user grabs toward the middle-right of the page, curl the
  // paper by deforming it on to a cylinder. The cylinder radius is taken
  // as the endpoint of the cone parameters: for example,
  // cylRadius = R*sin(theta) distance to where R is the the rightmost
  // point on the page, all the way up.
  float cylR = sqrt(  _vertexCountX * _vertexCountX
                  + (_vertexCountY /2 - Ay)*( _vertexCountY /2 - Ay));
  float cylRadius = cylR * sin(theta);

  // Flipping from top corner or bottom corner?
  float posNegOne;
  if (conicContribution > 0)
  {
    // Top corner
    posNegOne = 1.0f;
  }
  else
  {
    // Bottom corner
    posNegOne = -1.0f;
    Ay = -Ay + _vertexCountY;
  }

  conicContribution = abs(conicContribution);

  for (int j = 0; j < _vertexCountY; j++)
  {
    for (int i = 0; i < _vertexCountX; i++)
    {
      float x = (float)i;
      float y = (float)j;
      float z = 0;

      float coneX = x;
      float coneY = y;
      float coneZ = z;
      {
        // Compute conical parameters and deform
        float R = sqrt(x * x + (y - Ay)*(y - Ay));
        float r = R * sin(theta);
        float beta  = asin(x / R) / sin(theta);

        coneX = r * sin(beta);
        coneY = R + posNegOne * Ay - r * (1 - cos(beta)) * sin(theta);
        coneZ = r * (1 - cos(beta)) * cos(theta);

        // Then rotate by alpha about the y axis
        coneX = coneX * cos(alpha) - coneZ * sin(alpha);
        coneZ = coneX * sin(alpha) + coneZ * cos(alpha);
      }

      float cylX = x;
      float cylY = y;
      float cylZ = z;
      {
        float beta = cylX / cylRadius;

        // Rotate (0,0,0) by beta around line given by x = 0, z = cylRadius
        // aka Rotate (0,0,-cylRadius) by beta, then add cylRadius back
        // to z coordinate
        cylZ = -cylRadius;
        cylX = -cylZ * sin(beta);
        cylZ = cylZ * cos(beta);
        cylZ += cylRadius;

        // Then rotate by alpha about the y axis
        cylX = cylX * cos(alpha) - cylZ * sin(alpha);
        cylZ = cylX * sin(alpha) + cylZ * cos(alpha);
      }

      // Combine cone & cylinder results
      x = conicContribution * coneX + (1-conicContribution) * cylX;
      y = conicContribution * coneY + (1-conicContribution) * cylY;
      z = conicContribution * coneZ + (1-conicContribution) * cylZ;

      _vertexBuffer[j * _vertexCountX + i].position.x = x;
      _vertexBuffer[j * _vertexCountX + i].position.y = y;
      _vertexBuffer[j * _vertexCountX + i].position.z = z;
    }
  }
}
```

- unsets the _userCurl state bit to indicate the user is no longer curling the page
- sets the _autoCurl state bit to indicate an uncurl is in progress, with a time stamp of when the uncurl started

void page_curl::onRender is called by the Project Austin render loop for each frame. Note that this is the only function that actually deforms the vertex buffer. This code works as follows:

- If _userCurl or _autoCurl is set, the code deforms the vertex buffer to the parameters computed from _nextCurlParams and _currentCurlParams. Using both ensures smooth curling, as discussed later in this article.

The goal of the design is to allow for realistic page curling and uncurling without losing fluidity.

- If _autoCurl is set, the code calls continueAutoCurl
- Sets _currentCurlParams to _nextCurlParams

void page_curl::continueAutoCurl is called by page_curl::onRender if the page is uncurling. This code:

- computes _nextCurlParams based on when the uncurl started
- unsets _autoCurl if the page has completed curling

Figure 7 **The page_curl Class**

```
class page_curl
{
public:
  void attachPage(const std::shared_ptr<paper_sheet_node> &pageNode);
  void startUserCurl(float x, float y);
  void startAutoCurl();
  void onRender();

private:

  struct curl_parameters
  {
    curl_parameters() {}
    curl_parameters(float t, float a, float ang, float c) :
      theta(t), ay(a), angle(ang), conicContribution(c) {}
    float theta;  // Angle of right cone
    float ay;     // Location on y axis of cone apex
    float alpha;  // Rotation about y axis
    float conicContribution;  // South tip cone == -1, cylinder == 0,
north tip cone == 1
  };

  void continueAutoCurl();
  page_curl::curl_parameters computeCurlParameters(float x, float y);
  void page_curl::curlPage(page_curl::curl_parameters curlParams);

  ... Other helpers that will be discussed later ...

  std::shared_ptr<paper_sheet_node> _pageNode; // Page abstraction
  bool _userCurl;           // True if the user is curling the page
  bool _autoCurl;           // True if the page is uncurling
  float _autoCurlStartTime; // The time the user let go to start uncurling

  // Allows for smooth animations
  curl_parameters _currentCone;
  curl_parameters _nextCone;
};
```

page_curl::curl_parameters page_curl::computeCurlParameters(float x, float y) computes the curl parameters (theta, Ay, alpha, conicContribution) based on the user input.

Now that you've seen the overall architecture, I'll fill in each of the public and private methods. I chose the overall design to make it easy to ensure smooth animation. The key is splitting up startUserCurl and onRender, and maintaining the state between the two.

I'll now discuss some of these methods, providing either motivation or background regarding the design decisions.

Smooth Animation

Given the functions described previously, it might seem appropriate for startUserCurl to simply read the position of the user's finger, and for onRender to simply deform the page to those parameters.

Unfortunately, this particular animation can look ugly if the user moves her finger very fast. If onRender draws the deformed page at 60 frames per second (fps), it's possible that between two frames the user moved her finger halfway across the screen. On

Figure 8 **The Rendering Code**

```
void page_curl::onRender()
{
  // Read state under a lock
  curl_parameters nextCurlParams;
  curl_parameters currentCurlParams;
  bool userCurl;
  bool autoCurl;
  LOCK(_mutex)
  {
    nextCurlParams = _nextCurlParams;
    currentCurlParams = _currentCurlParams;
    userCurl = _userCurl;
    autoCurl = _autoCurl;
  }

  // Smooth going from currentCurlParams to nextCurlParams
  curl_parameters curl;
  float dt = nextCurlParams.theta - currentCurlParams.theta;
  float da = nextCurlParams.ay    - currentCurlParams.ay;
  float dr = nextCurlParams.alpha - currentCurlParams.alpha;
  float dc = nextCurlParams.conicContribution -
    currentCurlParams.conicContribution;
  float distance = sqrt(dt * dt + da * da + dr * dr + dc * dc);
  if (distance < constants::general::maxCurlDistance())
  {
    curl = nextCurlParams;
  }
  else
  {
    float scale = maxDistance / distance;
    curl.theta = currentCurlParams.theta + scale * dt;
    curl.ay =  currentCurlParams.ay  + scale * da;
    curl.alpha = currentCurlParams.alpha + scale * dr;
    curl.conicContribution = currentCurlParams.conicContribution + scale * dc;
  }

  // Deform the vertex buffer
  if (userCurl || autoCurl)
  {
    LOCK(_mutex)
    {
      _currentCurlParams = curl;
    }
    this->curlPage(curl);
  }

  // Continue (or stop) uncurling
  if (autoCurl)
  {
    this->continueAutoCurl();
  }
}
```

one frame, the page is deformed to a nearly flat state. If, on the very next frame, the page is deformed to a fully curled state, the fluidity of the animation is lost and ... well, it looks ugly.

To get around this, I keep track of not only _nextCurlParams (the desired location where the curl should go based on either user input or the formulas for uncurling), but also the current state of the curl in _currentCurlParams. If the desired curl location is too far from the existing curl location, then I should instead be curling to an intermediate value that ensures the animation is smooth.

The term "too far" is open to interpretation. Because there are four elements in the cone_parameters structure, and each one is a floating-point number, I treat _currentCurlParams and _nextCurlParams as points in four-dimensional space. The distance between the two curl parameters is then just the distance between two points.

Similarly, the term "intermediate value" is also open to interpretation. If the _nextCurlParams is too far from _currentCurlParams, I choose an intermediate point that's closer to _nextCurlParams proportional to the distance between the two points. So, if the user starts with a flat page and curls it extremely quickly, the page appears initially to spring forward quickly, but then slows down the closer it gets to the desired location. Because this happens at 60 fps, the overall effect is very minor, but from a usability standpoint the results look great.

Figure 8 shows the full rendering code.

Handling User Input (and Lack Thereof)

Project Austin, being a C++ DirectX XAML app, makes use of the WinRT APIs. Gesture recognition is handled by the OS—specifically by Windows::UI::Input::GestureRecognizer.

Figure 9 **The startAutoCurl Method**

```
void page_curl::startAutoCurl()
{
  LOCK(_mutex)
  {
    // It's possible the user let go, but the page is
    // already fully curled or uncurled
    bool shouldAutoCurl = !this->doneAutoCurling(curl);
    _userCurl = false;
    _autoCurl = shouldAutoCurl;

    if (shouldAutoCurl)
    {
      _autoCurlStartTime = constants::general::currentTime();
    }
  }
}
```

Figure 10 **Handling Auto-Uncurling**

```
void page_curl::continueAutoCurl()
{
  LOCK(_mutex)
  {
    if (this->doneAutoCurling(curl))
    {
      _autoCurl = false;
    }
    else
    {
      float time = constants::general::currentTime() - _autoCurlStartTime;
      _nextCurlParams = this->nextAutoCurlParams(_currentCurlParams, time * time);
    }
  }
}
```

Hooking up the onManipulationUpdated event to call startUserCurl(x, y) when the user is curling the page is straightforward. The code for startUserCurl is then:

```
// x is scaled to (0, 1) and y is scaled to (-1, 1)
void page_curl::startUserCurl(float x, float y)
{
  curl_parameters curl = this->computeCurlParameters(x, y);

  LOCK(_mutex)
  {
    // Set curl state, to be consumed by onRender()
    _nextCurlParams = curl;
    _userCurl = true;
    _autoCurl = false;
  }
}
```

The more interesting code is for handling auto-uncurling.

Similarly, it's straightforward to hook up the onManipulationCompleted event to call startAutoCurl when the user lets go of the page. **Figure 9** shows the code for startAutoCurl.

The more interesting code is for handling auto-uncurling when the user lets go of the page; the page will continue to uncurl until it's completely flat or until it has completely curled forward. I base this transformation on the current curl parameters and the elapsed time (squared) since uncurling started. This way, the page starts to uncurl slowly but as time passes it uncurls faster and faster. This is a cheap way to attempt to simulate gravity. **Figure 10** shows the code.

Tuning the Curling for Realism

Absent from the previous sections is the code for computeCurlParameters, doneAutoCurling and nextAutoCurlParams. These are tunable functions that involve constants, formulas and heuristics that are the result of hours of painstaking trial and error.

For example, I spent many, many hours trying to achieve reasonable results for computeCurlParameters. **Figure 11** shows two versions of the code—one that simulates the curling of a thick piece of paper (the pages inside a book), and a second that simulates a plastic cover (the cover of a softcover book, for example).

The code for knowing when curling is complete merely involves checking whether the page is completely flat or is completely curled:

```
bool page_curl::doneAutoCurling(curl_parameters curl)
{
  bool doneCurlBackwards =  (curl.theta > 89.999f)
                         && (curl.ay < -69.999f)
                         && (curl.alpha < 0.001f)
                         && (abs(curl.conicContribution) > 0.999f);
  bool doneCurlForwards = (curl.alpha > 99.999f);
  return doneCurlBackwards || doneCurlForwards;
}
```

And last, my version of auto-curl, shown in **Figure 12**, relies on the current curl position and the square of the elapsed time since the curl started. Instead of uncurling a page in the same way it's curled, I simply have the curl parameters linearly approach the parameters for a flat page, but let the page fall backward (if the user let go when the page was only slightly curled) or forward (if the user let go when the page was mostly curled). Using this technique and the square of the elapsed time, the page has a nice bounce to it when you let go. I really like how it looks.

One thing I wish I had implemented is page inertia when uncurling. Users should be able to fling the page. When they let go, the page should continue curling in the same direction it was flung, until drag forces it to stop and lay back flat. This could be implemented by adding history to onRender, keeping track of the last few positions of the user's finger, and making use of this in the formulas in nextAutoCurlParams.

Performance

The curlPage method has to do quite a bit of math to curl a single page. By my count, there are nine calls to the sin function, eight to cos, one to arcsin, one to sqrt, and around two dozen multiplies, plus additions and subtractions—for each vertex in the paper model—for each frame that's being rendered!

Project Austin aims for 60 fps; thus, processing each frame can take no more than 15 ms, lest the app feel sluggish.

The necessary performance is achieved by ensuring that the innermost loop is vectorized, where the Visual Studio C++ compiler generates Streaming SIMD Extensions 2 (SSE2) instructions to take advantage of CPU vector units. The compiler is able to vectorize all of the transcendental functions in the math.h header file.

In this case, the inner loop calculates the curled position for four vertices at a time. The performance boost frees up the CPU for other rendering tasks, such as applying shadows to the curled page.

Read more about the auto-vectorizer on MSDN and on the Parallel Programming in Native Code blog (bit.ly/bWfC5Y).

Wrapping Up

I'd like to thank the great people who worked on Project Austin, in particular Jorge Pereira, George Mileka and Alan Chan. By the time I started working on the project they already had a great app, and I feel fortunate to have spent some time adding a little realism to it. It helped me understand beauty in simplicity, and how tough it can be to make things simple!

You can find more information about the app, including some videos, on the MSDN blogs by searching for Project Austin. And you'll find code at austin.codeplex.com. ■

Figure 11 **Curling Two Kinds of Pages**

```
// Helper macro for a straight line F(x) that passes through {x1, y1} and {x2, y2}.
// This can't be a template function (C++ doesn't let you have float literals
// as template parameters).
#define STRAIGHT_LINE(x1, y1, x2, y2, x) (((y2 - y1) / (x2 - x1)) * (x - x1) + y1)

page_curl::curl_parameters page_curl::paperParams(float x, float y)
{
  float theta, ay, alpha;

  if (x > 0.95f)
  {
    theta = STRAIGHT_LINE(1.0f,  90.0f, 0.95f, 60.0f, x);
    ay    = STRAIGHT_LINE(1.0f, -20.0f, 0.95f, -5.0f, x);
    alpha = 0.0;
  }
  else if (x > 0.8333f)
  {
    theta = STRAIGHT_LINE(0.95f,  60.0f, 0.8333f, 55.0f, x);
    ay    = STRAIGHT_LINE(0.95f, -5.0f,  0.8333f, -4.0f, x);
    alpha = STRAIGHT_LINE(0.95f,  0.0f,  0.8333f, 13.0f, x);
  }
  else if (x > 0.3333f)
  {
    theta = STRAIGHT_LINE(0.8333f, 55.0f, 0.3333f,  45.0f, x);
    ay    = STRAIGHT_LINE(0.8333f, -4.0f, 0.3333f, -10.0f, x);
    alpha = STRAIGHT_LINE(0.8333f, 13.0f, 0.3333f,  35.0f, x);
  }
  else if (x > 0.1666f)
  {
    theta = STRAIGHT_LINE(0.3333f,  45.0f, 0.1666f,  25.0f, x);
    ay    = STRAIGHT_LINE(0.3333f, -10.0f, 0.1666f, -30.0f, x);
    alpha = STRAIGHT_LINE(0.3333f,  35.0f, 0.1666f,  60.0f, x);
  }
  else
  {
    theta = STRAIGHT_LINE(0.1666f,  25.0f, 0.0f,  20.0f, x);
    ay    = STRAIGHT_LINE(0.1666f, -30.0f, 0.0f, -40.0f, x);
    alpha = STRAIGHT_LINE(0.1666f,  60.0f, 0.0f,  95.0f, x);
  }

  page_curl::curl_parameters cp(theta, ay, alpha, y);
  return cp;
}

page_curl::curl_parameters page_curl::plasticParams(float x, float y)
{
  float theta, ay, alpha;

  if (x > 0.95f)
  {
    theta = STRAIGHT_LINE(1.0f,  90.0f, 0.9f,  40.0f, x);
    ay    = STRAIGHT_LINE(1.0f, -30.0f, 0.9f, -20.0f, x);
    alpha = 0.0;
  }
  else
  {
    theta = STRAIGHT_LINE(0.95f,  40.0f, 0.0f,  35.0f, x);
    ay    = STRAIGHT_LINE(0.95f, -20.0f, 0.0f, -25.0f, x);
    alpha = STRAIGHT_LINE(0.95f,   0.0f, 0.0f,  95.0f, x);
  }

  page_curl::curl_parameters cp(theta, ay, angle, y);
  return cp;
}
```

Figure 12 **My Version of Auto-Curling**

```
page_curl::curl_parameters page_curl::nextAutoCurlParams(
  curl_parameters curl, float time)
{
  curl_parameters nextCurl;
  if (curl.alpha > 40)
  {
    nextCurl.theta = min(curl.theta + time/800000.0f,  50.0f);
    nextCurl.ay    = curl.ay;
    nextCurl.alpha = min(curl.alpha + time/200000.0f, 100.0f);
  }
  else
  {
    nextCurl.theta = min(curl.theta + time/100000.0f, 90.0f);
    nextCurl.ay    = max(curl.ay - time/200000.0f,   -70.0f);
    nextCurl.alpha = max(curl.alpha - time/300000.0f,  0.0f);
  }

  if (curl.conicContribution > 0.0)
  {
    nextCurl.conicContribution =
      min(curl.conicContribution + time/100000.0f, 1.0f);
  }
  else
  {
    nextCurl.conicContribution =
      max(curl.conicContribution - time/100000.0f, -1.0f);
  }

  return nextCurl;
}
```

Eric Brumer *is a software development engineer at Microsoft, working on the Visual C++ compiler optimizer team. Reach him at ericbr@microsoft.com.*

THANKS to the following technical expert for reviewing this article: *George Mileka (Microsoft)*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

20 LIVE! CELEBRATING TWENTY YEARS

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

Intense Take-Home Training for Developers, Software Architects and Designers

ROCK YOUR CODE ON CAMPUS!

Visual Studio Live! is back on the Microsoft Campus – backstage passes in hand! Over 5 days and 65 sessions and workshops, you'll get an all-access look at the Microsoft Platform and practical, unbiased, developer training.

Register Today!

Use Promo Code REDAUG2

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

TUESDAY KEYNOTE: WINDOWS AZURE

Scott Guthrie

Corporate Vice President, Server and Tools Business Division, Microsoft

AGENDA AT-A-GLANCE

Windows 8 / WinRT | **Web and JavaScript Development**

START TIME	END TIME	
7:00 AM	8:00 AM	
8:00 AM	12:00 PM	**MW01** - Workshop: Building Windows 8 Applications - *Rockford Lhotka*
12:00 PM	2:30 PM	
2:30 PM	6:00 PM	**MW01** - Workshop: Building Windows 8 Applications - *Rockford Lhotka*

START TIME	END TIME	
7:30 AM	8:30 AM	
8:30 AM	9:30 AM	
9:45 AM	11:00 AM	**T01** - Interaction and Navigation Patterns in Windows 8 Applications - *Billy Hollis*
11:15 AM	12:30 PM	**T06** - Introduction to the WinRT API - *Jason Bock*
12:30 PM	2:30 PM	
2:30 PM	3:45 PM	**T11** - A Primer in Windows 8 Development with WinJS - *Philip Japikse*
3:45 PM	4:15 PM	
4:15 PM	5:30 PM	**T16** - Getting Beyond the Datagrid in Windows 8 - *Billy Hollis*
5:30 PM	7:30 PM	

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	9:00 AM	
9:15 AM	10:30 AM	**W01** - Expression Blend 5 for Developers: Design Your XAML or HTML5\CSS3 UI Faster - *Ben Hoelting*
10:45 AM	12:00 PM	**W06** - Using Your Favorite JavaScript Frameworks When Developing Windows Store Apps - *Keith Burnell*
12:00 PM	1:30 PM	
1:30 PM	2:45 PM	**W11** - Migrating from WPF or Silverlight to WinRT - *Rockford Lhotka*
2:45 PM	3:15 PM	
3:15 PM	4:30 PM	**W16** - Sharing Code Between Windows 8 and Windows Phone 8 apps - *Ben Dewey*
8:00 PM	10:00 PM	

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	9:15 AM	**TH01** - Windows Azure Mobile Services: Backend for Your Windows 8, iOS, and Android Apps - *Sasha Goldshtein*
9:30 AM	10:45 AM	**TH06** - Win8 + Cloud Services - *Rockford Lhotka*
11:00 AM	12:15 PM	**TH11** - Demystifying LOB Deployments in Windows 8 and Windows Phone 8 - *Tony Champion*
12:15 PM	2:45 PM	
2:45 PM	4:00 PM	**TH16** - Deep Dive into the Windows 8 Background APIs - *Tony Champion*
4:15 PM	5:30 PM	**TH21** - Windows 8 Apps with MVVM, HTML/JS, and Web API (An eCommerce Story) - *Ben Dewey*

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	12:00 PM	**FW01** - Workshop: Rich Data HTML Mobile and
12:00 PM	1:00 PM	
1:00 PM	5:00 PM	**FW01** - Workshop: Rich Data HTML Mobile and

*Speakers and sessions subject to change

EVENT SPONSOR Microsoft

PLATINUM SPONSORS esri, NETPath By Prospective Software

LOUNGE SPONSOR Dice

SUPPORTED BY Visual Studio, msdn magazine, Visual Studio MAGAZINE

REDMOND, WA | AUGUST 19-23, 2013

MICROSOFT CAMPUS

Visual Studio 2012 / .NET 4.5	SharePoint / Office	Azure / Cloud Computing	Data Management	Mobile Development	SQL Server

Visual Studio Live! Pre-Conference Workshops: Monday, August 19, 2013 (Separate entry fee required)

Pre-Conference Workshop Registration - Coffee and Morning Pastries			
MW02 - Workshop: End-to-End Service Orientation - Designing, Developing, & Implementing Using WCF and the Web API - *Miguel Castro*		**MW03** - Workshop: UX Design Bootcamp for Developers and Analysts - *Billy Hollis*	
Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
MW02 - Workshop: End-to-End Service Orientation - Designing, Developing, & Implementing Using WCF and the Web API - *Miguel Castro*		**MW03** - Workshop: UX Design Bootcamp for Developers and Analysts - *Billy Hollis*	

Visual Studio Live! Day 1: Tuesday, August 20, 2013

Registration - Coffee and Morning Pastries			
Keynote: Windows Azure - *Scott Guthrie, Corporate Vice President, Server and Tools Business Division, Microsoft*			
T02 - Controlling ASP.NET MVC4 - *Philip Japikse*	**T03** - What's New in Windows Azure - *Vishwas Lele*	**T04** - Data Visualization with SharePoint and SQL Server - *Paul Swider*	**T05** - Building Windows 8 Line of Business Apps - *Robert Green*
T07 - I'm Not Dead Yet! AKA The Resurgence of Web Forms - *Philip Japikse*	**T08** - Building LOB Apps in Windows Azure - *Vishwas Lele*	**T09** - A Developers Perspective on the Social Architecture of SharePoint 2013 - *Paul Swider*	**T10** - Working With Data in Windows Store Apps - *Robert Green*
Lunch - Visit Exhibitors			
T12 - jQuery Fundamentals - *Robert Boedigheimer*	**T13** - Moving Web Apps to the Cloud - *Eric D. Boyd*	**T14** - Developing Using the SharePoint 2013 REST Services - *Matthew DiFranco*	**T15** - What's New in the Visual Studio 2013 IDE - *Cathy Sullivan & Radhika Tadinada*
Sponsored Break - Visit Exhibitors			
T17 - Fiddler and Your Website - *Robert Boedigheimer*	**T18** - JavaScript, Meet Cloud: Node.js on Windows Azure - *Sasha Goldshtein*	**T19** - What's New in SharePoint 2013 Workflow Development? - *Matthew DiFranco*	**T20** - Windows Store Application Contracts and Extensibility - *Brian Peek*
Microsoft Ask the Experts & Exhibitor Reception – Attend Exhibitor Demos			

Visual Studio Live! Day 2: Wednesday, August 21, 2013

Registration - Coffee and Morning Pastries			
Keynote: To Be Announced			
W02 - Web API 101 - *Deborah Kurata*	**W03** - EF Code First: Magic: Unicorns and Beyond - *Keith Burnell*	**W04** - Building SharePoint Hosted Apps as Single Page Apps - *Andrew Connell*	**W05** - What's New in the .NET 4.5 BCL - *Jason Bock*
W07 - Exposing Data Services with ASP.NET Web API - *Brian Noyes*	**W08** - Display Maps in Windows Phone 8 - *Al Pascual*	**W09** - Deep Dive into the Cloud App Model for SharePoint - *Keenan Newton*	**W10** - Understanding Dependency Injection and Those Pesky Containers - *Miguel Castro*
Luncheon Round Table - Visit Exhibitors			
W12 - Knocking It Out of the Park, with Knockout.js - *Miguel Castro*	**W13** - Busy Developer's Guide to Cassandra - *Ted Neward*	**W14** - A Deep Dive into Creating Apps for Office - *Keenan Newton*	**W15** - Mastering Visual Studio 2012 - *Deborah Kurata*
Sponsored Break - Exhibitor Raffle @ 3:00 pm (Must be present to win)			
W17 - Tips for Building Multi-Touch Enabled Web Sites - *Ben Hoelting*	**W18** - Busy Developer's Guide to MongoDB - *Ted Neward*	**W19** - Real World Workflows with Visual Studio 2012, Custom Forms, Tasks, Events and Workflow CSOM - *Andrew Connell*	**W20** - TFS vs Team Foundation Service - *Brian Randell*
Lucky Strike Evening Out Party			

Visual Studio Live! Day 3: Thursday, August 22, 2013

Registration - Coffee and Morning Pastries			
TH02 - Building Rich Data HTML Client Apps with Breeze.js - *Brian Noyes*	**TH03** - Building Your First Windows Phone 8 Application - *Brian Peek*	**TH04** - Introducing SQL Server Data Tools - *Leonard Lobel*	**TH05** - Build It So You Can Ship It! - *Brian Randell*
TH07 - Busy Developer's Guide to AngularJS - *Ted Neward*	**TH08** - From Prototype to the Store: How to Truly Finish a Windows Phone App - *Nick Landry*	**TH09** - Big Data-BI Fusion: Microsoft HDInsight & MS BI - *Andrew Brust*	**TH10** - Better Process and Tools with Microsoft ALM - *Brian Randell*
TH12 - Design Patterns for Multi-Threaded Web Services - *Adam Wilson*	**TH13** - Connecting to Data from Windows Phone 8 - *Christopher Woodruff*	**TH14** - Programming the T-SQL Enhancements in SQL Server 2012 - *Leonard Lobel*	**TH15** - Building Great Windows Store Apps with XAML and C# - *Pete Brown*
Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
TH17 - Get your Node.js Under Control with TypeScript - *Yaniv Rodenski*	**TH18** - Developing Mobile Solutions with Azure and Windows Phone - *Christopher Woodruff*	**TH19** - Getting to Know the BI Semantic Model - *Andrew Brust*	**TH20** - WebMatrix 3: The Code Editor for the Cloud – *Justin Beckwith*
TH22 - SignalRity - *Yaniv Rodenski*	**TH23** - Designing Your Windows Phone Apps for Multitasking and Background Processing - *Nick Landry*	**TH24** - Intro to Windows Azure SQL Database and What's New - *Eric D. Boyd*	**TH25** - Web API 2 – Web Services for Websites, Modern Apps, and Mobile Apps - *Daniel Roth*

Visual Studio Live! Post Conference Workshops: Friday, August 23, 2013 (Separate entry fee required)

Post Conference Workshop Registration - Coffee and Morning Pastries	
Browser Clients with Knockout, JQuery, Breeze, and Web API - *Brian Noyes*	**FW02** - Workshop: SQL Server for Developers - *Andrew Brust & Leonard Lobel*
Lunch	
Browser Clients with Knockout, JQuery, Breeze, and Web API - *Brian Noyes*	**FW02** - Workshop: SQL Server for Developers - *Andrew Brust & Leonard Lobel*

PRODUCED BY
1105 MEDIA INC

vslive.com/redmond

Building Cross-Platform Web Services with ServiceStack

Ngan Le

I like working with Windows Communication Foundation (WCF) because there's excellent support for the framework in Visual Studio. I find it rather easy to build a WCF Web service from scratch and get it up and running in my development environment without installing additional tools and redistributables. I'll discuss my experiences with cross-platform development here, so you might be interested in this article if the following statements apply:

- You're already familiar with WCF and C#.
- You need to build a cross-platform Web service.

I think we can all agree that writing cross-platform apps is at the very least an inconvenience, but sometimes unavoidable. If you're anything like me—Windows-bound and heavily invested in C#—putting together a cross-platform Web service might entail significant overhead. This includes reconfiguring your beloved Windows computing environment to accommodate a completely different set of development tools and possibly learning yet another programming language. In this article, I'll show how you can leverage the WCF likeness of ServiceStack (an open source .NET and Mono REST Web services framework) to accomplish such a task without ever leaving Visual Studio or the Microsoft .NET Framework environment.

This article discusses:

- Typical layout of Web services
- Using Remote Procedure Calls or Data Transfer Objects
- An overview of ServiceStack
- Building equivalent Windows Communication Foundation and ServiceStack Web services
- Using built-in ServiceStack clients

Technologies discussed:

Windows Communication Foundation, ServiceStack

Code download available at:

archive.msdn.microsoft.com/mag201308Services

About Web Services

A typical Web service is laid out as depicted in **Figure 1**.

The service layer is where you define your Web service interface. This is the only layer with which the client needs to interact to consume your Web service.

The business layer is usually heavy with business logic, obviously. This is where the meat of your Web service implementation resides, keeping your service layer light and focusing on client/server contracts and communication.

The data layer is meant to encapsulate your data access and provide abstracted data models for manipulation at the business layer.

In this article, I'll focus on the service layer.

Figure 1 **A Typical Web Service Layout**

Remote Procedure Calls versus Data Transfer Objects

Some Web services take the Remote Procedure Call (RPC) approach, where each request is designed to resemble a function call:

```
public interface IService {
  string DoSomething(int input);
}
```

The RPC approach tends to make the Web service more susceptible to breaking interface changes. For example, in the preceding code snippet, if a later version of the Web service requires two inputs from the client to execute the DoSomething method—or needs to return another field in addition to the string value—a breaking change to old clients is unavoidable. Of course, you can always create a parallel DoSomething_v2 method to take two input arguments, but over time that would clutter your Web service interface and risk confusing your consumers, both old and new.

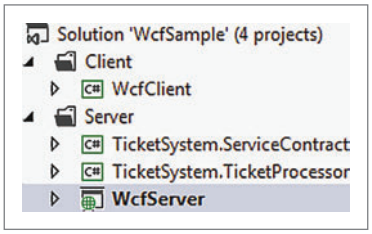


Figure 2 **WCF Ticket Service Visual Studio Projects**

> Conventional wisdom favors defining Web service interfaces geared toward the Data Transfer Object (DTO) model.

Conventional wisdom favors defining Web service interfaces geared toward the Data Transfer Object (DTO) model. The following code shows how the Web method DoSomething can be transformed under the DTO model:

```
public class DoSomethingRequest {
  public int Input { get; set; }
}

public class DoSomethingResponse {
  public string Result { get; set; }
}

public interface IService {
  DoSomethingResponse DoSomething(DoSomethingRequest request);
}
```

Following this school of thought, each Web method takes a single request DTO and returns a single response DTO. The idea is that adding new fields to the request DTO and response DTO won't break older clients.

It's worth noting that both RPC-style and DTO-style Web service interfaces are supported by WCF, but ServiceStack only supports the DTO style. ServiceStack embraces the principle of the remote DTO style of Web service interfaces, for less chattiness and more chunkiness in the Web service interface design. This is key to understanding ServiceStack, because the framework is designed in a way that reinforces the principle.

What Is ServiceStack?

As mentioned, ServiceStack is an open source, cross-platform Mono Web service framework, and it's gaining popularity. Web services built with ServiceStack can run in a Windows environment with .NET code or in a Linux environment with Mono support. The OSes supported by Mono include:

- Linux
- Mac OS X, iOS
- Sun Solaris
- BSD
- Microsoft Windows
- Nintendo Wii
- Sony PlayStation 3

For more information about the platforms supported by Mono, see mono-project.com/Supported_Platforms.

If you like working with the .NET Framework and WCF and are in need of deploying your .NET Web services in an environment other than Windows, ServiceStack is an ideal option. Due to the similarity of ServiceStack and WCF, the transition from one to the other requires little adjustment in terms of development environment and tools. You get to continue writing C# code inside Visual Studio.

ServiceStack enforces remote Web service best-practice, convention-based DTO standards for its Web service interface, while WCF leaves it to you to freely define a Web service API as you see fit. ServiceStack also provides an out-of-the-box response status object that can be used to compose the response DTO, encouraging a more direct and simplistic error-handling scheme. Although this can easily be implemented with WCF, it's not an obvious route. Recent versions of ServiceStack also provide an exception-based error-handling mechanism that's similar to—although not as sophisticated as—WCF fault-based error handling via fault contracts.

The standards enforced by ServiceStack are easily implemented in WCF with a little extra typing. However, in addition to portability, ServiceStack is a viable option to build a RESTful Web service because it establishes conventions that simplify HTTP URI routing. At the same time, ServiceStack forces each Web service request to be implemented in a separate class, promoting a natural separation of concerns for a RESTful service model.

ServiceStack also offers other perks, such as out-of-the-box logging and basic data validation utility. You can read more about ServiceStack at servicestack.net.

This article assumes that you have some familiarity with WCF and the .NET Framework. To better demonstrate how WCF

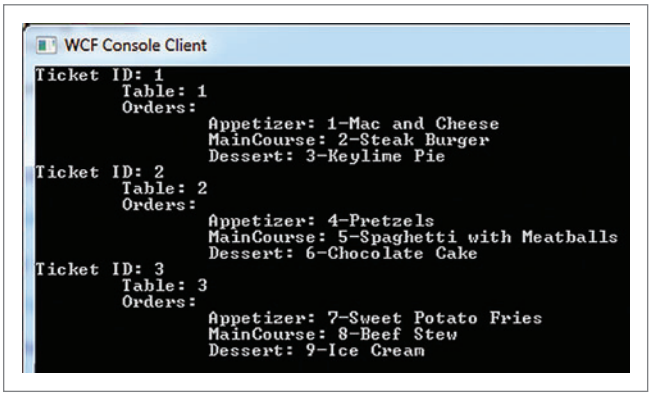


Figure 3 **The WCF Console Client**

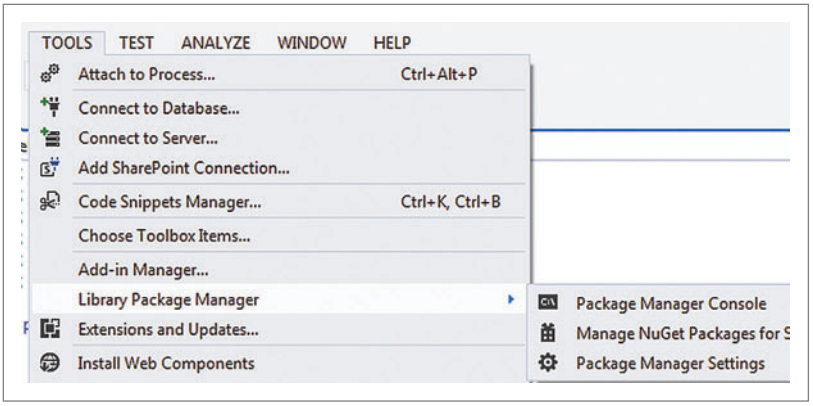


Figure 4 **The NuGet Package Manager Console in Visual Studio**

concepts can translate to ServiceStack concepts, I'll first implement the service layer in WCF. Then I'll show you how to transform the WCF Web service into a cross-platform Web service by porting it to an equivalent Web service using ServiceStack.

Building a Simple WCF Web Service

To demonstrate how ServiceStack can serve as a substitute for WCF in a multiplatform environment, I'll start with a simple WCF Web service.

The Web service is a trivial restaurant ticketing system called TicketService that implements the following service contract:

```
[ServiceContract]
public interface ITicketService {
  [OperationContract]
  List<Ticket> GetAllTicketsInQueue();

  [OperationContract]
  void QueueTicket(Ticket ticket);

  [OperationContract]
  Ticket PullTicket();
}
```

The TicketService allows its clients to queue a new ticket, pull a ticket off the queue and retrieve a complete inventory of all tickets currently in the queue.

The Web service consists of three Visual Studio projects, as shown in **Figure 2** (running my sample solution, which can be downloaded at archive.msdn.microsoft.com/mag201308Services, requires Visual Studio 2012 with Web Developer Tools and the .NET Framework 4.5).

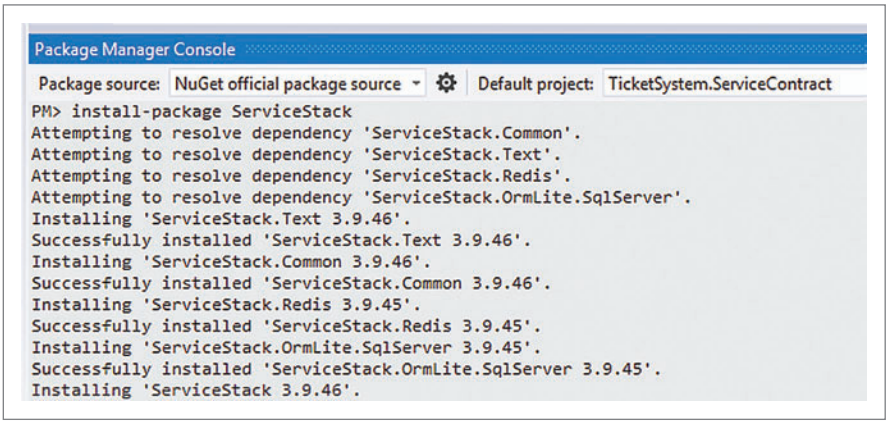


Figure 5 **Installing ServiceStack Redistributables with NuGet Package Manager**

The three projects are briefly described as follows:

- The TicketSystem.ServiceContract project is where the service interface is defined.
- The TicketSystem.TicketProcessor project contains the implementation details of the Web service business logic. This project contains no references to WCF.
- The WcfServer project implements TicketSystem.ServiceContracts and hosts the Web service in IIS.

I also put together a WCF client to consume the TicketService. The WCF client is a console application that uses code generated from adding a WCF service reference to the TicketService to communicate with the Web service using SOAP over IIS. Here's the client console implementation:

```
static void Main(string[] args) {
  Console.Title = "WCF Console Client";

  using (TicketServiceClient client = new TicketServiceClient()) {
    Ticket[] queuedTickets = client.GetAllTicketsInQueue();

    foreach (Ticket ticket in queuedTickets) {
      PrintTicket(ticket);
    }
  }
}
```

The client code implementation serves up the information shown in **Figure 3**.

Building an Equivalent ServiceStack Web Service

Prerequisite for Working with ServiceStack To work with ServiceStack, first you must download the redistributables. The easiest way to do this is through the NuGet Visual Studio extension that allows you to easily install and update third-party libraries and tools. You can download and install the NuGet client from nuget.codeplex.com. After NuGet is installed, you should see in Visual Studio the menu items shown in **Figure 4**.

Now I'll do a step-by-step walk-through on how to create a ServiceStack Web service that's the equivalent of the WCF Web service I built previously.

First, take the WCF Web service example as the starting point. Then remove the WcfClient and the WcfServer projects from the solution. These projects are specific to WCF, and they'll be replaced with ServiceStack-compatible projects later.

In the Package Manager Console window, select the TicketSystem.ServiceContract project. At the command prompt, type "install-package ServiceStack," as shown in **Figure 5**.

This step downloads the latest redistributables necessary to build a .NET Web service using ServiceStack. The redistributables are staged under a folder name "Packages" placed in your Visual Studio solution root folder. In addition, the DLL references are added to the TicketSystem.ServiceContract project. Along with that is a packages.config file created in

your project root folder, providing you with version and runtime information for each ServiceStack DLL:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="ServiceStack" version="3.9.46" targetFramework="net45" />
  <package id="ServiceStack.Common" version="3.9.46"
    targetFramework="net45" />
  <package id="ServiceStack.OrmLite.SqlServer" version="3.9.45"
    targetFramework="net45" />
  <package id="ServiceStack.Redis" version="3.9.45"
    targetFramework="net45" />
  <package id="ServiceStack.Text" version="3.9.46"
    targetFramework="net45" />
</packages>
```

Next, convert the WCF data contracts defined in TicketSystem.ServiceContract project to something that ServiceStack can understand.

Convert WCF Data Contracts to ServiceStack Data Contracts

WCF uses data contracts to establish the means of communication between the client and the server. ServiceStack does the same. WCF requires that appropriate attributes are put on any data object and data member you want to serialize and send over the wire; otherwise, WCF simply ignores them. This is where ServiceStack and WCF differ. ServiceStack will serialize all Plain Old CLR Objects (POCOs) referenced in the service contract and make them visible to the client side. Following are WCF and ServiceStack code representations of the same Ticket data contract reference in the TicketService interface.

Here's the WCF <Ticket> data contract:

```
[DataContract]
public class Ticket {
  [DataMember]
  public int TicketId { get; set; }

  [DataMember]
  public int TableNumber { get; set; }

  [DataMember]
  public int ServerId { get; set; }

  [DataMember]
  public List<Order> Orders { get; set; }

  [DataMember]
  public DateTime Timestamp { get; set; }
}
```

Here's the ServiceStack <Ticket> data contract:

```
public class Ticket {
  public int TicketId { get; set; }

  public int TableNumber { get; set; }

  public int ServerId { get; set; }

  public List<Order> Orders { get; set; }

  public DateTime Timestamp { get; set; }
}
```

The main difference between ServiceStack and WCF with regard to the service interface is that ServiceStack puts additional restrictions on the service interface. ServiceStack dictates that each unique request is identified by a unique request object, because there's no concept of a Web service "operation" (that is, method name) in the world of ServiceStack. This means that you can't reuse a request DTO across multiple service implementations with ServiceStack. All ServiceStack Web service operation contracts would equate to something similar to the following WCF service contracts:

```
[ServiceContract]
public interface ITicketService {
  [OperationContract]
  List<Ticket> GetAllTicketsInQueue(GetAllTicketsInQueueRequest request);

  [OperationContract]
  void QueueTicket(QueueTicketRequest request);

  [OperationContract]
  Ticket PullTicket(PullTicketRequest request);
}
```

The preceding code is nothing more than the original RPC-style TicketService WCF service interface shown here, but transformed to embrace the DTO convention:

```
[ServiceContract]
public interface ITicketService {
  [OperationContract]
  List<Ticket> GetAllTicketsInQueue();

  [OperationContract]
  void QueueTicket(Ticket ticket);

  [OperationContract]
  Ticket PullTicket();
}
```

Here's the equivalent TicketService ServiceStack service interface:

```
public class GetAllTicketsInQueueRequest {}

public class QueueTicketRequest {
  public Ticket Ticket { get; set; }
}

public class PullTicketRequest {}

public interface ITicketService {
  List<Ticket> Any(GetAllTicketsInQueueRequest request);

  void Any(QueueTicketRequest request);

  Ticket Any(PullTicketRequest request);
}
```

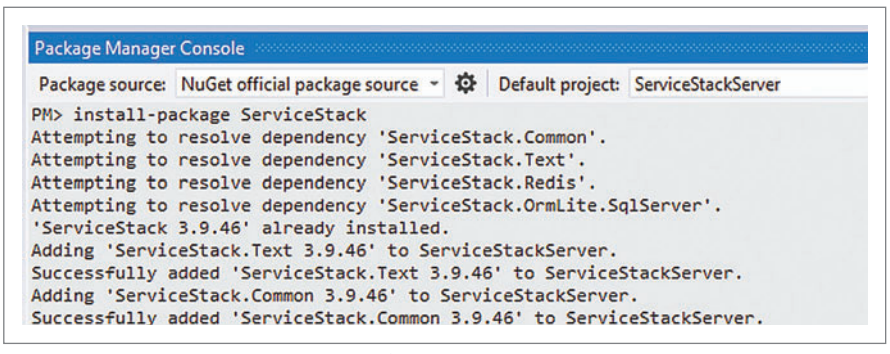


Figure 6 **Add ServiceStack Library References to the ServiceStackServer Project**

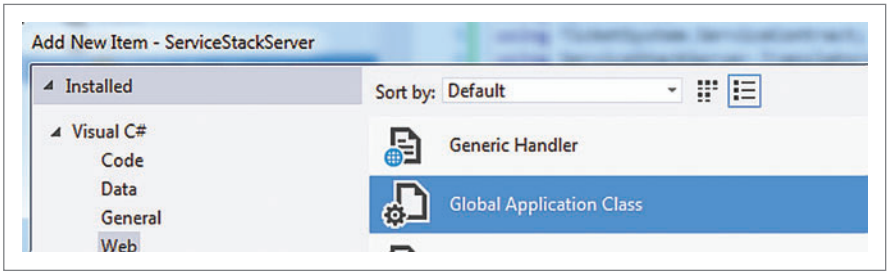


Figure 7 **Add Global.asax to ServiceStackServer**

ServiceStack supports different actions, such as Any, Get and Post. The choices you make here only impact HTTP requests. Specifying Any on a Web service request means the operation can be invoked by both an HTTP GET and an HTTP POST. This enforcement simplifies RESTful Web service implementation, which is beyond the scope of this article. To turn your ServiceStack Web service into a RESTful Web service, simply add URL [Route(…)] attributes to your Web service request declarations.

Create an ASP.NET Hosted ServiceStack Web Service Now that you have the Web service interface defined for your ServiceStack Web service, it's time to implement it and get it launched.

First, add an ASP.NET Empty Web Application. This is how I chose to host my ServiceStack Web service for convenience. It's by no means the only method for hosting a Web service built on ServiceStack. You can also host such a service inside a Windows Service or in the form of a console application running under a Web server, or even independent of one.

I named this Visual Studio project ServiceStackServer. It's the equivalent of the WcfServer project in the WCF service code sample.

Use the NuGet Package Manager Console to add ServiceStack references to ServiceStackServer as shown in **Figure 6**.

Now you have what you need to implement the ITicketService interface. The TicketService class must extend the ServiceStack.ServiceInterface.Service class provided by the framework, like this:

```
public class TicketService : ServiceStack.ServiceInterface.Service,
  ITicketService {

  public List<Ticket> Any(GetAllTicketsInQueueRequest request) {
    // Implement ...
  }

  public void Any(QueueTicketRequest request) {
    // Implement ...
  }

  public Ticket Any(PullTicketRequest request) {
    // Implement ...
  }
}
```

Everything else is the same as the WCF implementation.

Next, add a Global Application Class named Global.asax to the ServiceStackServer project as shown in **Figure 7**. This is where you initialize your ASP.NET Web application.

You're required to inherit from the ServiceStack.WebHost.Endpoints.AppHostBase class if you want to host your Web service inside an ASP.NET application. See **Figure 8** for an example.

If running under IIS 7 and later, the configuration item exhibited in **Figure 9** must be added to your Web.config file.

When you launch your ServiceStack Web application, your service contracts are listed as operations metadata, as shown in **Figure 10**. Now your Web service is ready to accept client requests. In the next section, I'll provide some consumer examples for the ServiceStack Web service.

ServiceStack Built-in Clients

The ServiceStack development group is very outspoken against generated Web service client code, so the framework provides a set of built-in Web service clients listed under the ServiceStack.ServiceClient.Web namespace. All built-in clients implement ServiceStack.Service.IServiceClient. Those that support REST implement ServiceStack.Service.IRestClient.

Available clients include:

- JsonServiceClient
- JsvServiceClient
- XmlServiceClient
- MsgPackServiceClient
- ProtoBufServiceClient
- Soap11ServiceClient
- Soap12ServiceClient

Each one supports a different serialization/deserialization format. Their usage is interchangeable because they implement a set of common interfaces.

To keep things simple, create a console application called ServiceStackClient to consume your ServiceStack TicketService using the JsvServiceClient. If you want to try out a different client, just replace the following JsvServiceClient code with any of the available clients previously listed:

```
static void Main(string[] args) {
  Console.Title = "ServiceStack Console Client";

  using (var client = new JsvServiceClient("http://localhost:30542")) {
    List<Ticket> queuedTickets =
      client.Send<List<Ticket>>(new GetAllTicketsInQueueRequest());

    if (queuedTickets != null) {
      foreach (Ticket ticket in queuedTickets) {
        PrintTicket(ticket);
      }
    }
  }
}
```

The console application does the following:

- Queries the TicketService for all tickets in the queue.
- Prints out the resulting tickets it got back from the TicketService.

Figure 8 **Hosting a ServiceStack Web Service Inside ASP.NET**

```
public class Global : System.Web.HttpApplication {

  public class TicketServiceHost : ServiceStack.WebHost.Endpoints.AppHostBase {

    // Register your Web service with ServiceStack.
    public TicketServiceHost()
      : base("Ticket Service", typeof(TicketService).Assembly) {}

    public override void Configure(Funq.Container container) {
      // Register any dependencies your services use here.
    }
  }

  protected void Application_Start(object sender, EventArgs e) {
    // Initialize your Web service on startup.
    new TicketServiceHost().Init();
  }
}
```

Figure 9 **The Web.config File for IIS 7 and Later**

```
<configuration>
  <system.web>...</system.web>
  <!--Required for IIS 7 (and above) -->
  <system.webServer>
    <validation validateIntegratedModeConfiguration="false" />
    <handlers>
      <add path="*" name="ServiceStack.Factory"
        type="ServiceStack.WebHost.Endpoints.
          ServiceStackHttpHandlerFactory, ServiceStack"
        verb="*" preCondition="integratedMode"
          resourceType="Unspecified"
        allowPathInfo="true" />
    </handlers>
  </system.webServer>
</configuration>
```

The console output is the exact same as that generated by the WCF client, shown in **Figure 3**.

More to Offer

This article only scratched the surface on what ServiceStack has to offer WCF users seeking cross-platform solutions. I haven't even discussed ServiceStack's support for streaming, async requests and message queues, for example.

As you can see, a few extra steps are required to build a ServiceStack Web service as opposed to a WCF Web service. However, I think the effort is rather negligible to transform the service layer of a Windows-bound WCF Web service into a multiplatform ServiceStack Web service. If I had gone a different route in search of platform independence—for example, using a Java Web service—the effort and the learning curve would've been much greater in comparison.

All things considered, if your Web service is destined to only run on a Windows OS, then WCF is arguably the better solution. There's less overhead when building a WCF Web service from scratch in the Windows environment, and you don't have yet another third-party redistributable to maintain and deploy to the target system. ■

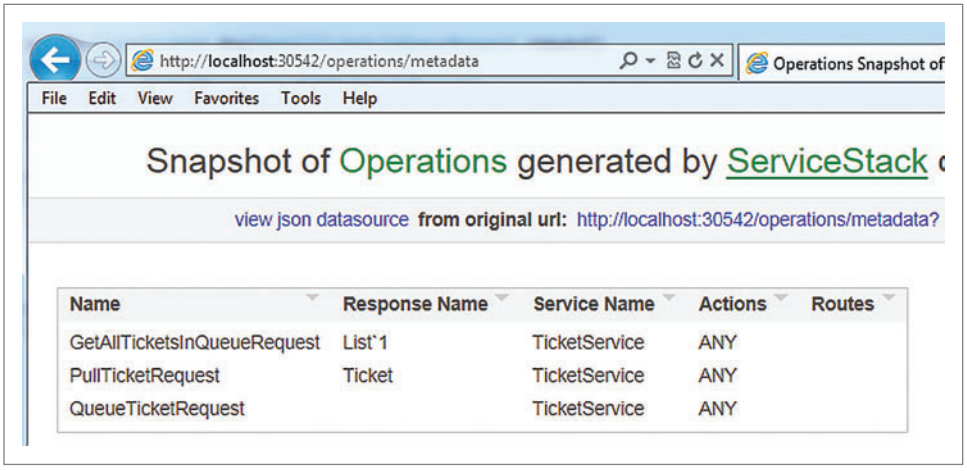


Figure 10 **TicketService Metadata**

Ngan Le *is a senior software engineer working closely with WCF who recently started to explore ServiceStack as a cross-platform solution for Web services.*

Thanks to the following technical expert for reviewing this article: *Andrew Oakley (Microsoft)*

Unit and Integration Testing of SSIS Packages

Pavle Gudurić

I worked on a project where we built extract, transform and load (ETL) processes with more than 150 packages. Many of them contained complex transformations and business logic, thus were not simple "move data from point A to point B" packages. Making minor changes was not straightforward and results were often unpredictable. To test packages, we used to fill input tables or files with test data, execute the package or task in Microsoft Business Intelligence Development Studio (BIDS), write a SQL query and compare the output produced by the package with what we thought was the correct output. More often we just ran the whole ETL process on a sample database and just sampled the output data at the end of the process—a time-consuming and unreliable procedure. Unfortunately, this is a common practice among SQL Server Integration Services (SSIS) developers. Even more challenging is to determine what effects the execution of one package has on subsequent packages. As you build your ETL process, you create a network of connected packages and different resources. It's difficult to maintain a complete overview of the numerous dependencies among all of these at all times.

This article explains how to perform unit and integration testing of SSIS packages by introducing a library called SSISTester, which is built on top of the managed SSIS API. After reading this article you should be able to use the described techniques and tools to automate unit and integration testing of your existing and new SSIS projects. To understand the article, you should have previous experience with SSIS and C#.

This article discusses:

- The SSISTester library
- Creating and executing unit tests
- Testing control flow tasks and preceding constraints
- Creating integration tests
- Executing live tests
- Test Engine internals

Technologies discussed:

SQL Server 2012, Visual Studio 2012, C#

Code download available at:

archive.msdn.microsoft.com/mag201308SSIS

SSISTester

When I started thinking about a testing framework for SSIS packages, I found three aspects to be important. First, I wanted to have a similar UX to writing tests using the Visual Studio testing framework, so the typical methodology involving setup, verification and cleanup (aka teardown) steps had to be applied. Second, I wanted to use existing and proven tools to write, execute and manage tests. Once again, Visual Studio was the obvious choice. And third, I wanted to be able to code tests in C#. With that in mind I wrote SSISTester, a .NET library that sits on top of the SSIS

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ORLANDO

20 LIVE!
CELEBRATING TWENTY YEARS

Access 3 of the best attended sessions on the hottest subjects from Visual Studio Live! Chicago here FREE:

WE'VE GOT YOUR TICKET TO FREE SESSIONS

PREVIEW VISUAL STUDIO LIVE! EVENT CONTENT
AT NO CHARGE!

Flip over for more details

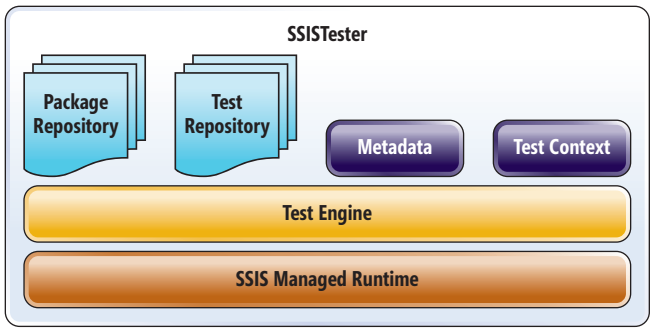


Figure 1 **Logical Components of the SSISTester Library**

runtime and exposes an API that allows you to write and execute tests for SSIS packages. The main logical components of the library are depicted in **Figure 1**.

The Package Repository is used to store raw XML representations of target packages. Each time a test is executed, a new instance of the Microsoft.SqlServer.Dts.Runtime.Package class is deserialized from XML with all fields and properties set to their default values. This is important because you don't want different tests that target the same package to accidently reuse any of the values set by previous tests.

Instances of test classes are stored within the Test Repository. These classes contain methods that implement your test cases. When a test is executed, these methods are called by the Test Engine. The specific rules that must be followed when creating test classes will be described in detail later.

Metadata contains the attributes needed to decorate a test class so it can be recognized as a test implementation. The Test Engine looks for these attributes when loading tests into the Test Repository.

The Test Context represents a set of classes that provide access to the runtime information during different phases of the test execution. For example, you can use these classes to access different aspects of a package being tested, such as variables, properties, preceding constraints, connection managers, currently executing task, package errors and so forth.

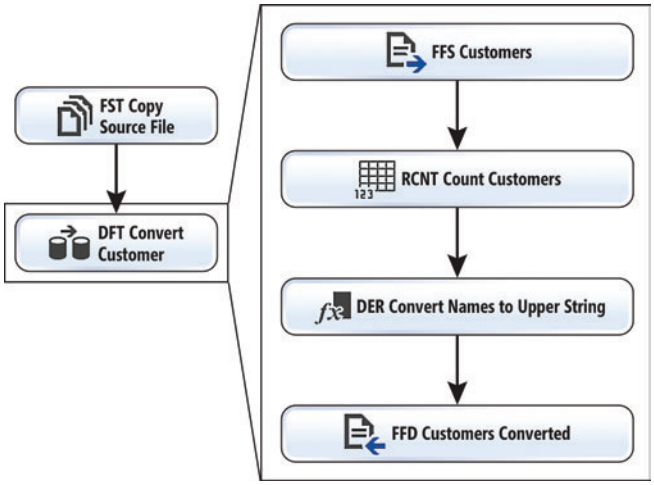


Figure 2 **Control Flow (Left) and Data Flow (Right) of the CopyCustomers.dtsx Package**

The Test Engine refers to the core classes and interfaces of the SSISTester API that directly utilize the managed SSIS runtime. They are used to load packages and test classes into their respective repositories, as well as to execute tests and to create test results.

Mini ETL

To create packages and test classes, I'll use Visual Studio 2012 and SQL Server 2012, and I'll use three packages to illustrate a simple ETL scenario in which customer data, delivered as a text file, is transformed and stored within a database. The packages are CopyCustomers.dtsx, LoadCustomers.dtsx and Main.dtsx. CopyCustomers.dtsx copies the Customers.txt file from one location to another and on the way it converts all customer names to uppercase text. Customers.txt is a simple CSV file that contains ids and names of customers, like so:

```
id,name
1,company1
5,company2
11,company3
```

LoadCustomers.dtsx loads the converted names into the Demo database. Before it loads data into a target table called CustomersStaging, it truncates all previously stored data. At the end of the process, it stores the number of customers into a variable. Here's the script to create the Demo database and the CustomersStaging table:

```
CREATE
DATABASE [Demo]
GO
USE [Demo]
GO
CREATE TABLE [dbo].[CustomersStaging](
  [Id] [int] NULL,
  [Name] [nvarchar](255) NULL
) ON [PRIMARY]
GO
```

The package Main.dtsx contains two Execute Package tasks that execute the sub-packages CopyCustomers.dtsx and LoadCustomers.dtsx, respectively. Connection managers in both CopyCustomers.dtsx and LoadCustomers.dtsx are configured using expressions and package variables. The same package variables are retrieved from the parent package configuration when executed from within another package.

Creating Unit Tests

To begin, create a console project and add assembly references to SSIS.Test.dll and SSIS.Test.Report.dll. I'm going to create a unit test for the CopyCustomers.dtsx package first. **Figure 2** shows the control flow (left) and data flow (right) for CopyCustomers.dtsx.

Every unit test is implemented in a single class that derives from the BaseUnitTest class and must be decorated with the UnitTest attribute:

```
[UnitTest("CUSTOMERS", "CopyCustomers.dtsx")]
public class CopyCustomersTest : BaseUnitTest{
  protected override void Setup(SetupContext context){}
  protected override void Verify(VerificationContext context){}
  protected override void Teardown(TeardownContext context){}
}
```

The UnitTest attribute marks a class as a unit test implementation so it can be found by the Test Engine. The first parameter corresponds to the Package Repository where a target package will be loaded during test execution, CUSTOMERS in this example. The second parameter can be the name of a target package, the path to a task in the control flow, the path to an event handler or the path to

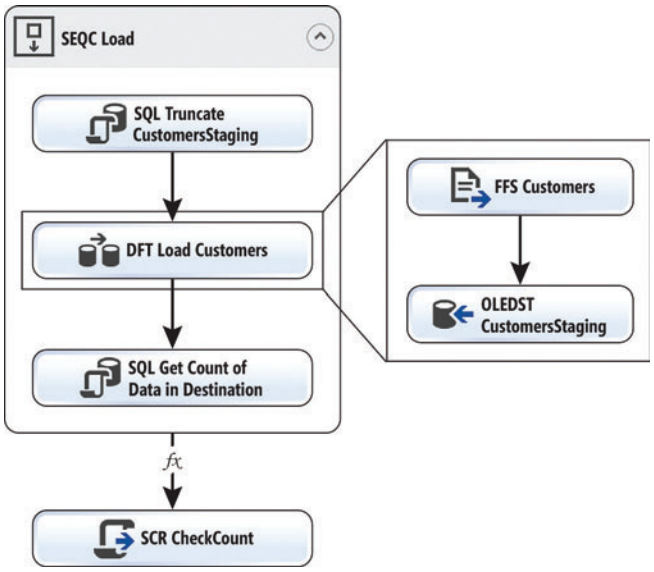


Figure 3 **Control Flow (Left) and Data Flow (Right) of the LoadCustomers.dtsx Package**

a preceding constraint. In this example it's the name of the CopyCustomers.dtsx package because I want to test the whole package. Basically, the UnitTest attribute tells the Test Engine to look for the CopyCustomers.dtsx package in the CUSTOMERS repository and execute it during the CopyCustomersTest test.

The base class BaseUnitTest that all unit test implementations need to derive from contains three methods that have to be implemented: Setup, Verify and Teardown.

These three methods are executed during different test phases. The Setup method runs before a target package is executed by the Test Engine. Setup prepares the package and all inputs and outputs the package depends on so it can be successfully validated and executed. In the following example, I set paths to the package variables that are used as connection strings in the connection managers:

```
protected override void Setup(SetupContext context){
  if(File.Exists(@"C:\TestFiles\Archive\Customers.txt"))
    File.Delete(@"C:\TestFiles\Archive\Customers.txt");
  if(File.Exists(@"C:\TestFiles\Converted\Customers.txt"))
    File.Delete(@"C:\TestFiles\Converted\Customers.txt");
  DtsVariable sourceFile = context.Package.GetVariable("SourcePath");
  sourceFile.SetValue(@"\\nc1\Customers\Customers.txt");
  DtsVariable destinationFile = context.Package.GetVariable("DestinationPath");
  destinationFile.SetValue(@"C:\TestFiles\Archive\Customers.txt");
  DtsVariable convertedFile = context.Package.
GetVariable("ConvertDestinationPath");
  convertedFile.SetValue(@"C:\TestFiles\Converted\Customers.txt");
}
```

After the Setup method has successfully executed, Test Engine executes the target package. When the package has executed, Test Engine calls the Verify method and I can check whether my assertions are true:

```
protected override void Verify(VerificationContext context){
  Assert.AreEqual(true, context.Package.IsExecutionSuccess);
  Assert.AreEqual(true, File.Exists(@"C:\TestFiles\Archive\Customers.txt"));
  Assert.AreEqual(true, File.Exists(@"C:\TestFiles\Converted\Customers.txt"));
  string[] lines = File.ReadAllLines(@"C:\TestFiles\Converted\Customers.txt");
  Assert.AreEqual("COMPANY2", lines[2].Split(',')[1]);
}
```

The first assert checks whether the package has executed successfully. The second one determines whether the FST Copy Source File file system task copied the \\nc1\Customers\Customers.txt file to the C:\TestFiles\Archive\ folder. The last two asserts validate whether the DFT Convert Customer Names data flow task correctly converted company names to uppercase. Earlier, I briefly described the testing context. Here you can see how I used the context parameter to access a package object within the Setup and Verify methods.

At the end of the test, I use the Teardown method to delete the files that were copied or created by the package:

```
protected override void Teardown(TeardownContext context){
  File.Delete(@"C:\TestFiles\Archive\Customers.txt");
  File.Delete(@"C:\TestFiles\Converted\Customers.txt");
}
```

Testing Control Flow Tasks

Tests can target specific tasks in the control flow as well. For example, to test the DFT Load Customers data flow in the LoadCustomers.dtsx package, I used an additional parameter of the UnitTest attribute, called ExecutableName, to tell the Test Engine that I want to test this task:

```
[UnitTest("CUSTOMERS", "LoadCustomers.dtsx",ExecutableName =
  @"\[LoadCustomers]\[SEQC Load]\[DFT Load customers]"))]
public class LoadCustomersTest : BaseUnitTest{
}
```

ExecutableName represents the path that combines names of all nested containers beginning with a package name.

Control and data flow for LoadCustomers.dtsx are shown in **Figure 3**.

When a test targets a specific task, only that task is executed by the Test Engine. If the successful execution of the target task depends on the execution of preceding tasks, the results of executing those tasks need to be manually generated. The DFT Load Customers data flow expects the target table to be truncated by the SQL Truncate CustomersStaging task. Further, the data flow expects the transformed Customers.txt file at a specific location. Because this file is created by the CopyCustomers.dtsx package, I need to copy it manually. Here's the Setup method that does all this:

```
protected override void Setup(SetupContext context){
  string dbConStr = @"Data Source=.;Integrated Security=SSPI;Initial
Catalog=Demo";
  string ssisConStr = @"Provider=SQLNCLI11;" + dbConStr;
  File.Copy(@"\\nc1\Customers\Customers.txt", @"C:\TestFiles\Converted\
Customers.txt");
  context.DataAccess.OpenConnection(dbConStr);
  context.DataAccess.ExecuteNonQuery("truncate table [dbo].[CustomersStaging]");
  context.DataAccess.CloseConnection();
  DtsConnection conn = context.Package.GetConnection("CustomerDB");
  conn.SetConnectionString(ssisConnStr);
  conn = context.Package.GetConnection("CustomersSrc");
  conn.SetConnectionString(@"C:\TestFiles\Converted\Customers.txt");
}
```

By using File.Copy, I copy the Customers.txt to the location expected by the data flow. Then I use the DataAccess property of the SetupContext to execute a truncate statement on the target table. This property exposes a lightweight ADO.NET wrapper that enables you to execute SQL commands without having to use SqlConnection and SqlCommand classes every time you want to access the database. At the end, I use the Package property to set the connection strings to the underlying connection managers.

Testing Preceding Constraints

Writing tests that target preceding constraints is also possible. For example, the CountConstraint that precedes the SCR CheckCount script task in the LoadCustomers.dtsx package has an expression

Figure 4 **The Complete Unit Test**

```
[UnitTest("CUSTOMERS", "LoadCustomers.dtsx",PrecedenceConstraintsTestOnly = true))]
public class LoadCustomersConstraintsTest : BaseUnitTest{
  private DtsPrecedenceConstraint _countConstraint;
  protected override void Setup(SetupContext context){
    DtsVariable variable = context.Package.GetVariable("CustomerCount");
    variable.SetValue(0);
    _countConstraint =
      context.Package.GetPrecedingConstraintForPath(
      @"\[LoadCustomers]\[SCR    CheckCount].[CountConstraint]");
    _countConstraint.SetExecutionResult(DtsExecutionResult.Success);
  }
  protected override void Verify(VerificationContext context)
  {
    Assert.AreEqual(false, _countConstraint.Evaluate());
  }
  protected override void Teardown(TeardownContext context){}
}
```

that checks whether the variable CustomerCount is greater than zero. If this expression evaluates to true and the SEQC Load task executes successfully, then the script task is executed. **Figure 4** shows the complete unit test.

To prepare the precedence constraint to be tested, I need to do two things. First, I have to set the CustomerCount variable to some value, because the expression in the precedence constraint refers to it. In this case, I choose 0. Next, I set the execution result of the preceding task to success, failure or completion. I do this by using the SetExecutionResult method to simulate success of the preceding task. This means that CountConstraint should evaluate to false and this is what I expect in the Verify method. You can have only one class where you implement unit tests for all preceding constraints in a package. Therefore, there's no target path to the particular constraint in the UnitTest attribute, only a Bool flag that tells the engine that this is a unit test class for precedence constraints. The reason for this is that with precedence constraints, there's no need to execute the package or task before the Verify method is called.

Executing Unit Tests

Before I can execute my tests, I need to load target packages and tests into their repositories. To do this, I need a reference to the Test Engine. Open the Program.cs file and replace the empty Main method with this one:

```
static void Main{
  IUnitTestEngine engine = EngineFactory.
GetClassInstance<IUnitTestEngine>();
  engine.LoadPackages("CUSTOMERS", @"C:\TargetPackages\");
  engine.LoadUnitTests();
  engine.ExecuteUnitTestsWithGui();
}
```

The first line creates a reference to the Test Engine. To load all packages from the folder C:\TargetPackages\ into the CUSTOMERS repository, I use the LoadPackages method. The LoadUnitTests method loads all classes in the calling assembly that are decorated with the UnitTest attribute into the specified test repository. Finally, I call ExecuteUnitTestsWithGui to start the execution of tests and to open the monitoring GUI, which is shown in **Figure 5**.

The GUI in **Figure 5** is practical if you want to test locally on your machine and you don't want to start Visual Studio. If you'd like to test packages on a server, you could make small modifications to the program and schedule it to run tests directly on a build server, for example:

```
static void Main{
  IUnitTestEngine engine = EngineFactory.
GetClassInstance<IUnitTestEngine>();
  engine.LoadPackages("CUSTOMERS", @"C:\TargetPackages\");
  engine.LoadUnitTests();
  engine.ExecuteUnitTests();
  engine.UnitTestResults.SaveAsHtml(@"C:\TestResults\");
}
```

The IUnitTestEngine interface has the UnitTestResults property that lets you access test results and save them as an HTML report. I replaced ExecuteUnitTestsWithGui with ExecuteUnitTests, which doesn't show the monitoring GUI. You could also run tests inside Visual Studio or use ReSharper so you don't need to start the console program. To do this, I created the new class called SSISUnitTestAdapter, shown in **Figure 6**.

> You can have only one class where you implement unit tests for all preceding constraints in a package.

If you've worked with the Microsoft unit testing framework before, you'll recognize the TestClass, AssemblyInitialize and TestMethods attributes. The three test methods, CopyCustomersTest, LoadCustomersTest and LoadCustomersConstraintsTest, wrap the call of the ExecuteUnitTest method, which in turn executes the Setup, Verify and Teardown methods of the class that's passed as parameter. The Prepare method creates the Test Engine object and loads packages and unit tests into their respective repositories. I used slightly different methods called LoadRepositoryUnitTests to load tests bound to the CUSTOMERS repository only. This is useful if you don't want to load all tests. You can execute all tests by clicking on Tests | Execute | All Tests in Visual Studio.

Creating Integration Tests

The basic idea of unit tests is to isolate all of the possible effects other packages or tasks may have on the one being tested. Sometimes it can be challenging to create a realistic test setup and the initial conditions needed for a unit test to ensure the package or task being tested behaves like a part of a complete ETL process. Because you usually implement ETL processes with a number of packages, you need to perform integration tests to be sure that each package works well when run as part of that

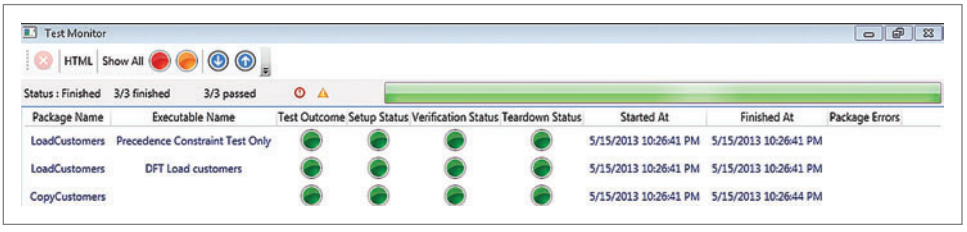


Figure 5 **The Monitoring GUI During the Execution of Tests**

Figure 6 **The SSISUnitTestAdapter Class**

```
[TestClass]
public class SSISUnitTestAdapter{
  IUnitTestEngine Engine {get;set;}
  [AssemblyInitialize]
  public static void Prepare(TestContext context){
    Engine = EngineFactory.GetClassInstance<IUnitTestEngine>();
    Engine.LoadPackages("CUSTOMERS", @"C:\TargetPackages\");
    Assembly testAssembly =
      Assembly.GetAssembly(typeof(CopyCustomersTest));
    Engine.LoadRepositoryUnitTests(testAssembly, "CUSTOMERS");
  }
  [TestMethod]
  public void CopyCustomersTest(){
    Engine.ExecuteUnitTest(typeof(CopyCustomersTest));
  }
  [TestMethod]
  public void LoadCustomersTest(){
    Engine.ExecuteUnitTest(typeof(LoadCustomersTest));
  }
  [TestMethod]
  public void LoadCustomersConstraintsTest(){
    Engine.ExecuteUnitTest(typeof(LoadCustomersConstraintsTest));
  }
}
```

process. The idea is to define probing points in your ETL process where you want to perform tests, without having to stop the whole process. As the process progresses and reaches the probing point, your tests are executed and you can verify a "live" work-in-progress ETL process; hence the name, "live test."

A live test is basically a post-condition—defined for a package, task or event handler—that needs to be satisfied after the package, task or event handler has executed. This post-condition corresponds to the verification step of a unit test. Live tests are different from the unit tests because it's not possible to prepare the test prior to package execution or to perform a clean-up step afterward. This is because unlike a unit test, a live test doesn't execute the package; it's the other way round: A package executes a test when it comes to the probing point for which a post-condition is defined.

Figure 7 illustrates this difference. Note the position of the package in both figures. When running unit tests, the Test Engine explicitly executes a unit test by calling its Setup, Verify and Teardown methods. A package is executed as a part of this Setup-Verify-Teardown sequence.

On the other hand, when running live tests, the Test Engine executes a package explicitly, which in turn triggers the execution of action methods that implement the post-conditions for a package and its tasks.

In order to create a live test for the CopyCustomers.dtsx package, I created the new class called CopyCustomers, shown in **Figure 8**.

Each live test class must derive from the BaseLiveTest class, a major difference when compared with a unit test. The BaseLiveTest class is used internally by the Test Engine to execute live tests and has no methods that have to be overridden. The ActionClass attribute marks this class as a live test. The parameters are the same as when using the UnitTest attribute—repository and target package. Note that unlike unit tests where each test is implemented in a single, separate class, only one class is needed to implement all post-conditions for a package. Live test classes can have an arbitrary number of post-conditions that should be evaluated. These post-conditions correspond to the Verify method in a unit test and are implemented as methods decorated with the ActionMethod attribute. In the example in **Figure 8**, I have one post-condition for each task in the package and one for the package itself. ActionMethod accepts a path to the target task, which is the same as the ExecutableName in the UnitTest attribute. This tells the Test Engine to execute this method when the target task has executed. Unlike the Verify method, which is always executed, these post-conditions might not be called when, for example, the target task doesn't execute successfully or the preceding constraint evaluates to false. The ActionContext parameter provides the same functionality as the VerificationContext.

Executing Live Tests

The steps necessary to execute live tests are slightly different than when executing unit tests. To execute live tests, replace the Main method in the Program.cs file with the code in **Figure 9**.

I need an instance of ILiveTestEngine, which I create using EngineFactory. Loading packages is the same as when using IUnitTestEngine. The LoadActions method loads all actions defined in the calling assembly and is practically an equivalent of LoadUnitTests. At this point, however, the similarity with unit tests stops. Instead of executing unit tests, I tell the Test Engine to execute the Main.dtsx package by calling the ExecuteLiveTestsWithGui.

When the Main.dtsx package starts, it runs the CopyCustomers.dtsx by executing the EPT CopyCustomers task. Each successfully finished task in the CopyCustomers.dtsx triggers one of the corresponding action methods in the CopyCustomersLiveTests class. It's important to note that this test implicitly tests the configuration settings of CopyCustomers.dtsx package.

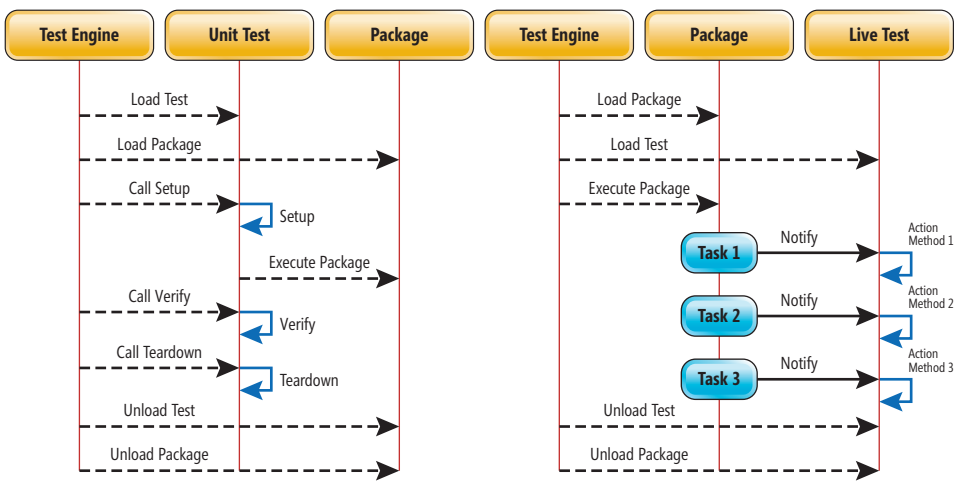
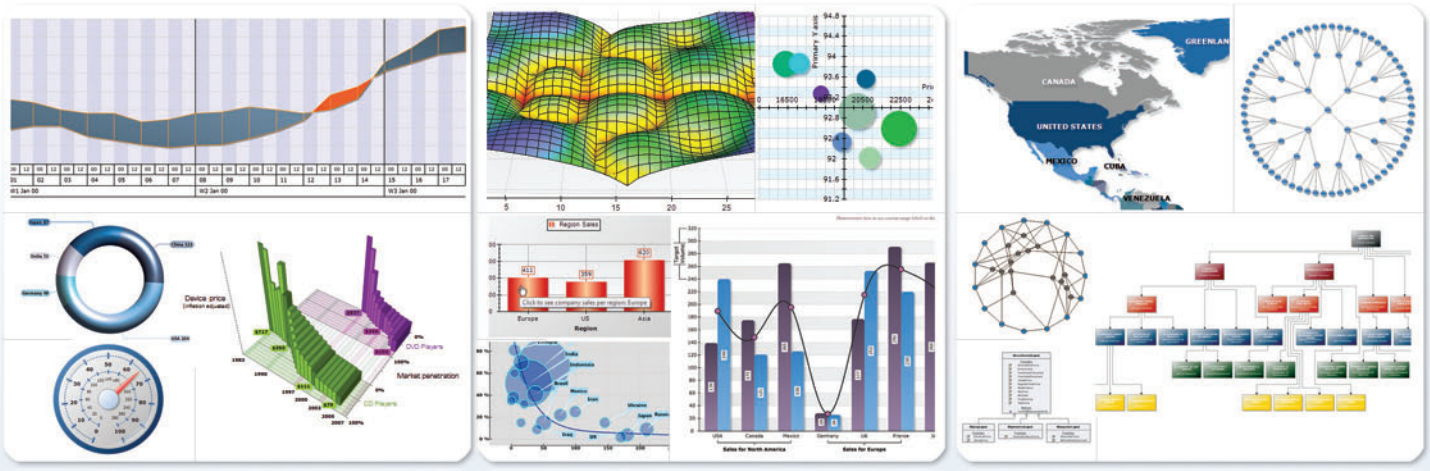


Figure 7 **Sequence Diagrams for Unit Test (Left) and Live Test (Right) Execution**

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.



Nevron Vision for .NET

Incorporates components that help you create enterprise grade digital dashboards, scorecards, diagrams, maps, MMI interfaces and much more.

Nevron Vision for SharePoint

The leading data visualization web parts for SharePoint 2007, 2010 and 2013.
Helps you convert your SharePoint pages into interactive dashboards and reports.

Nevron Vision for SSRS

The leading data visualization report items for SSRS 2005, 2008, 2008R2 and 2012.
Helps you deliver deeper data insights with more engaging looks.

Nevron components integrate seamlessly in Web and Desktop .NET applications, SQL Server Reporting Services 2005/2008/2008R2 and 2012 reports and SharePoint 2007/2010/2013 portals and deliver an unmatched set of enterprise-grade features. That is why Nevron is the trusted vendor by many Fortune 500 companies for their most demanding data visualization needs.

Make sure that your data is making the visual statement
it deserves by downloading your free evaluation copy
from www.nevron.com today.

Configured variables inherit their values from the Main.dtsx package. Please note that these variables are used as connection strings in the flat file connection managers of the CopyCustomers.dtsx package. This basically means that execution success of the tasks in the CopyCustomers.dtsx package depends on whether the value handover between these two packages works properly. This is a simple example of how interactions and dependencies between packages are tested, but you can imagine more complex scenarios where isolated unit tests wouldn't be enough to cover the test case.

Figure 8 **The CopyCustomers Class**

```
[ActionClass("CUSTOMERS", "CopyCustomers.dtsx")]
public class CopyCustomers : BaseLiveTest{
  [ActionMethod(@"\[CopyCustomers]")]
  public void TestWholePackage(ActionContext context){
    Assert.AreEqual(true, context.Package.IsExecutionSuccess);
  }
  [ActionMethod(@"\[CopyCustomers]\[FST Copy Source File]")]
  public void TestCopySourceFile(ActionContext context){
    Assert.AreEqual(true, context.ActiveExecutable.IsExecutionSuccess);
    Assert.AreEqual(true, File.Exists(@"C:\TestFiles\Archive\Customers.txt"));
  }
  [ActionMethod(@"\[CopyCustomers]\[DFT Convert customer names]")]
  public void TestConvertCustomersNames(ActionContext context){
    Assert.AreEqual(true, context.ActiveExecutable.IsExecutionSuccess);
    string[] lines = File.ReadAllLines(@"C:\TestFiles\Converted\Customers.txt");
    Assert.AreEqual("COMPANY2", lines[2].Split(',')[1]);
  }
}
```

Figure 9 **The Main Method for Executing Live Tests**

```
static void Main{
  string dbConStr = @"Data Source=.;Integrated Security=SSPI;Initial Catalog=Demo";
  string ssisConStr = @"Provider=SQLNCLI11;" + dbConStr;
  ILiveTestEngine engine = EngineFactory.
GetClassInstance<ILiveTestEngine>();
  engine.LoadPackages("CUSTOMERS", @"C:\TargetPackages\");
  engine.LoadActions();
  ExecutionParameters params = new ExecutionParameters();
  params.AddVariable(@"\[Main].[ConnectionString]", ssisConStr);
  params.AddVariable(@"\[Main].[CopyCustomersPath]", @"C:\TargetPackages\
CopyCustomers.dtsx");
  params.AddVariable(@"\[Main].[LoadCustomersPath]", @"C:\TargetPackages\
LoadCustomers.dtsx");
  params.AddVariable(@"\[Main].[ConvertDestinationPath]",
    @"C:\TestFiles\Converted\Customers.txt");
  params.AddVariable(@"\[Main].[DestinationPath]", @"C:\TestFiles\
Archive\Customers.txt");
  params.AddVariable(@"\[Main].[SourcePath]", @"\\nc1\Customers\Customers.txt");
  engine.SetExecutionParameters(parameters);
  engine.ExecuteLiveTestsWithGui("CUSTOMERS", "Main.dtsx");
}
```

Figure 10 **The LoadUnitTests Method**

```
public void LoadUnitTests(){
  Assembly assembly = Assembly.GetCallingAssembly();
  IEnumerable<Type> types = assembly.GetTypes().Where(t =>
t.GetCustomAttributes(false).OfType<UnitTestAttribute>().Any() &&
t.BaseType != null && t.BaseType.Name.Equals("BaseUnitTest"));

  foreach (Type t in types)
  {
    var attribute =
      t.GetCustomAttributes(false).OfType<UnitTestAttribute>().Single();
    DtsPackage package =
      _packages[attribute.Repository].GetForName(attribute.PackageName);
    string executable = attribute.ExecutableName;
    bool precedenceTestOnly = attribute.PrecedenceConstraintsTestOnly;

    var test = (BaseUnitTest)Activator.CreateInstance(t);
    test.TestClass = t;
    test.SetTestTargets(package, executable, precedenceTestOnly);
    test.Started += BaseUnitTestStarted;
    test.Finished += BaseUnitTestFinished;

    _unitTests.Add(test);
  }
}
```

Figure 11 **The ExecuteTest Method**

```
public void ExecuteTest(){
  Result = new UnitTestResult(Package, Executable) { TestOutcome =
    TestOutcome.InProgress, StartedAt = DateTime.Now };
  ExecuteSetup(CreateSetupContext());
  if (!Result.IsSetupSuccess)
    ExecuteTeardown(CreateTeardownContext());
  else{
    if(!PrecedenceOnly)
      Executable.Execute();
    ExecuteVerify(CreateVerifyContext());
    ExecuteTeardown(CreateTeardownContext());
    Result.FinishedAt = DateTime.Now;
  }
}
```

Test Engine Internals

The core class that implements the main functions of the SSISTester library is TestEngine. It's an internal class that's exposed through the IUnitTestEngine and ILiveTestEngine interfaces. The two methods that reveal most of the inner logic are LoadUnitTests (shown in **Figure 10**) and ExecuteUnitTests.

LoadUnitTests basically iterates all classes decorated with the UnitTest attribute and creates an instance for each. These instances are then cast to BaseUnitTest and are assigned the target package previously loaded from the package repository. At the end, all instances are saved in the _unitTests list. The method ExecuteUnitTests iterates all BaseUnitTest instances and calls ExecuteTests on each:

```
public void ExecuteUnitTests(){
  foreach (BaseUnitTest t in _unitTests){
    t.ExecuteTest();
  }
}
```

The actual execution of unit tests is implemented in the ExecuteTest method (shown in **Figure 11**) in the BaseUnitTest class.

The most important aspect of this method is that it executes the Setup, Verify and Teardown methods as well as the target package.

Wrapping Up

The examples presented here, along with the accompanying project, should allow you to start testing your SSIS projects. Automating the testing of your SSIS packages can save you a lot of time. What's more important, automated testing is more reliable because it's done continuously and you can cover more packages. Once you have written tests, you can always run them during automated build processes. In the end, this means fewer errors and better quality. ■

Pavle Gudurić *is a software engineer located in Germany. He has a master's degree in e-business and several technical certifications, and develops business intelligence (BI) solutions in the finance industry. Reach him at pavgud@gmail.com.*

Thanks to the following technical experts for reviewing this article: *Christian Landgrebe (LPA) and Andrew Oakley (Microsoft)*

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

ORLANDO • NOVEMBER • 18-22, 2013

LIVE EVENT

Modern Apps LIVE!

MODERN APPS FROM START TO FINISH

THE FUTURE OF SOFTWARE DEVELOPMENT IS BACK!

Critics raved about the Modern Apps Live! performance in March, so it's back for an encore performance! Presented in partnership with Magenic, Modern Apps Live! brings Development Managers, Software Architects and Development Leads together to break down the latest and greatest techniques in low-cost, high-value application development.

Presented in partnership with Magenic

SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA

WHAT SETS MODERN APPS LIVE! APART?

A singular topic focus: sessions build on each other as the conference progresses, leaving you with a holistic understanding of modern applications.

REGISTER TODAY TO SAVE $400!

SUPER EARLY BIRD SAVINGS END SEPTEMBER 12

Use Promo Code MALAUG2

CONNECT WITH MODERN APPS LIVE!

- twitter.com – @modernappslive
- facebook.com – Search "Modern Apps Live"
- linkedin.com – Search "Modern Apps Live" group!

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

Modern Apps Live! is part of Live! 360, the ultimate IT and Developer line-up. This means you'll have access to four (4) events, 22 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!

Modern Apps LIVE!
MODERN APPS FROM START TO FINISH

SharePoint LIVE!
TRAINING FOR COLLABORATION

SQL Server LIVE!
TRAINING FOR DBAs AND IT PROS

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

modernappslive.com | live360events.com

Architecture for Hosting Third-Party .NET Plug-Ins

Gennady Slobodsky and Levi Haskell

Last November Bloomberg L.P. released App Portal, an application platform that allows independent third-party software developers to sell their Microsoft .NET Framework Windows Presentation Foundation (WPF)-based applications to more than 300,000 users of the Bloomberg Professional service.

In this article we'll present a general-purpose architecture similar to the one employed by the Bloomberg App Portal for hosting third-party "untrusted" .NET applications. The accompanying source code (archive.msdn.microsoft.com/mag201308Plugins) contains a reference implementation of a .NET plug-in host and a demo plug-in using the Bloomberg API to chart historical pricing information for a given security.

This article discusses:

- Plug-in host architecture
- Launching plug-ins
- Bloomberg App Portal

Technologies discussed:

Windows Presentation Foundation, .NET App Domains, Bloomberg Desktop API

Code download available at:

archive.msdn.microsoft.com/mag201308Plugins

Plug-in Host Architecture

The architecture presented in **Figure 1** consists of a main application process and a plug-in hosting process.

Developers implementing a plug-in hosting infrastructure should carefully consider the pros and cons of implementing a plug-in host as a separate process. In the case of the App Portal scenario, we believe the pros of this approach significantly outweigh the cons, but we'll list the most important factors for your own consideration.

The pros of implementing a plug-in host as a separate process include:

- It decouples the main application process from plug-ins and, as a result, reduces the possibility of any negative impact plug-ins can have on the performance or usability of the application. It lessens the risk of plug-ins blocking the main application's UI thread. Also, it's less likely to cause memory or any other critical resource leaks in the main process. This approach also reduces the possibility for a poorly written plug-in to bring down the main application process by causing either "managed" or "unmanaged" unhandled exceptions.
- It can potentially improve security of the entire solution by applying sandboxing technologies similar to ones used by The Chromium Projects (see bit.ly/k4V3wq for details).
- It leaves more virtual memory space available for the main application process (this is more important for 32-bit processes, which are limited by 2GB of virtual memory space available for process user-mode code).

- It lets you extend functionality of non-.NET applications using .NET plug-ins.

The cons are mostly related to an increase of overall implementation complexity:

- You need to implement a separate inter-process communication (IPC) mechanism (special attention should be paid to the versioning of the IPC interface when the main application process and the plug-in host have different release or deployment cycles).
- You must manage the lifetime of the plug-in hosting process.

Addressing user security is one of the major concerns when designing the hosting process for untrusted third-party plug-ins. Defining a proper security architecture is worth a separate conversation and beyond the scope of this article.

The .NET application domain (System.AppDomain class) provides a comprehensive and robust solution for hosting .NET plug-ins.

An AppDomain has the following powerful features:

- Type-safe objects in one AppDomain can't directly access objects in another AppDomain, allowing the host to enforce isolation of one plug-in from another.
- An AppDomain can be individually configured, allowing the host to fine-tune the AppDomain for different types of plug-ins by providing different configuration settings.
- An AppDomain can be unloaded, allowing the host to unload plug-ins and all associated assemblies, with the exception of assemblies loaded as domain-neutral (using the loader-optimization options LoaderOptimization.MultiDomain or LoaderOptimization.MultiDomainHost). This feature makes the hosting process more robust by allowing the host to unload plug-ins that fail in the managed code.

The main application and plug-in host processes can interface using one of the various IPC mechanisms available, such as COM, named pipes, Windows Communication Foundation (WCF) and so on. In our proposed architecture, the role of the main application process is to manage creation of a composite UI and provide various application services for plug-ins. **Figure 2** shows the Bloomberg Launchpad view, representing such a composite UI. A "Stealth Analytics" component is created and rendered by a WPF-based plug-in hosted by the Bloomberg App Portal, while all other components are created and rendered by a Win32-based Bloomberg Terminal application. The main application process sends commands to a plug-in hosting process through a plug-in Controller Proxy.

A Plug-in Controller is running in the Default AppDomain of the plug-in host process and is responsible for processing commands received from the main application process, loading plug-ins into dedicated AppDomains and managing their lifetimes.

The sample source code provides a reference implementation of our architecture and consists of the hosting infrastructure and the SAPP and DEMO plug-ins.

Application Directory Structure

As **Figure 3** shows, the base application directory contains three assemblies:

- Main.exe represents the main application process and provides the UI for launching plug-ins.
- PluginHost.exe represents the plug-in hosting process.
- Hosting.dll contains the PluginController, responsible for instantiating plug-ins and managing their lifetimes.

API assemblies provided for use by the plug-ins are deployed into a separate subdirectory called PAC, which stands for Private Assembly Cache, a concept similar to the .NET Global Assembly Cache (GAC) but which, as its name suggests, holds items that are private to the application.

Each plug-in is deployed into its own subdirectory under the Plugins folder. The name of the folder corresponds to the plug-in's four-letter mnemonic used to launch it from the UI command line. The reference implementation contains two plug-ins. The first one, associated with mnemonic SAPP, is an empty WPF UserControl that simply prints its name. The second one, associated with mnemonic DEMO, shows a price history chart for a given security using the Bloomberg Desktop API (DAPI).

Inside each plug-in subdirectory there's a Metadata.xml file as well as one or more .NET assemblies. SAPP's Metadata.xml contains the plug-in's Title (used as the window title for the plug-in) and the names of the plug-in MainAssembly and MainClass, implementing the plug-in's entry point:

```
<?xml version="1.0" encoding="utf-8" ?>
<Plugin>
  <Title>Simple App</Titlte>
  <MainAssembly>SimpleApp</MainAssembly>
  <MainClass>SimpleApp.Main</MainClass>
</Plugin>
```

Launching Plug-Ins

A plug-in hosting process creates a single instance of PluginController in the default AppDomain at startup time. The application's main process uses .NET Remoting to call the PluginController.Launch(string[] args) method to launch the plug-in associated with the mnemonic entered by the user (SAPP or DEMO in the sample reference implementation). The PluginController instance must override the InitializeLifetimeService method inherited from

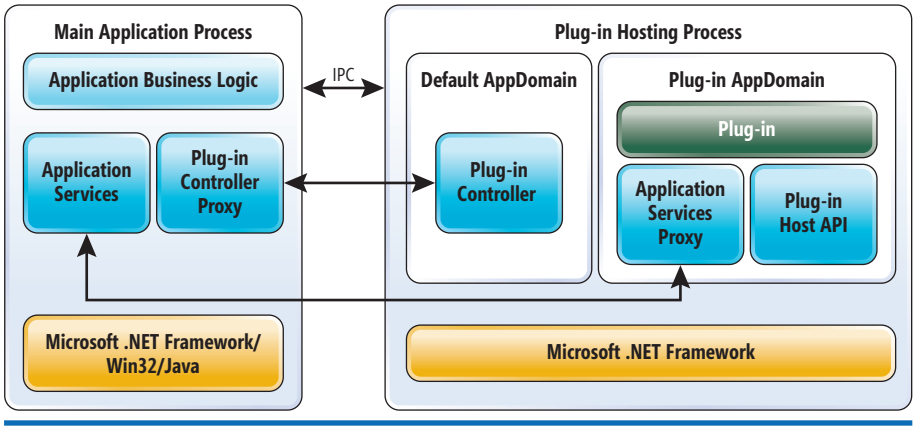


Figure 1 **Architecture for Hosting .NET Plug-Ins**

Figure 2 **An Example Composite UI**

- It improves the isolation of plug-ins from each other.
- It simplifies the development process by using the location of the plug-in's main assembly as a base directory in the same way as a standalone .NET application.
- It requires special hosting infrastructure logic to locate and load infrastructure and PAC assemblies located outside the plug-in's base directory.

When launching a plug-in, we first create a new plug-in hosting AppDomain:

```
var domain =
  AppDomain.CreateDomain(
  mnemonic, null, setup);
```

Next, we load the Hosting.dll into the newly created AppDomain, create an instance of the PluginContainer class and call the Launch method to instantiate the plug-in.

Seemingly, the easiest way to accomplish these tasks would be to use the AppDomain.CreateInstanceFromAndUnwrap method, because this approach allows you to directly specify the location of the assembly. However, using this approach will cause Hosting.dll to be loaded into the load-from context instead of the default context. Using the load-from context will have a number of subtle side effects, such as the inability to use native images or load assemblies as domain-neutral. Increased plug-in startup time will be the most obvious negative result of using the load-from context. More information about assembly load contexts can be found on the MSDN Library page, "Best Practices for Assembly Loading," at bit.ly/2Kwz8u.

A much better approach is to use AppDomain.CreateInstanceAndUnwrap and specify the location of the Hosting.dll and dependent assemblies in the XML configuration information of the AppDomain using the <codeBase> element. In the reference implementation, we dynamically generate configuration XML and System.MarshalByRefObject to extend its own lifetime, otherwise the object will be destroyed after five minutes (the default lifetime of the MarshalByRefObject):

```
public override object InitializeLifetimeService()
{
  return null;
}

Public class PluginController : MarshalByRefObject
{
  // ...
  public void Launch(string commandLine)
  {
    // ...
  }
}
```

The PluginController sets the base directory for the new AppDomain as per the directory structure shown in **Figure 3**:

```
var appPath = Path.Combine(_appsRoot, mnemonic);
var setup = new AppDomainSetup {ApplicationBase = appPath};
```

Using a different base directory for the AppDomain has the following important ramifications:

- bin
  - Main.exe
  - PluginHost.exe
  - Hosting.dll
  - PAC
    - Bloomberglp.Blpapi.dll
    - System.Windows.Controls.DataVisualization.Toolkit.dll
    - WPFToolkit.dll
  - Plugins
    - SAPP
      - Metadata.xml
      - SimpleApp.dll
    - DEMO
      - Metadata.xml
      - DapiSample.dll

Figure 3 **The Base Application Directory Structure**

Figure 4 **An Example of Generated AppDomain Configuration**

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity
          name="PluginHost.Hosting"
          publicKeyToken="537053e4e27e3679" culture="neutral"/>
        <codeBase version="1.0.0.0" href="Hosting.dll"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="Bloomberglp.Blpapi"
          publicKeyToken="ec3efa8c033c2bc5" culture="neutral"/>
        <codeBase version="3.6.1.0" href="PAC/Bloomberglp.Blpapi.dll"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="WPFToolkit"
          publicKeyToken="51f5d93763bdb58e" culture="neutral"/>
        <codeBase version="3.5.40128.4" href="PAC/WPFToolkit.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

WINDOWS FORMS | WPF | ASP.NET

FLOW TYPE LAYOUT REPORTING

Reuse MS Word documents or templates as your reporting templates.

Easy database connection with master-detail nested blocks.

Powerful, programmable template designer with full sources for Visual Studio®.

Integrate dynamic 2D and 3D charting to your reports.

Create print-ready, digitally signed Adobe PDF and PDF/A documents.

Create flow type layouts with tables, columns, images, headers and footers and more.

TX TEXT CONTROL® word processing components

Visual Studio Microsoft Partner

US +1 855-533-8398
EU +49 421-4270671-0

WWW.TEXTCONTROL.COM

Figure 5 **The Run Method of the PluginContainer Class**

```
private void Run()
{
  var metadata = new XPathDocument(
    Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Metadata.xml"))
    .CreateNavigator().SelectSingleNode("/Plugin");

  Debug.Assert(metadata != null);

  var mainAssembly = (string) metadata.Evaluate("string(MainAssembly)");
  var mainClass = (string) metadata.Evaluate("string(MainClass)");
  var title = (string) metadata.Evaluate("string(Title)");

  Debug.Assert(!string.IsNullOrEmpty(mainAssembly));
  Debug.Assert(!string.IsNullOrEmpty(mainClass));
  Debug.Assert(!string.IsNullOrEmpty(title));

  var rootElement = ((Func<string[], UIElement>) Delegate.CreateDelegate(
    typeof (Func<string[], UIElement>),
    Assembly.Load(mainAssembly).GetType(mainClass),
    "CreateRootElement"))(_args);

  var window =
    new Window
    {
      SizeToContent = SizeToContent.WidthAndHeight,
      Title = title,
      Content = rootElement
    };

  new Application().Run(window);
  AppDomain.Unload(AppDomain.CurrentDomain);
}
```

assign it to the new AppDomain using the AppDomainSetup.SetConfigurationBytes method. An example of the generated XML is shown in **Figure 4**.

The PluginContainer class derived from System.MarshalByRefObject doesn't need to override default lifetime management as in the case of the PluginController class, because it handles only a single remote call (the Launch method) immediately after creation:

```
var host = (PluginContainer) domain.CreateInstanceAndUnwrap(
  pluginContType.Assembly.FullName, pluginContType.FullName);
host.Launch(args);
```

Figure 6 **The DEMO Plug-In**

Figure 7 **The DEMO Plug-in XAML**

```
<UserControl x:Class="DapiSample.MainView"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c=
    "clr-namespace:System.Windows.Controls.DataVisualization.Charting;
    assembly=System.Windows.Controls.DataVisualization.Toolkit"
  xmlns:v=
    "clr-namespace:System.Windows.Controls.DataVisualization;
    assembly=System.Windows.Controls.DataVisualization.Toolkit"
  Height="800" Width="1000">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="30"/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <c:Chart x:Name=
        "_chart" Background="White" Grid.Row="1" Visibility="Hidden">
        <c:Chart.Axes>
          <c:LinearAxis x:Name=
            "_linearAxis" Orientation="Y" ShowGridLines="True"/>
          <c:DateTimeAxis x:Name=
            "_DateAxis" Orientation="X" ShowGridLines=
            "True" Interval="1" IntervalType="Months" />
        </c:Chart.Axes>
        <c:LineSeries x:Name=
          "_lineSeries" DependentValuePath="Value"
          IndependentValuePath="Date" ItemsSource="{Binding}"/>
        <c:Chart.LegendStyle>
          <Style TargetType="{x:Type v:Legend}">
            <Setter Property="Width" Value="0"></Setter>
            <Setter Property="Height" Value="0"></Setter>
          </Style>
        </c:Chart.LegendStyle>
      </c:Chart>
    <TextBox Grid.Row="0" x:Name=
      "_security" IsReadOnly="True" TextAlignment="Center"/>
  </Grid>
</UserControl>
```

The Launch method of the PluginContainer class creates a UI thread for the plug-in and sets the COM apartment state to single-threaded apartment (STA) as WPF requires:

```
[SecurityCritical]
public void Launch(string[] args)
{
  _args = args;

  var thread = new Thread(Run);
  thread.TrySetApartmentState(ApartmentState.STA);
  thread.Start();
}
```

The Run method of the PluginContainer class (see **Figure 5**) is the startup method of the plug-in's UI thread. It extracts names of the plug-in's main assembly and the main class specified by the MainAssembly, and the MainClass elements of the Metadata.xml file, loads the main assembly, and uses reflection to find the entry point in the main class.

In the reference implementation, the entry point is defined as a public static method of the main class named CreateRootElement, accepting an array of strings as a startup argument and returning an instance of System.Windows.UIElement.

After calling the entry point method, we wrap its return value in a WPF window object and launch the plug-in. The Run method of the System.Windows.Application class, shown in **Figure 5**, enters a message loop and doesn't return until the plug-in's main window is closed. After that, we schedule unloading of the plug-in's AppDomain and clean up all resources it was using.

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

Figure 8 **Obtaining a Reference Data Service Object**

```
private Session _session;
private Service _refDataService;

var sessionOptions = new SessionOptions
  {
    ServerHost = "localhost",
    ServerPort = 8194,
    ClientMode = SessionOptions.ClientModeType.DAPI
  };

_session = new Session(sessionOptions, ProcessEventCallBack);

if (_session.Start())
{
  // Open service
  if (_session.OpenService("//blp/refdata"))
  {
    _refDataService = _session.GetService("//blp/refdata");
  }
}
```

Figure 9 **Requesting Historical Pricing Information**

```
public void RequestReferenceData(
  string security, DateTime start, DateTime end, string periodicity)
{
  Request request = _refDataService.
CreateRequest("HistoricalDataRequest");
  Element securities = request.GetElement("securities");
  securities.AppendValue(security);
  Element fields = request.GetElement("fields");
  fields.AppendValue("PX_LAST");
  request.Set("periodicityAdjustment", "ACTUAL");
  request.Set("periodicitySelection", periodicity);
  request.Set("startDate", string.Format(
    "{0}{1:D2}{2:D2}", start.Year, start.Month, start.Day));
  request.Set("endDate", string.Format(
    "{0}{1:D2}{2:D2}", end.Year, end.Month, end.Day));
  _session.SendRequest(request, null);
}
```

Figure 10 **Processing the Reference Data Service Response Message**

```
private void ProcessEventCallBack(Event eventObject, Session session)
{
  if (eventObject.Type == Event.EventType.RESPONSE)
  {
    List<DataPoint> series = new List<DataPoint>();
    foreach (Message msg in eventObject)
    {
      var element = msg.AsElement;
      var sd = element.GetElement("securityData");
      var fd = sd.GetElement("fieldData");
      for (int i = 0; i < fd.NumValues; i++)
      {
        Element val = (Element)fd.GetValue(i);
        var price = (double)val.GetElement("PX_LAST").GetValue();
        var dt = (Datetime)val.GetElement("date").GetValue();
        series.Add(new DataPoint(
          new DateTime(dt.Year, dt.Month, dt.DayOfMonth),
          price));
      }

      if (MarketDataEventHandler != null)
        MarketDataEventHandler(series);
    }
  }
}

private void OnMarketDataHandler(List<DataPoint> series)
{
  Dispatcher.BeginInvoke((Action)delegate
  {
    _chart.DataContext = series;
  });
}
```

The DEMO Plug-In

The DEMO plug-in application provided as part of the reference implementation can be launched using the command DEMO IBM Equity. It demonstrates how easy it is to start creating compelling applications targeted for financial professionals using our proposed architecture, the Bloomberg API and WPF.

The DEMO plug-in displays historical pricing information for a given security, which is functionality found in any financial application (see **Figure 6**). The DEMO plug-in uses the Bloomberg DAPI and requires a valid subscription to the Bloomberg Professional service. More information about the Bloomberg API can be found at openbloomberg.com/open-api.

The XAML shown in **Figure 7** defines the UI of the DEMO plug-in. The points of interest are instantiation of the Chart, LinearAxis, DateTimeAxis and LineSeries classes, as well as the setup of binding for LineSeries DependentValuePath and IndependentValuePath. We decided to use WpfToolkit for data visualization, because it works well in a partially trusted environment, provides required functionality and is licensed under the Microsoft Public License (MS-PL).

In order to access the Bloomberg API, a reference to the Bloomberglp.Blpapi assembly should be added to the project references list, and the following code has to be added to the list of using statements:

```
using Bloomberglp.Blpapi;
```

The application starts by establishing a new API session and obtaining a Reference Data Service (RDS) object, used for static pricing, historical data, and intraday tick and bar requests, as shown in **Figure 8**.

The next step is to request the historical pricing information for a given market security.

Create a Request object of type HistoricalDataRequest and construct the request by specifying the security, field (PX_LAST -last price), periodicity, and start and end dates in the format of YYYYMMDD (see **Figure 9**).

The final steps, shown in **Figure 10**, are to process the RDS response message asynchronously, construct a time series and visualize the data by setting the _chart.DataContext property.

Build Your Own

We presented a generic architecture for hosting untrusted .NET WPF-based plug-ins that was successfully used for implementation of the Bloomberg App Portal platform. The code download accompanying this article can help you build your own plug-in hosting solution, or it might inspire you to build an application for the Bloomberg App Portal using the Bloomberg API. ■

Gennady Slobodsky *is an R&D manager and architect at Bloomberg L.P. He specializes in building products utilizing Microsoft and open source technologies. A Gotham dweller, he enjoys long walks in Central Park and along Museum Mile.*

Levi Haskell *is an R&D team leader and architect at Bloomberg L.P. He enjoys dissecting .NET internals and building enterprise systems.*

Thanks to the following technical experts for reviewing this article: *Reid Borsuk (Microsoft) and David Wrighton (Microsoft)*

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.

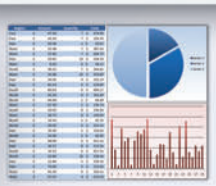
Excel Reporting for ASP.NET, WinForms, WPF and Silverlight

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

JAMES McCAFFREY

Converting Numeric Data to Categorical Data

A fundamental task in the field of machine learning is converting numeric data to categorical data. For example, if you have a data set of people's heights in inches, such as 59.5, 64.0 and 75.5, you might want to convert this numeric data into categorical data, for example 0, 1, and 2, to represent short, medium, and tall. Informally, this process is sometimes called binning data. In machine-learning literature, the process is usually called discretization of continuous data.

There are several scenarios where you might want to discretize data. Many machine-learning algorithms, such as naive Bayes classification and prediction, work only with categorical data. So if your raw data is numeric and you want to apply naive Bayes, you have to discretize the data. You might also have mixed numeric and categorical data, such as the data often found in an Excel spreadsheet. Very few machine-learning algorithms work with mixed data, so you can convert the numeric data to categorical data and then use a machine-learning algorithm that works with categorical data. Data clustering using category utility is an example.

Perhaps because the topic isn't very glamorous, there are few resources available that describe exactly how to implement discretization algorithms. In this article I'll present a powerful discretization algorithm. Although the ideas aren't new, to the best of my knowledge the implementation presented in this article hasn't been published before.

The best way to see where this article is headed is to examine the demo program shown in **Figure 1**. The demo sets up 20 data points that represent people's heights in inches. The raw data is shown in the histogram in **Figure 2**. The demo analyzes the data and creates a discretizer object, then shows an internal representation of the discretizer. The discretizer maintains a copy of the distinct values from the raw data in a sorted (from low values to high) array named data. The number of categories has been computed to be three and is stored in member variable k.

Each of the data points has been assigned to one of the three categories. Each category is encoded as a zero-based integer, 0 through 2, and assignment information is stored in an array named clustering. The first three data points (60.0, 61.0, 62.0) have been assigned to category 0. The next four data points (66.0, 67.0, 68.0, 69.0) have been assigned to category 1, and the final four data points (73.0, 74.0, 76.0, 78.0) have been assigned to category 2. The arithmetic mean of the first three data points in category 0 is 61.00, and the means of the data points in categories 1 and 2 are 67.50 and 75.25, respectively.

```
file:///C:/BinningData/bin/Debug/BinningData.EXE

Begin discretization of continuous data demo

Raw data:

66.00 66.00 66.00 67.00 67.00 67.00 67.00 68.00 68.00 69.00
73.00 73.00 73.00 74.00 76.00 78.00 60.00 61.00 62.00 62.00

Creating a discretizer on the raw data

Discretizer creation complete

Displaying internal structure of the discretizer:

------------------------------------------------------------

Distinct data:
60.00 61.00 62.00 66.00 67.00 68.00 69.00 73.00 74.00 76.00
78.00

k = 3

Clustering:
0 0 0 1 1 1 1 2 2 2 2

Means:
61.00 67.50 75.25

------------------------------------------------------------

Generating three existing and three new data values

Data values:

62.00 66.00 73.00 59.50 75.50 80.50

Discretizing the data:

62.00 -> 0
66.00 -> 1
73.00 -> 2
59.50 -> 0
75.50 -> 2
80.50 -> 2


End discretization demo
```

Figure 1 **Converting Numeric Data to Categorical Data**

After creating the discretizer object, the demo program sets up three existing data values (62.0, 66.0, 73.0) and three new data values (59.5, 75.5, 80.5). The point here is that sometimes you have a fixed data set, but in other scenarios, new data is generated dynamically and must be converted. The discretizer converts each of the six numeric data points to categorical data: 0, 1, 2, and 0, 2, 2, respectively.

This article assumes you have at least intermediate-level programming skills. I coded the discretization algorithm using C#, but you shouldn't have too much trouble refactoring the code to another language such as Python or Visual Basic. I've omitted most normal error-checking to keep the size of the code small

Code download available at archive.msdn.microsoft.com/mag201308TestRun.

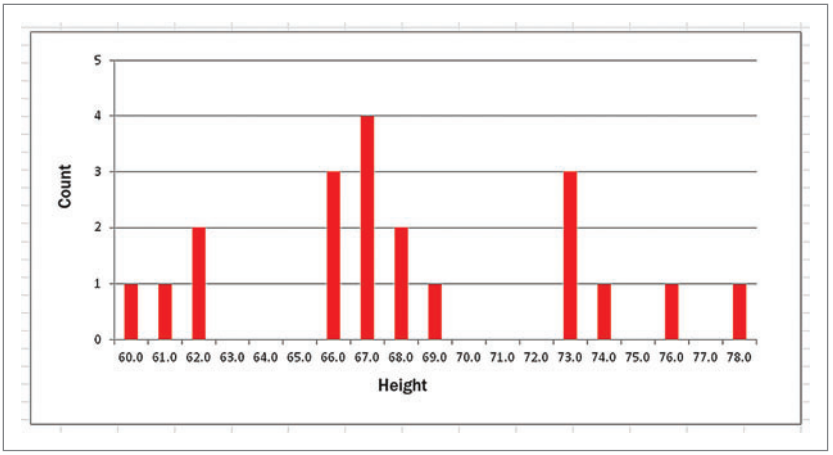


Figure 2 **The Raw Data to Categorize**

and the main ideas clear. The demo code is too long to present in its entirety in this article, but the entire source code is available at archive.msdn.microsoft.com/mag201308TestRun.

Not As Easy As It Seems

At first thought, converting numeric data to categorical data seems like an easy problem. One simple approach would be to divide the raw source data into equal intervals. For example, for the data in the demo and **Figure 2**, the range is 78.0 - 60.0 = 18.0. Dividing this by k = 3 intervals gives an interval width of 6.0 inches. So any data between 60.0 and 66.0 would be assigned to category 0, data between 66.0 and 72.0 would be assigned to category 1, and data between 72.0 and 78.0 would be assigned to category 2. The problem with this equal-interval approach is that it ignores natural breaks in the data. If you look at **Figure 2**, you'll see that a good discretizing algorithm should break data somewhere between 63.0 and 65.0, not at 66.0.

A second naive approach to discretization would be to use equal frequency. For the example data, because there are 11 distinct data points, with k = 3 categories, and 11 / 3 = 3 (with integer division truncation), you could assign the first three data points to category 0, the second three data points to category 1, and the last five data points to category 2. The equal-frequency approach also ignores natural breaks in the data.

The discretization algorithm presented in this article uses a data-clustering approach. The raw data is clustered using a k-means algorithm (which does take into account natural breaks in the data) and then uses the means generated by the clustering algorithm to categorize new data. For example, in **Figure 1**, the three means are 61.00, 67.50 and 75.25. To associate numeric value 62.5 to a categorical value, the discretizer determines which of the three mean values is closest to 62.5 (which is 61.0) and then assigns the cluster value associated with 61.0 (which is 0) as the category value for 62.5.

Figure 3 **The Demo Program Structure**

```
using System;
using System.Collections.Generic;
namespace BinningData
{
  class BinningProgram
  {
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin discretization of continuous data demo\n");

        double[] rawData = new double[20] {
          66, 66, 66, 67, 67, 67, 67, 68, 68, 69,
          73, 73, 73, 74, 76, 78,
          60, 61, 62, 62 };

        Console.WriteLine("Raw data:");
        ShowVector(rawData, 2, 10);

        Console.WriteLine("\nCreating a discretizer on the raw data");
        Discretizer d = new Discretizer(rawData);
        Console.WriteLine("\nDiscretizer creation complete");

        Console.WriteLine("\nDisplaying internal structure of the discretizer:\n");
        Console.WriteLine(d.ToString());
        Console.WriteLine("\nGenerating three existing and three new data values");
        double[] newData = new double[6] { 62.0, 66.0, 73.0, 59.5, 75.5, 80.5 };
        Console.WriteLine("\nData values:");
        ShowVector(newData, 2, 10);

        Console.WriteLine("\nDiscretizing the data:\n");
        for (int i = 0; i < newData.Length; ++i)
        {
          int cat = d.Discretize(newData[i]);
          Console.WriteLine(newData[i].ToString("F2") + " -> " + cat);
        }

        Console.WriteLine("\n\nEnd discretization demo");
        Console.ReadLine();
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
        Console.ReadLine();
      }

    } // Main

    public static void ShowVector(double[] vector, int decimals,
      int itemsPerRow) { . . }
  } // Program

  public class Discretizer
  {
    public Discretizer(double[] rawData) { . . }
    private static double[] GetDistinctValues(double[] array) { . . }
    private static bool AreEqual(double x1, double x2) { . . }
    public int Discretize(double x) { . . }
    public override string ToString() { . . }
    private void InitializeClustering() { . . }
    private int[] GetInitialIndexes() { . . }
    private int InitialCluster(int di, int[] initialIndexes) { . . }
    private void Cluster() { . . }
    private bool ComputeMeans() { . . }
    private bool AssignAll() { . . }
    private int MinIndex(double[] distances) { . . }
    private static double Distance(double x1, double x2) { . . }
  }
} // ns


The Discretizer Class
The Discretizer class has four data members:

private double[] data;
private int k;
private double[] means;
private int[] clustering;
```

Figure 4 **The Cluster Method**

```
private void Cluster()
{
  InitializeClustering();
  ComputeMeans();
  bool changed = true; bool success = true;
  int ct = 0;
  int maxCt = data.Length * 10; // Heuristic
  while (changed == true && success == true && ct < maxCt) {
    ++ct;
    changed = AssignAll();
    success = ComputeMeans();
  }
}
```

K-Means Clustering

The k-means clustering algorithm is quite simple. There are many variations of the algorithm. In its most basic form, for a given set of data points and a given number of clusters k, the initialization process assigns each data point to a randomly selected cluster. Then the means of the data points in each of the clusters are computed. Next, each data point is scanned and reassigned to the cluster that has a mean that's closest to the data point. The compute-means, reassign-cluster steps are repeated until no data point is reassigned to a new cluster.

Overall Program Structure

The demo program shown in **Figure 1** is a single console application. I used Visual Studio 2012, but the demo has no significant dependencies and any version of Visual Studio with the Microsoft .NET Framework 2.0 or greater should work. I created a new C# console application and named it BinningData. After the template code loaded, in the Solution Explorer window, I renamed file Program.cs to the more descriptive BinningProgram.cs, and Visual Studio automatically renamed the program class. At the top of the source code, I deleted all namespace references except the ones to System and Collections.Generic.

The overall program structure, with some minor edits, is presented in **Figure 3**. The key calling statements can be summarized like so:

```
double[] rawData = new int[] { 66.0, 66.0, ... };
Discretizer d = new Discretizer(rawData);
double numericVal = 75.5;
int catVal = d.Discretize(numericVal);
```

The Discretizer constructor uses numeric data to enable a Discretize method that accepts a numeric value, and which returns a zero-based integer categorical value. Notice that the Discretizer determines the number of categories automatically.

Array data holds the distinct values from the raw data and is used to create the clustering. Integer k is the number of clusters to assign the data to, which is also the number of data categories. The array named means has size k and holds the arithmetic means of the data points assigned to each cluster at any given time during the execution of the clustering algorithm.

The array named clustering encodes how the data is clustered at any given point in time. The index of array clustering indicates the index of a data point stored in array data, and the value in array clustering indicates the current cluster assignment. For example, if clustering[9] = 2, then the data point at data[9] is assigned to cluster 2.

Figure 5 **The ComputeMeans Method**

```
private bool ComputeMeans()
{
  double[] sums = new double[k];
  int[] counts = new int[k];

  for (int i = 0; i < data.Length; ++i)
  {
    int c = clustering[i]; // Cluster ID
    sums[c] += data[i];
    counts[c]++;
  }

  for (int c = 0; c < sums.Length; ++c)
  {
    if (counts[c] == 0)
      return false; // fail
    else
      sums[c] = sums[c] / counts[c];
  }

  sums.CopyTo(this.means, 0);
  return true; // Success
}
```

The Discretizer Constructor

The Discretizer constructor is defined as:

```
public Discretizer(double[] rawData)
{
  double[] sortedRawData = new double[rawData.Length];
  Array.Copy(rawData, sortedRawData, rawData.Length);
  Array.Sort(sortedRawData);
  this.data = GetDistinctValues(sortedRawData);
  this.clustering = new int[data.Length];
  this.k = (int)Math.Sqrt(data.Length); // heuristic
  this.means = new double[k];
  this.Cluster();
}
```

The first step is to extract the distinct values from the raw data. There are several ways to do this. The code here sorts a copy of the raw data then calls a helper method, GetDistinctValues. Once the distinct values have been determined, the clustering array can be allocated.

Here's method GetDistinctValues:

```
private static double[] GetDistinctValues(double[] array)
{
  List<double> distinctList = new List<double>();
  distinctList.Add(array[0]);
  for (int i = 0; i < array.Length - 1; ++i)
    if (AreEqual(array[i], array[i + 1]) == false)
      distinctList.Add(array[i+1]);

  double[] result = new double[distinctList.Count];
  distinctList.CopyTo(result);
  return result;
}
```

Because the source data has been sorted, the method can perform a single scan looking for instances where two consecutive values aren't equal. The raw data is type double, which means that comparing two values for exact equality can be dicey, so a helper method, AreEqual, is used:

```
private static bool AreEqual(double x1, double x2)
{
  if (Math.Abs(x1 - x2) < 0.000001) return true;
  else return false;
}
```

Method AreEqual uses an arbitrary closeness threshold of 0.000001. You might want to pass this value into the Discretizer object as an input parameter. A variable named epsilon is often used in this scenario.

After extracting the distinct values from the raw data, the next step is to determine k, the number of clusters, which is also the number of categories. Here a rule of thumb is used and k becomes the square root of the number of data items. An alternative is to write the constructor so that the value of k is passed in as a parameter. Determining the optimal value of k is essentially an unsolved machine-learning research problem.

After the value of k has been computed, the constructor allocates space for array means, and then calls the Cluster method. That method performs k-means clustering

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

on the data and the values in the final means array can be used to assign a category value to any numeric value.

The Clustering Algorithm

The heart of the Discretizer class is the code that performs k-means clustering. The Cluster method is listed in **Figure 4**.

Method Cluster is relatively short because it farms out all of the hard work to helper methods. Method InitializeClustering assigns all data points to an initial cluster. Then the means of the data points assigned to each cluster are computed using the clustering assignments.

Inside the main clustering algorithm loop, all data points are assigned to a cluster by method AssignAll. Method AssignAll calls helper methods Distance and MinIndex. Method Distance defines the distance between two data points:

```
private static double Distance(double x1, double x2)
{
  return Math.Sqrt((x1 - x2) * (x1 - x2));
}
```

Here, Euclidean distance (defined as the square root of the squared difference) is used. Because the data points are single values rather than vectors, the Euclidean distance is equivalent to Math.Abs(x1 - x2), so you might want to use this simpler computation.

The loop exits when there's no change in the clustering array, indicated by the return value of AssignAll, or when the means array can't be computed because the count of values assigned to a cluster is zero, or when a maximum loop counter value is reached. Here, maxCt is arbitrarily assigned a value of 10 times the number of data points. In general, the clustering algorithm here converges extremely quickly, and a loop exit condition of reaching maxCt is likely due to a logic error, so you might want to check for this.

Because the clustering process repeatedly reassigns values to clusters, it's possible that the number of values assigned to a cluster could become zero, making a mean impossible to compute. Helper method ComputeMeans attempts to compute all k means but returns false if a count is zero. The method is presented in **Figure 5**.

Initializing the Clustering

The clustering initialization process is a bit tricky. Suppose the data consists of 11 sorted values as shown in **Figure 1**, and k, the number of clusters, has been set to three. After initialization, the goal is for array member clustering to have three 0-values in cells 0 through 2, three 1-values in cells 3 through 5, and the remaining five 2-values in cells 6 through 10. In other words, the clustering should be evenly distributed by frequency.

The first step is to generate border values of {3, 6, 9}, which implicitly define intervals of 0-2, 3-5 and 6-greater. This is done by helper method GetInitialIndexes, which just divides the number of data points by the number of clusters:

```
private int[] GetInitialIndexes()
{
  int interval = data.Length / k;
  int[] result = new int[k];
  for (int i = 0; i < k; ++i)
    result[i] = interval * (i + 1);
  return result;
}
```

The second step is to define a helper method that computes the cluster value for a given data index value, using the border values:

```
private int InitialCluster(int di, int[] initialIndexes)
{
  for (int i = 0; i < initialIndexes.Length; ++i)
    if (di < initialIndexes[i])
      return i;
  return initialIndexes.Length - 1; // Last cluster
}
```

The third step is to assign all data indexes to a cluster:

```
private void InitializeClustering()
{
  int[] initialIndexes = GetInitialIndexes();
  for (int di = 0; di < data.Length; ++di)
  {
    int c = InitialCluster(di, initialIndexes);
    clustering[di] = c;
  }
}
```

In essence, the initialization process is the equal-frequency approach described previously in this article.

The Discretize Method

After the data has been clustered, the final values in the means array can be used to assign a zero-based categorical value to a numeric value. Method Discretize is:

```
public int Discretize(double x)
{
  double[] distances = new double[k];
  for (int c = 0; c < k; ++c)
    distances[c] = Distance(x, data[means[c]]);
  return MinIndex(distances);
}
```

The method computes the distance from the input value x to each of the k means and then returns the index of the closest mean, which is a cluster ID and is also a categorical value. For example, if the final means are 61.00, 67.50 and 75.25, and x is 70.00, the distance from x to mean[0] = sqrt((70.00 - 61.00)^2) = sqrt(81.00) = 9.00. Similarly, mean[1] = sqrt((70.00 - 67.50)^2) = 2.50, and mean[2] = sqrt((70.00 - 75.25)^2) = 5.25. The smallest distance is 2.50, which is at index [1], so 70.00 is assigned to categorical value 1.

Wrapping Up

The code presented in this article can be used as is to provide you with high-quality numeric-to-categorical data conversion for machine learning. You might want to encapsulate the Discretizer class in a class library rather than embedding the code directly into an application.

The primary feature you may want to customize is the determination of the number of categories, k. One possibility is to set a threshold value. Below the threshold, each data point generates a category value. For example, suppose you're dealing with people's ages. Suppose these ages range from 1 to 120. With only 120 distinct possible values, instead of computing k as the square root of 120 (which would give you 10 categories), you could just set k equal to 120. ■

Dr. James McCaffrey *works for Microsoft at the company's Redmond, Wash., campus. He has worked on several Microsoft products including Internet Explorer and MSN Search. He's the author of ".NET Test Automation Recipes" (Apress, 2006), and can be reached at jammc@microsoft.com.*

Thanks to the following technical expert for reviewing this article: *Richard Hughes (Microsoft Research)*

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.

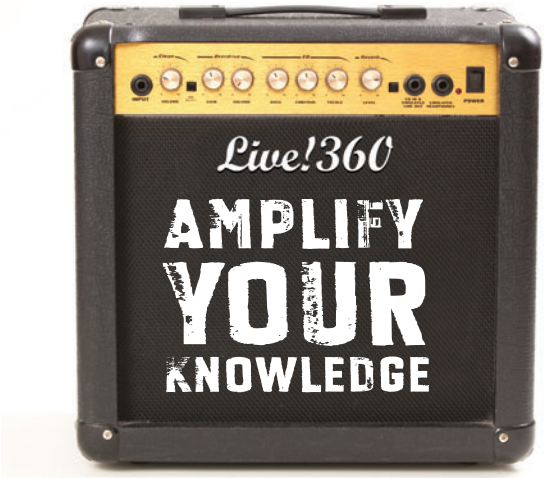
SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

ORLANDO | NOVEMBER 18-22, 2013

Royal Pacific Resort at Universal Orlando



LIVE! 360
IT EVENTS WITH PERSPECTIVE

WHETHER YOU ARE AN:

- 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.

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, 22 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!

REGISTER TODAY SAVE $400!

REGISTER BEFORE SEPTEMBER 12

Use Promo Code ORLAUG2

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

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive
facebook.com – Search "VSLive"
linkedin.com – Join the "Visual Studio Live" group!

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

vslive.com/orlando | live360events.com

TED NEWARD

Going Dynamic with the Gemini Library

Readers of my articles or blog posts will know this already, but for those who've stumbled onto this article (whether by accident or because they thought it was a horoscope), I tend to spend a lot of time looking at other languages and platforms. This is usually done in pursuit of concepts or ideas that help model software more effectively, efficiently or accurately.

One recent trend—though it's not all *that* recent—within the Web community has been the pursuit of "dynamic" or "scripting" languages, particularly two of them: Ruby (for which the Ruby on Rails framework, also sometimes called RoR, was written) and JavaScript (for which we have Node.js for executing server-side applications, along with hundreds of frameworks). Both of these languages are characterized by a lack of what we're used to in the C# and Visual Basic world: strict adherence to what a class defines as the sole definition for what an object consists of.

In JavaScript (a language that's sometimes characterized by smart-aleck presenters like myself as "Lisp with curly brackets"), for example, an object is a fully mutable entity, meaning you can add properties or methods as needed (or wanted):

```
var myCar = new Object();
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
myCar.makeSounds = function () {
  console.log("Vroom! Vroom!")
}
```

The object myCar, when first constructed, has no properties or methods on it—these are implicitly added when the data values ("Ford," "Mustang," 1969 and the function) are set to those names (make, model, year and makeSounds). In essence, each object in JavaScript is just a dictionary of name/value pairs, where the value of the pair can be either a data element or a function to be invoked. Among language designers, the ability to play with type information like this is often called a Metaobject Protocol (MOP), and a constrained subset of this is often called aspect-oriented programming (AOP). It's a flexible, powerful approach to objects, but one that's very different from that of C#. Rather than try to create a complex class hierarchy in which you try to capture every possible variation through inheritance, as traditional C# object design would suggest, the MOP approach says that things in the real world aren't all the exact same (except for their data, of course), and the way in which you model them shouldn't be, either.

Developers who've been part of the Microsoft .NET Framework community for many years now will recall that an earlier version of C# introduced the *dynamic* keyword/type, which allows you to declare a reference to an object whose members are discovered at run time, but this is a different problem. (The dynamic feature set makes it easier to write reflection-style code, not create MOP kinds of objects.) Fortunately, C# developers have the option of both: traditional static type definitions, through the standard C# class design mechanisms; or flexible type definitions, through an open source library called Gemini that builds on top of dynamic functionality to give you near-JavaScriptian features.

Gemini Basics

Like a lot of the packages I've discussed in this column, Gemini is available through NuGet: "Install-Package Gemini" in the Package Manager Console brings the goodness into your project. Unlike other packages you've seen, however, when Gemini is installed into the project it doesn't bring an assembly or two (or three or more). Instead, it brings with it several source files and puts them into a folder called "Oak" and adds them directly to the project. (As of this writing, Gemini 1.2.7 consists of four files: Gemini.cs, GeminiInfo.cs, ObjectExtensions.cs and a text file containing the release notes.) The reason for the folder named Oak is actually very reasonable: Gemini is actually a subset of a larger project (called, not surprisingly, Oak) that brings a lot of this dynamic programming goodness to the ASP.NET MVC world—I'll explore the larger Oak package in a future column.

Of its own accord, the fact that Gemini is delivered as source really isn't a big deal—the code lives in its own namespace (Oak) and will simply be compiled into the project as the rest of the source files are. On a practical note, however, having the source files makes it absurdly easy to step through the Gemini source code when something goes wrong, or even thumb through the code just to see what's available, because IntelliSense is sometimes completely defeated by the use of the *dynamic* keyword/type.

Getting Started

Again, as is my habit, I begin by creating a unit test project in which to write some exploration tests; into that project I install Gemini and test it out by creating a simple "hello world"-like test:

```
[TestMethod]
public void CanISetAndGetProperties()
{
  dynamic person = new Gemini(
    new { FirstName = "Ted", LastName = "Neward" });

  Assert.AreEqual(person.FirstName, "Ted");
  Assert.AreEqual(person.LastName, "Neward");
}
```

Figure 1 **A Method That Takes an Object and Prints Out a Message**

```
string SayHello(dynamic thing)
{
  return String.Format("Hello, {0}, you are {1} years old!",
    thing.FirstName, thing.Age);
}
[TestMethod]
public void DemonstrateStructuralTyping()
{
  dynamic person = new Gemini(
    new { FirstName = "Ted", LastName = "Neward", Age = 42 });
    string message = SayHello(person);
    Assert.AreEqual("Hello, Ted, you are 42 years old!", message);

    dynamic pet = new Gemini(
      new { FirstName = "Scooter", Age = 3, Hunter = true });
  string otherMessage = SayHello(pet);
  Assert.AreEqual("Hello, Scooter, you are 3 years old!", otherMessage);
}
```

What's happening here is subtle, but powerful: Gemini, the object on the other side of the "person" reference, is a type that's essentially empty of all properties or methods, until those members are either assigned to (such as in the case of the preceding code) or explicitly added to the object through the methods SetMember and GetMember, like so:

```
[TestMethod]
public void CanISetAndGetPropertiesDifferentWays()
{
  dynamic person = new Gemini(
    new { FirstName = "Ted", LastName = "Neward" });

  Assert.AreEqual(person.FirstName, "Ted");
  Assert.AreEqual(person.LastName, "Neward");

  person = new Gemini();
  person.SetMember("FirstName", "Ted");
  person.SetMember("LastName", "Neward");

  Assert.AreEqual(person.GetMember("FirstName"), "Ted");
  Assert.AreEqual(person.GetMember("LastName"), "Neward");
}
```

While I do this for data members here, it's also equally easy to do this for behavioral members (that is, methods) by setting them to instances of DynamicMethod (which returns void) or DynamicFunction (which expects to return a value), each of which takes no parameters. Or you can set them to their "WithParam" partners, if the method or function can take a parameter, like so:

```
[TestMethod]
public void MakeNoise()
{
  dynamic person =
    new Gemini(new { FirstName = "Ted", LastName = "Neward" });
  person.MakeNoise =
    new DynamicFunction(() => "Uh, is this thing on?");
  person.Greet =
    new DynamicFunctionWithParam(name => "Howdy, " + name);

    Assert.IsTrue(person.MakeNoise().Contains("this thing"));
}
```

One interesting little tidbit arises out of the Gemini library, by the way: Gemini objects (absent any kind of alternative implementation) use "structural typing" to determine whether they're equal or if they satisfy a particular implementation. Contrary to OOP-type systems, which use the inheritance/IS-A test to determine whether a given object can satisfy the restrictions placed on an object parameter's type, structurally typed systems instead just ask whether the object passed in has all the requirements (members, in this case) needed to make the code run correctly. Structural typing, as it's known among the functional languages, also goes by the term "duck typing" in dynamic languages (but that doesn't sound as cool).

Consider, for a moment, a method that takes an object and prints out a friendly message about that object, as shown in **Figure 1**.

Normally, in a traditional object-oriented hierarchy, Person and Pet would likely come from very different branches of the inheritance tree—people and pets don't generally share a lot of common attributes in a software system (despite what cats think). In a structurally or duck-typed system, however, less work needs to go in to making the inheritance chain deep and all-encompassing—if there's a human that also hunts, then, hey, that human has a "Hunter" member on it, and any routine that wants to check on the Hunter status of the object passed in can use that member, whether it's a human, cat or Predator drone.

Interrogation

The trade-off in the duck-typing approach, as many will note, is that the compiler can't enforce that only certain kinds of objects can be passed in, and the same is true of Gemini types—particularly because most Gemini code idiomatically stores the object behind a *dynamic* reference. You need to take a little extra time and effort to ensure the object being handed in satisfies the requirements, or else face some runtime exceptions. This means interrogating the object to see if it has the necessary member, which is done in Gemini using the RespondsTo method; there are also some methods to return the various members that Gemini recognizes as being a part of a given object.

Consider, for example, a method that expects an object that knows how to hunt:

Figure 2 **Dynamic Programming When It Works**

```
[TestMethod]
public void AHuntingWeWillGo()
{
  dynamic pet = new Gemini(
    new
    {
      FirstName = "Scooter",
      Age = 3,
      Hunter = true,
      Hunt = new DynamicFunction(() => new Random().Next(4))
    });
  int hunted = Hunt(pet);
  Assert.IsTrue(hunted >= 0 && hunted < 4);

  // ...
}
```

Figure 3 **Dynamic Programming When It Fails**

```
[TestMethod]
public void AHuntingWeWillGo()
{
  // ...

  dynamic person = new Gemini(
    new
    {
      FirstName = "Ted",
      LastName = "Neward",
      Age = 42
    });
  hunted = Hunt(person);
  Assert.IsTrue(hunted >= 0 && hunted < 4);
}
```

```
int Hunt(dynamic thing)
{
  return thing.Hunt();
}
```

When Scooter is passed in, things work fine, as shown in **Figure 2**.

But when something that doesn't know how to hunt is passed in, exceptions will result, as shown in **Figure 3**.

To prevent this, the Hunt method should test to see if the member in question exists by using the RespondsTo method. This is a simple wrapper around the TryGetMember method and is intended for simple Boolean yes/no responses:

```
int Hunt(dynamic thing)
{
  if (thing.RespondsTo("Hunt"))
    return thing.Hunt();
  else
    // If you don't know how to hunt, you probably can't catch anything.
  return 0;
}
```

By the way, if this all seems like fairly simple boilerplate or a wrapper around a Dictionary<string,object>, that's not an incorrect assessment—underlying the Gemini class is that exact Dictionary interface. But the wrapper types help ease some of the type system gyrations that would otherwise be necessary, as does the use of the *dynamic* keyword.

But what happens when several objects share similar kinds of behavior? For example, four cats all know how to hunt, and it would be somewhat inefficient to write a new anonymous method definition for all four, particularly as all four share feline hunting instincts. In traditional OOP this wouldn't be a problem, because they'd all be members of the Cat class and thus share the same implementation. In MOP systems, such as JavaScript, there's typically a mechanism to allow an object to defer or "chain" a property or request call to another object, called a "prototype." In Gemini you use an interesting combination of static typing and MOP called "extensions."

Prototype

First, you need a base type that identifies cats:

```
public class Cat : Gemini
{
  public Cat() : base() { }
  public Cat(string name) : base(new { FirstName = name }) { }
}
```

Figure 4 **Writing Methods Without Writing Methods**

```
[TestMethod]
public void HtmlizeKittyNames()
{
  Gemini.Extend<Cat>(cat =>
  {
    cat.MakeNoise = new DynamicFunction(() => "Meow");
    cat.Hunt = new DynamicFunction(() => new Random().Next(4));

    var members =
      (cat.HashOfProperties() as IDictionary<string, object>).ToList();
    members.ForEach(keyValuePair =>
    {
      cat.SetMember(keyValuePair.Key + "Html",
        new DynamicFunction( () =>
          Htmlize(cat.GetMember(keyValuePair.Key))));
    });
  });

  dynamic scooter = new Cat("Sco<tag>oter");
  Assert.AreEqual("Sco<tag>oter", scooter.FirstName);
  Assert.AreEqual("Sco&lt;tag&gt;oter", scooter.FirstNameHtml());
}
```

Note that the Cat class inherits from Gemini, which is what will enable the Cat class to have all the dynamic flexibility that's been discussed so far—in fact, the second Cat constructor uses the same Gemini constructor that's been used to create all the dynamic instances thus far. This means that all of the preceding prose still holds for any Cat instance.

But Gemini also allows us to make declarations of how Cats can be "extended," so that every Cat gains the same functionality without having to explicitly add it to each instance.

Extending a Class

For a practical use of this functionality, presume that this is a Web application that's being developed. Frequently, you need to HTML-escape the name values being stored and returned, in order to avoid accidentally allowing HTML injection (or worse, SQL injection):

```
string Htmlize(string incoming)
{
  string temp = incoming;
  temp = temp.Replace("&", "&amp;");
  temp = temp.Replace("<", "&lt;");
  temp = temp.Replace(">", "&gt;");
  return temp;
}
```

This is a pain to remember on every model object defined in the system; fortunately, MOP allows you to systematically "reach in" and define new behavioral members on the model objects, as shown in **Figure 4**.

Essentially, the Extend call is adding new methods to every Cat type, suffixed with "Html," so the property FirstName can be accessed in an HTML-safe version by calling the FirstNameHtml method instead.

And this can be done entirely at run time for any Gemini-inheriting type in the system.

Persistence and More

Gemini isn't intended to replace the entirety of the C# environment with a bunch of dynamically resolved objects—far from it. In its home usage, inside of the Oak MVC framework, Gemini is used to add persistence and other useful behavior to model classes (among other things) and to add validation without cluttering the user's code or requiring partial classes. Even outside of Oak, however, Gemini represents some powerful design mechanics, which some readers may remember from part 8 of my Multiparadigmatic .NET series from a while back (msdn.microsoft.com/magazine/hh205754).

Speaking of Oak, that's on tap for next time, so stick around to see how all this dynamic stuff plays out in a real-world scenario.

Happy coding! ■

Ted Neward *is a principal with Neward & Associates LLC. He has written more than 100 articles and authored and coauthored a dozen books, including "Professional F# 2.0" (Wrox, 2010). He is an F# MVP and noted Java expert, and speaks at both Java and .NET conferences around the world. He consults and mentors regularly—reach him at ted@tedneward.com or Ted.Neward@neudesic.com if you're interested in having him come work with your team. He blogs at blogs.tedneward.com and can be followed on Twitter at twitter.com/tedneward.*

Thanks to the following technical expert for reviewing this article: *Amir Rajan (Improving Enterprises)*

RACHEL APPEL

Navigation Essentials in Windows Store Apps

Ensuring that users can quickly and painlessly access the content they want, when they want, is an essential part of any modern app. This means navigation has to stay out of the way until it's wanted and not force users into hard-to-reverse choices. This is a UI design technique called "content over chrome." The best way to enforce this design paradigm is to bake most of the navigation directly into the content. This is easy to do in Windows Store apps, as they follow this navigational design principle.

Windows Store App Navigation Overview

In Windows Store apps, developers bake navigation into the content, creating a smooth, flowing experience for users while they're wading through content, using one of these navigation patterns:

- Hierarchical
- Flat
- Split/single-page application (SPA)

> In Windows Store apps, developers bake navigation into the content, creating a smooth, flowing experience for users.

Visual Studio 2012 supports all three navigational models through a set of basic app templates: Grid, Split and Navigation (HTML only). In HTML apps, each template contains a navigator.js file with code that performs navigation between pages as well as code that manages the back button. In XAML apps, navigation is built into the event model.

The Hierarchical Navigation Pattern

Relational data makes a great candidate for hierarchical navigation. This is because most often you must traverse relational content in a particular order—for example, in a master/detail scenario—or it doesn't make sense. This isn't always the case, though, and that's where flat navigation comes into play (more on this shortly). Even though related data can be several levels deep, putting more than three levels directly into a navigation system makes it extremely difficult on users, according to usability studies. After carefully researching navigation usability, the Windows Design Language team created the hierarchical navigation system consisting of these three navigational levels:

- **Hub:** This is the opening stage of data containing master data at the front and center. This is usually a list of master items, such as music artists. From this list, the user can drill into the specifics of each artist. Upon its release, Visual Studio 2013 (currently in preview) will introduce a new Hub template for creating robust navigation scenarios.
- **Section:** This is a level-two view that includes all members of a specific group that the user selected from the Hub page. An example of section navigation would be browsing all the albums of a particular artist.
- **Details:** This includes the nitty-gritty details of one particular item—for example, information about one particular song in an album or an individual photo in a collection.

Examples of these three styles are shown in the context of a CNN app in **Figures 1**, **2** and **3**, respectively.

If you still believe you need more levels of navigation, consider using the navigational helpers such as semantic zoom or dropdown menus, or other UI components for content filtering and sorting in addition to your regular navigation. I'll look more closely at these navigational aids later in this article.

The Visual Studio Grid template has the hierarchical navigation with all three levels built right in. All you need to do is retrieve data and plug it into the template for it to work. The data can be as simple as a WinJS.Binding.List or a plain old CLR object (POCO) in C# apps, the same structures as used in the Grid, Split and Navigation app templates in Visual Studio.

Notice that all sections of apps that use a hierarchical pattern contain a back button, which is a critical part of navigation in Windows Store apps, as well as Web apps. Having a way to back out and undo the previous navigation command enhances and eases the UX.

The Flat Navigational Pattern

Sporting a navigation bar at the top of the screen, flat navigation apps simply show several individual choices that take you directly to some content that might or might not be related. **Figure 4** illustrates a great example of this in a weather app, with tabs at the top of the screen. As you tap or click one, you go directly to the associated page. Flat navigation is great for nonrelational content. For example, the tabs in the weather app take you to unrelated pages of your choosing, rather than applying any formal structure.

If you've determined that you need deep levels of hierarchical navigation, starting with the flat navigation system could help. Use the top navigation bar as a hub page. When the user makes a choice, the app should navigate to the requested place. Using this technique adds just one level of navigation, so you might want to incorporate navigational helpers as well.

The Blank template in Visual Studio is the most conducive to flat navigation, although you can add a top nav bar to any project type to institute flat navigation.

> Flat navigation is great for nonrelational content.

Incorporating flat navigation into your app is as easy as adding a Windows Library for JavaScript (WinJS) NavBar control, a new control from Windows 8.1. **Figure 5** shows an example in HTML.

The NavBar control is available in WinJS only, so to create a top nav bar in XAML, you must code an app bar and style it for display at the top of the page, as this example demonstrates:

```
<Page.TopAppBar>
  <AppBar x:Name="topAppBar">
    <Grid>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
        <Button Style="{StaticResource SaveAppBarButtonStyle}"
          Click="Button_Click"/>
        <Button Style="{StaticResource UploadAppBarButtonStyle}"
          Click="Button_Click"/>
      </StackPanel>
    </Grid>
  </AppBar>
</Page.TopAppBar>
```

You can have both a top nav bar and a standard bottom app bar, and you can add menus, dropdowns, and other navigational helpers to both.

The Split/SPA Navigational Pattern

Just as it sounds, an SPA app consists of one page for the entire app. Technically, SPAs aren't really just one gigantic page; rather, they consist of one main page where the app loads sections of content and commands at run time, dynamically, driven by user activity. In Windows Store apps built with JavaScript, you can implement SPA apps by managing app components called page fragments.

The Split navigation template in Visual Studio 2012 reveals an SPA template similar to the Grid template; however, the Split template contains two—rather than three—levels of navigation. This means instead of hub/section/details, the Split template starts at the section level, showing a listing of items, and as the user selects one, its details load into the page alongside the other items.

SPA navigation in HTML apps is done by first creating a or container element into which the other pages can load themselves. You can use the WinJS PageControlNavigator control (defined in navigator.js) to create this container:

```
<div id="contenthost" data-win-control="Application.PageControlNavigator"
  data-win-options="{home: '/pages/home/home.html'}"></div>
```

Once the PageControlNavigator is in place (by default in the Grid, Split and Navigation templates), the code from \js\navigator.js does all the work of loading pages into the PageControlNavigator without modifications on your end.

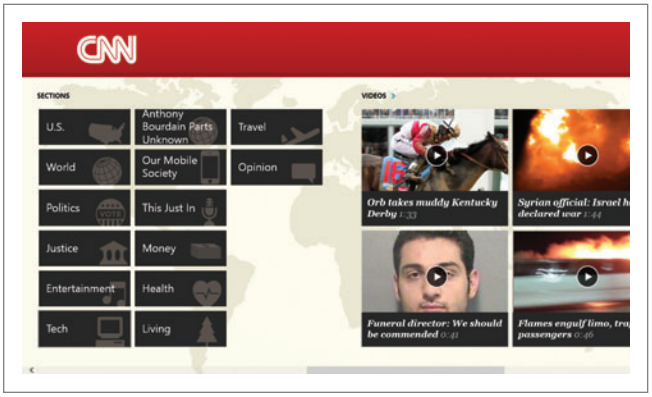


Figure 1 **The Hub Navigation Level**

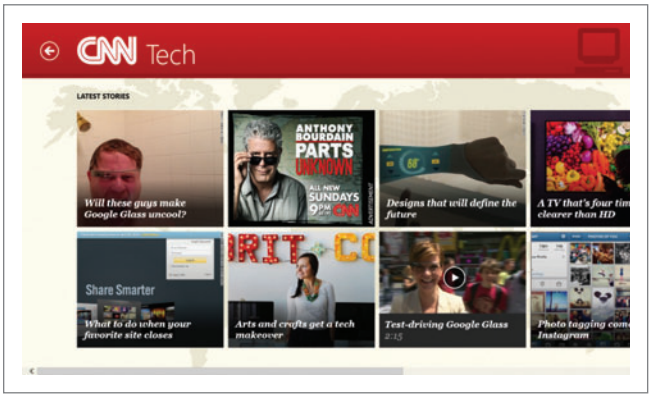


Figure 2 **The Section Navigation Level**

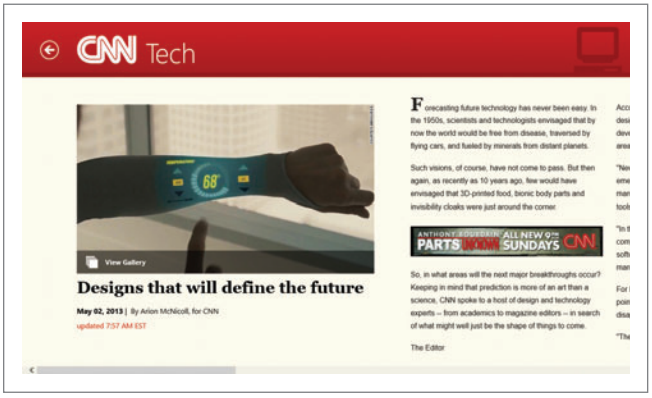


Figure 3 **The Details Navigation Level**

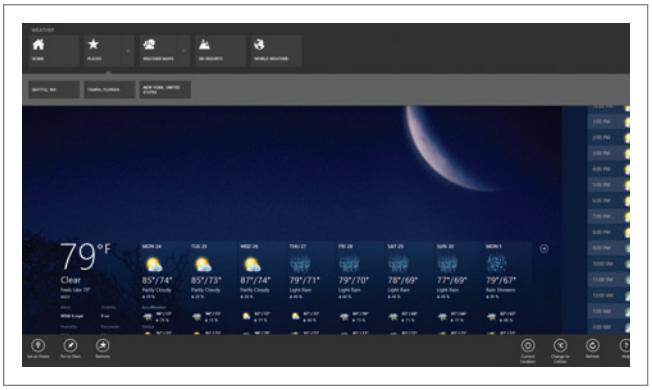


Figure 4 **Flat Navigation in a Weather App**

Figure 5 **Using the WinJS NavBar Control**

```
<div id="NavBar" data-win-control="WinJS.UI.NavBar">
  <div id="GlobalNav" data-win-control="WinJS.UI.NavBarContainer">
    <div data-win-control="WinJS.UI.NavBarCommand" data-win-options="{
      label: 'Home',
      icon: 'url(../images/homeIcon.png)',
      location: '/html/home.html',
      split: false
    }">
    </div>
    <div data-win-control="WinJS.UI.NavBarCommand" data-win-options="{
      label: 'Your apps',
      icon: WinJS.UI.AppBarIcon.favorite,
      location: '/html/yourapps.html',
      split: false
    }">
    </div>
  </div>
</div>
```

The SPA pattern discussed here works great in Web sites and other app types—including Windows Store apps, of course—and the SPA app is fast becoming a favorite of modern app developers, including those on other platforms.

Navigating with Semantic Zoom

Have you ever used a data-intensive app in which menus or links don't seem to help get you to the data you need? That's where semantic zoom comes into play. Using semantic zoom allows you to repurpose and visualize data in a different and more navigable way. This could be either looking at data from a 13,000-foot overview (this is how the Windows start page itself works), or it can be showing the data in aggregate groupings so users can find what they're looking for quickly. Netflix, as demonstrated in **Figure 6**, makes good use of semantic zoom to indicate movie categories. Semantic zoom is particularly useful when you have a large volume of data.

> Using semantic zoom allows you to repurpose and visualize data in a different and more navigable way.

You can implement semantic zoom in Windows Store apps as a control in both WinJS and XAML apps. Here's an example in HTML/JavaScript:

```
<div data-win-control="WinJS.UI.SemanticZoom">
  <!—Original view of the data, for example, the list of all movies -->
  <!-- The zoomed-out version of the data, as shown in Figure 1. -->
</div>
```

Here's an example in XAML/C#:

```
<SemanticZoom>
  <SemanticZoom.ZoomedInView>
    <!—Original view of the data, for example, the list of all movies -->
  </SemanticZoom.ZoomedInView>
  <SemanticZoom.ZoomedOutView>
    <!-- The zoomed-out version of the data, as shown in Figure 1. -->
  </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

Users can invoke semantic zoom by using a pinch gesture with two fingers on touchscreen devices or with the mouse+scroll wheel. For more on the SemanticZoom control and other Windows Store app controls, see my column, "Mastering Controls and Settings in Windows Store Apps Built with JavaScript," at msdn.microsoft.com/magazine/dn296546.

Navigation by Sorting, Filtering and Searching

Sure, you could argue that filtering isn't true navigation, but being able to sort data will most certainly help users limit their selections and find desired data quickly with less hassle, which is the point of navigation. At the very minimum, sorting and filtering can provide much-needed enhancements to standard app navigation. In the case of Windows Store apps, filtering is a means to enhance the app's primary navigational scheme. Fine-tuning the visualization via small UI controls really makes the UX precise and enjoyable for the user.

In Windows Store apps built with JavaScript, you can implement dropdowns for sorting and filtering by using the standard HTML <select> element, like this:

```
<select>
  <option>Appels</option>
  <option>Bananas</option>
  <option>Grapes</option>
  <option>Oranges</option>
  <option>Pears</option>
</select>
```

In XAML apps you call the same control a ComboBox, and it looks similar to this code:

```
<ComboBox x:Name="Fruits" SelectionChanged="Fruits_SelectionChanged" >
  <x:String>Appels</x:String>
  <x:String>Bananas</x:String>
  <x:String>Oranges</x:String>
  <x:String>Grapes</x:String>
  <x:String>Pears</x:String>
</ComboBox>
```

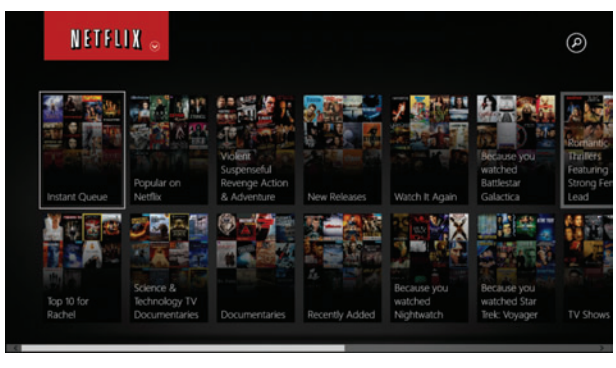


Figure 6 **Netflix Makes Great Use of Semantic Zoom**

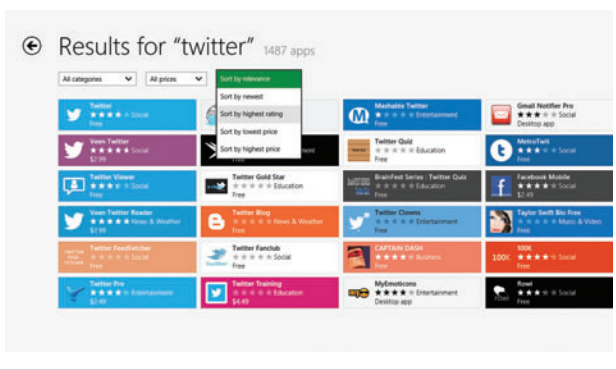


Figure 7 **The Windows Store App Filters**

The Windows Store app itself makes great use of dropdowns to filter out apps. Navigate to any category and you'll find filters for price and relevance as illustrated in **Figure 7**. During searches in the store, even more filters appear.

Finally, let's not forget how search is an important part of navigation. Everybody uses search—and not just through the popular Web search engines. Everybody loves getting specific content delivered right to their fingertips, so make sure that your apps provide this door-to-door service for users. Because search, like the settings charm, is part of the OS, in Windows Store apps you can implement search through a contract.

Other Navigational Helpers

Horizontal scrolling might not seem like it's part of navigation, but it can be. In Windows Store apps, the panoramic view means that you can add completely separate sections of data all into one, organized, single view. This allows users to scroll to different sections without actually navigating, per se, and is often a staple of news apps where some content is text and other is video. Combine panorama with semantic zoom, and users can jump right to sections of the data they need. Consider apps such as the CNN and ABC news apps, Bing Finance, Netflix and others that extensively scroll long horizontally.

In addition to these navigation helpers, using built-in features of Windows 8 for some activity will cut down on the required amount of navigational elements in the app. For example, rather than building into the app an about, settings or other informational page (traditionally located in the Windows Help menu in programs in previous versions of Windows), you can take advantage of the settings charm to display help and app info. This means you don't have to supply yet another navigational menu and option, and users benefit by having a consistent theme throughout all Windows Store apps to get to settings and the like.

No More Lost and Confused Users

A rock-solid navigational scheme is an important aspect of modern app—and particularly Windows Store app—development. Without easy-to-use navigation, users will become lost and confused. Fortunately, the built-in templates of Windows Store apps in both HTML and XAML help you create easy-to-use apps for users. ■

Rachel Appel *is a consultant, author, mentor and former Microsoft employee with more than 20 years of experience in the IT industry. She speaks at top industry conferences such as Visual Studio Live!, DevConnections, MIX and more. Her expertise lies within developing solutions that align business and technology focusing on the Microsoft dev stack and open Web. For more about Appel, visit her Web site at rachelappel.com.*

Thanks to the following technical experts at Microsoft for reviewing this article: *Megan Bates, Ross Heise, Maria Kang and Eric Schmidt*

CHARLES PETZOLD

Finger Painting with Direct2D Geometries

As OSes have evolved over the years, so have the basic archetypal applications that every developer should know how to code. For old command-line environments, a common exercise was a hex dump—a program that lists the contents of a file in hexadecimal bytes. For graphical mouse-and-keyboard interfaces, calculators and notepads were popular.

In a multi-touch environment like Windows 8, I'd nominate two archetypal applications: photo scatter and finger paint. The photo scatter is a good way to learn how to use two fingers to scale and rotate visual objects, while finger painting involves tracking individual fingers to draw lines on the screen.

Figure 1 **A BasicFingerPaint Drawing**

I explored various approaches to Windows 8 finger painting in Chapter 13 of my book, "Programming Windows, 6th Edition" (Microsoft Press, 2012). Those programs used only the Windows Runtime (WinRT) for rendering the lines, but now I'd like to revisit the exercise and use DirectX instead. This will be a good way to become familiar with some important aspects of DirectX, but I suspect it will also eventually allow us some additional flexibility not available in the Windows Runtime.

The Visual Studio Template

As Doug Erickson discussed in his March 2013 article, "Using XAML with DirectX and C++ in Windows Store Apps" (msdn.microsoft.com/magazine/jj991975), there are three ways to combine XAML and DirectX within a Windows Store application. I'll be using the approach that involves a SwapChainBackgroundPanel as the root child element of a XAML Page derivative. This object serves as a drawing surface for Direct2D and Direct3D graphics, but it can also be overlaid with WinRT controls, such as application bars.

Visual Studio 2012 includes a project template for such a program. In the New Project dialog box, choose Visual C++, and Windows Store at the left, and then the template called Direct2D App (XAML). The other DirectX template is called Direct3D App and creates a DirectX-only program without any WinRT controls or graphics. However, these two templates are somewhat misnamed because you can do 2D or 3D graphics with either one of them.

Code download available at archive.msdn.microsoft.com/mag201308DXF.

The Direct2D App (XAML) template creates a simple Windows Store application with program logic divided between a XAML-based UI and DirectX graphics output. The UI consists of a class named DirectXPage that derives from Page (just as in a normal Windows Store application) and consists of a XAML file, header file and code file. You use DirectXPage for handling user input, interfacing with WinRT controls, and displaying XAML-based graphics and text. The root element of DirectXPage is the SwapChainBackgroundPanel, which you can treat as a regular Grid element in XAML and as a DirectX rendering surface.

The project template also creates a class named DirectXBase that handles most of the DirectX overhead, and a class named SimpleTextRenderer that derives from DirectXBase and performs application-specific DirectX graphics output. The name SimpleTextRenderer refers to what this class does within the application created from the project template. You'll want to rename this class, or replace it with something that has a more appropriate name.

In a multi-touch environment like Windows 8, I'd nominate two archetypal applications: photo scatter and finger paint.

From Template to Application

Among the downloadable code for this column is a Visual Studio project named BasicFingerPaint that I created using the Direct2D (XAML) template. I renamed SimpleTextRenderer to FingerPaintRenderer and added some other classes as well.

The Direct2D (XAML) template implies an architecture that separates the XAML and DirectX parts of the application: All the application's DirectX code should be restricted to DirectXBase (which you shouldn't need to alter), the renderer class that derives from DirectXBase (in this case FingerPaintRenderer), and any other classes or structures these two classes might need. Despite its

Figure 2 **Pointer Event Methods Making Calls to the Renderer Class**

```
void DirectXPage::OnPointerPressed(PointerRoutedEventArgs^ args)
{
  NamedColor^ namedColor = dynamic_cast<NamedColor^>(colorComboBox->SelectedItem);
  Color color = namedColor != nullptr ? namedColor->Color : Colors::Black;

  int width = widthComboBox->SelectedIndex !=
    -1 ? (int)widthComboBox->SelectedItem : 5;

  m_renderer->BeginStroke(args->Pointer->PointerId,
                          args->GetCurrentPoint(this)->Position,
                          float(width), color);

  CapturePointer(args->Pointer);
}

void DirectXPage::OnPointerMoved(PointerRoutedEventArgs^ args)
{
  IVector<PointerPoint^>^ pointerPoints = args->GetIntermediatePoints(this);

  // Loop backward through intermediate points
  for (int i = pointerPoints->Size - 1; i >= 0; i--)
    m_renderer->ContinueStroke(args->Pointer->PointerId,
                               pointerPoints->GetAt(i)->Position);
}

void DirectXPage::OnPointerReleased(PointerRoutedEventArgs^ args)
{
  m_renderer->EndStroke(args->Pointer->PointerId,
                        args->GetCurrentPoint(this)->Position);
}

void DirectXPage::OnPointerCaptureLost(PointerRoutedEventArgs^ args)
{
  m_renderer->CancelStroke(args->Pointer->PointerId);
}

void DirectXPage::OnKeyDown(KeyRoutedEventArgs^ args)
{
  if (args->Key == VirtualKey::Escape)
      ReleasePointerCaptures();
}
```

name, DirectXPage should *not* need to contain any DirectX code. Instead, DirectXPage instantiates the renderer class, which it saves as a private data member named m_renderer. DirectXPage makes many calls into the renderer class (and indirectly to DirectXBase) to display graphical output and notify DirectX of window size changes and other important events. The renderer class doesn't call into DirectXPage.

In the DirectXPage.xaml file I added combo boxes to the application bar that let you select a drawing color and line width, and buttons to save, load, and clear drawings. (The file I/O logic is extremely rudimentary and doesn't include amenities, such as warning you if you're about to clear a drawing you haven't saved.)

Figure 3 **The Renderer's StrokeInfo Structure and Two Collections**

```
struct StrokeInfo
{
  StrokeInfo() : Color(0, 0, 0),
                 Geometry(nullptr)
  {
  };
  std::vector<D2D1_POINT_2F> Points;
  Microsoft::WRL::ComPtr<ID2D1PathGeometry> Geometry;
  float Width;
  D2D1::ColorF Color;
};

std::vector<StrokeInfo> completedStrokes;
std::map<unsigned int, StrokeInfo> strokesInProgress;
```

As you touch a finger to the screen, move it, and lift it, PointerPressed, PointerMoved, and PointerReleased events are generated to indicate the finger's progress. Each event is accompanied by an ID number that lets you track individual fingers, and a Point value indicating the current position of the finger. Retain and connect these points, and you've rendered a single stroke. Render multiple strokes, and you have a complete drawing. **Figure 1** shows a BasicFingerPaint drawing consisting of nine strokes.

> The PointerId object is a unique integer that differentiates fingers, mouse and pen.

In the DirectXPage codebehind file, I added overrides of the Pointer event methods. These methods call corresponding methods in FingerPaintRenderer that I named BeginStroke, ContinueStroke, EndStroke and CancelStroke, as shown in **Figure 2**.

The PointerId object is a unique integer that differentiates fingers, mouse and pen. The Point and Color values passed to these methods are basic WinRT types, but they're not DirectX types. DirectX has its own point and color structures named D2D1_POINT_2F and D2D1::ColorF. DirectXPage doesn't know anything about DirectX, so the FingerPaintRenderer class has the responsibility of performing all conversions between the WinRT data types and DirectX data types.

Constructing Path Geometries

In BasicFingerPaint, each stroke is a collection of connected short lines constructed from tracking a series of Pointer events. Typically, a finger-paint application will render these lines on a bitmap that can then be saved to a file. I decided not to do that. The files you save and load from BasicFingerPaint are collections of strokes, which are themselves collections of points.

How do you use Direct2D to render these strokes on the screen? If you look through the drawing methods defined by ID2D1DeviceContext (which are mostly methods defined by ID2D1RenderTarget), three candidates jump out: DrawLine, DrawGeometry and FillGeometry.

Figure 4 **Creating a Path Geometry from Points**

```
ComPtr<ID2D1PathGeometry>
  FingerPaintRenderer::CreatePolylinePathGeometry
    (std::vector<D2D1_POINT_2F> points)
{
  // Create the PathGeometry
  ComPtr<ID2D1PathGeometry> pathGeometry;
  HRESULT hresult = m_d2dFactory->CreatePathGeometry(&pathGeometry);

  // Get the GeometrySink of the PathGeometry
  ComPtr<ID2D1GeometrySink> geometrySink;
  hresult = pathGeometry->Open(&geometrySink);

  // Begin figure, add lines, end figure, and close
  geometrySink->BeginFigure(points.at(0), D2D1_FIGURE_BEGIN_HOLLOW);
  geometrySink->AddLines(points.data() + 1, points.size() - 1);
  geometrySink->EndFigure(D2D1_FIGURE_END_OPEN);
  hresult = geometrySink->Close();

  return pathGeometry;
}
```

DrawLine draws a single straight line between two points with a particular width, brush and style. It's reasonable to render a stroke with a series of DrawLine calls, but it's probably more efficient to consolidate the individual lines in a single polyline. For that, you need DrawGeometry.

In Direct2D, a geometry is basically a collection of points that define straight lines, Bezier curves and arcs. There's no concept of line width, color or style in a geometry. Although Direct2D supports several types of simple geometries (rectangle, rounded rectangle, ellipse), the most versatile geometry is represented by the ID2D1PathGeometry object.

A path geometry consists of one or more "figures." Each figure is a series of connected lines and curves. The individual components of the figure are known as "segments." A figure might be closed—that is, the last point might connect with the first point—but it need not be.

To render a geometry, you call DrawGeometry on the device context with a particular line width, brush and style. The FillGeometry method fills the interior of closed areas of the geometry with a brush.

Figure 5 **Accumulating Strokes in FingerPaintRenderer**

```
void FingerPaintRenderer::BeginStroke(unsigned int id, Point point,
                                      float width, Color color)
{
  // Save stroke information in StrokeInfo structure
  StrokeInfo strokeInfo;
  strokeInfo.Points.push_back(Point2F(point.X, point.Y));
  strokeInfo.Color = ColorF(color.R / 255.0f, color.G / 255.0f,
                            color.B / 255.0f, color.A / 255.0f);
  strokeInfo.Width = width;

  // Store in map with ID number
  strokesInProgress.insert(std::pair<unsigned int, StrokeInfo>(id, strokeInfo));
  this->IsRenderNeeded = true;
}

void FingerPaintRenderer::ContinueStroke(unsigned int id, Point point)
{
  // Never started a stroke, so skip
  if (strokesInProgress.count(id) == 0)
      return;

  // Get the StrokeInfo object for this finger
  StrokeInfo strokeInfo = strokesInProgress.at(id);
  D2D1_POINT_2F previousPoint = strokeInfo.Points.back();

  // Skip duplicate points
  if (point.X != previousPoint.x || point.Y != previousPoint.y)
  {
    strokeInfo.Points.push_back(Point2F(point.X, point.Y));
    strokeInfo.Geometry = nullptr;          // Because now invalid
    strokesInProgress[id] = strokeInfo;
    this->IsRenderNeeded = true;
  }
}

void FingerPaintRenderer::EndStroke(unsigned int id, Point point)
{
  if (strokesInProgress.count(id) == 0)
      return;

  // Get the StrokeInfo object for this finger
  StrokeInfo strokeInfo = strokesInProgress.at(id);

  // Add the final point and create final PathGeometry
  strokeInfo.Points.push_back(Point2F(point.X, point.Y));
  strokeInfo.Geometry = CreatePolylinePathGeometry(strokeInfo.Points);

  // Remove from map, save in vector
  strokesInProgress.erase(id);
  completedStrokes.push_back(strokeInfo);
  this->IsRenderNeeded = true;
}
```

To encapsulate a stroke, FingerPaintRenderer defines a private structure called StrokeInfo, as shown in **Figure 3**.

Figure 3 also shows two collections used for saving StrokeInfo objects: The completedStrokes collection is a vector collection, while strokesInProgress is a map collection using the pointer ID as a key.

> In Direct2D, a geometry is basically a collection of points that define straight lines, Bezier curves and arcs.

The Points member of the StrokeInfo structure accumulates all the points that make up a stroke. From these points, an ID2D1PathGeometry object can be constructed. **Figure 4** shows the method that performs this job. (For clarity, the listing doesn't show the code that checks for errant HRESULT values.)

An ID2D1PathGeometry object is a collection of figures and segments. To define the contents of a path geometry, you first call Open on the object to obtain an ID2D1GeometrySink. On this geometry sink, you call BeginFigure and EndFigure to delimit each figure, and between those calls, AddLines, AddArc, AddBezier and others to add segments to that figure. (The path geometries created by FingerPaintRenderer have only a single figure containing multiple straight line segments.) After calling Close on the geometry sink, the path geometry is ready to use but has become immutable. You can't reopen it or change anything in it.

For this reason, as your fingers move across the screen and the program accumulates points and shows strokes in progress, new path geometries must be continually built and old ones abandoned.

When should these new path geometries be created? Keep in mind that an application can receive PointerMoved events faster than the video refresh rate, so it doesn't make sense to create the path geometry in the PointerMoved handler. Instead, the program handles this event by just saving the new point, but not if it duplicates the previous point (which sometimes happens).

Figure 5 shows the three primary methods in FingerPaintRenderer involved in the accumulation of points that make up a stroke. A new StrokeInfo is added to the strokeInProgress collection during BeginStroke; it's updated during ContinueStroke, and transferred to the completedStrokes collection in EndStroke.

Figure 6 **The Rendering Loop in DirectXPage**

```
void DirectXPage::OnRendering(Object^ sender, Object^ args)
{
  if (m_renderer->IsRenderNeeded)
  {
    m_timer->Update();
    m_renderer->Update(m_timer->Total, m_timer->Delta);
    m_renderer->Render();
    m_renderer->Present();

    m_renderer->IsRenderNeeded = false;
  }
}
```

Notice that each of these methods sets IsRenderNeeded to true, indicating that the screen needs to be redrawn. This represents one of the structural changes I had to make to the project. In a newly created project based on the Direct2D (XAML) template, both DirectXPage and SimpleTextRenderer declare a private Boolean data member named m_renderNeeded. However, only in DirectXPage is the data member actually used. This isn't quite as it should be: Often the rendering code needs to determine when the screen must be redrawn. I replaced those two m_renderNeeded data members with a single public property in FingerPaintRenderer named IsRenderNeeded. The IsRenderNeeded property can be set from both DirectXPage and FingerPaintRenderer, but it's used only by DirectXPage.

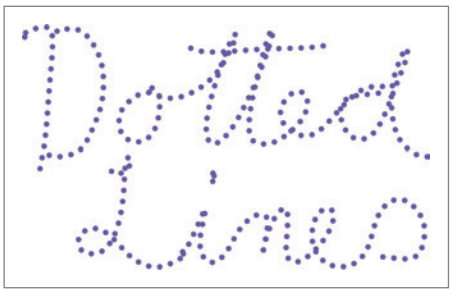


Figure 7 **Rendering a Path Geometry with a Dotted Line**

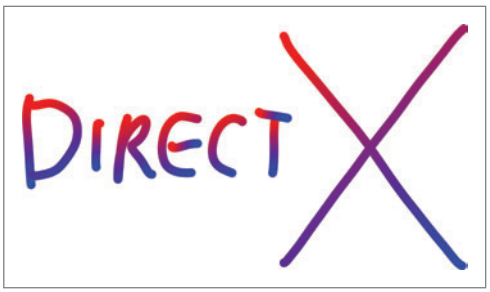


Figure 8 **The GradientFingerPaint Program**

The Rendering Loop

In the general case, a DirectX program can redraw its entire screen at the video refresh rate, which is often 60 frames per second or thereabouts. This facility gives the program maximum flexibility in displaying graphics involving animation or transparency. Rather than figuring out what part of the screen needs to be updated and how to avoid messing up existing graphics, the entire screen is simply redrawn.

In a program such as BasicFingerPaint, the screen only needs to be redrawn when something changes, which is indicated by a true setting of the IsRenderNeeded property. In addition, redrawing might conceivably be limited to certain areas of the screen, but this is not quite so easy with an application created from the Direct2D (XAML) template.

To refresh the screen, DirectXPage uses the handy CompositionTarget::Rendering event, which is fired in synchronization with the hardware video refresh. In a DirectX program, the handler for this event is sometimes known as the rendering loop, and is shown in **Figure 6**.

The Update method is defined by the renderer. This is where visual objects are prepared for rendering, particularly if they require timing information provided by a timer class created by the project template. FingerPaintRenderer uses the Update method to create path geometries from point collections, if necessary. The Render method is declared by DirectXBase but defined by FingerPaintRenderer, and is responsible for rendering all the graphics. The method named Present—it's a verb, not a noun—is defined by DirectXBase, and transfers the composited visuals to the video hardware.

The Render method begins by calling BeginDraw on the program's ID3D11DeviceContext object and concludes by calling EndDraw. In between, it can call drawing functions. The rendering of each stroke during the Render method is simply:

```
m_solidColorBrush->SetColor(strokeInfo.Color);
m_d2dContext->DrawGeometry(strokeInfo.Geometry.Get(),
                           m_solidColorBrush.Get(),
                           strokeInfo.Width,
                           m_strokeStyle.Get());
```

The m_solidColorBrush and m_strokeStyle objects are data members.

What's the Next Step?

As the name implies, BasicFingerPaint is a very simple application. Because it doesn't render strokes to a bitmap, an eager and persistent finger painter could cause the program to generate and render thousands of geometries. At some point, screen refresh might suffer.

However, because the program maintains discrete geometries rather than mixing everything together on a bitmap, the program could allow individual strokes to be later deleted or edited, perhaps by changing the color or width, or even moved to a different location on the screen.

Because each stroke is a single path geometry, applying different styling is fairly easy. For example, try changing one line in the CreateDeviceIndependentResources method in FingerPaintRenderer:

```
strokeStyleProps.dashStyle = D2D1_DASH_STYLE_DOT;
```

Now the program draws dotted lines rather than solid lines, with the result shown in **Figure 7**. This technique only works because each stroke is a single geometry; it wouldn't work if the individual segments comprising the strokes were all separate lines.

Another possible enhancement is a gradient brush. The GradientFingerPaint program is very similar to BasicFingerPaint except that it has two combo boxes for color, and uses a linear gradient brush to render the path geometry. The result is shown in **Figure 8**.

Although each stroke has its own linear gradient brush, the start point of the gradient is always set to the upper-left corner of the stroke bounds, and the end point to the bottom-right corner. As you draw a stroke with a finger, you can often see the gradient changing as the stroke gets longer. But depending how the stroke is drawn, sometimes the gradient goes along the length of the stroke, and sometimes you barely see a gradient at all, as is obvious with the two strokes of the X in **Figure 8**.

Wouldn't it be better if you could define a gradient that extended along the full length of the stroke, regardless of the stroke's shape or orientation? Or how about a gradient that's always perpendicular to the stroke, regardless of how the stroke twists and turns?

As they ask in the science fiction movies: How is such a thing even possible? ■

Charles Petzold *is a longtime contributor to* MSDN Magazine *and the author of "Programming Windows, 6th Edition," (Microsoft Press, 2012) a book about writing applications for Windows 8. His Web site is charlespetzold.com.*

Thanks to the following technical experts for reviewing this article: *James McNellis (Microsoft)*

The Decade of UX

A named decade doesn't really begin until about three years in. What we think of as the Sixties (the "Free Love Decade") didn't start until JFK's assassination in 1963, the Seventies (the "Me Decade") until the 1973 energy crisis, the Eighties (the "Reagan Decade") until the 1983 release of "Return of the Jedi," and so on.

We're now three years into the 2010s, so it's time to name them. UX has become the defining factor in the success or failure of software. Therefore, I hereby declare this the "Decade of User Experience," or DoUX (which, appropriately, means "sweet" in French).

You might have gotten away with a crappy UX 20 years ago, but customers today demand much more. Our standard of care is continually rising (one word: iPhone). A computer that users can't figure out how to use, or that requires expensive training to use, or that makes expensive mistakes because the program misleads users, is a very expensive paperweight.

> Not only do companies need to increase their UX efforts, but every developer now needs to know UX, even if it's not his primary job, just as every soldier needs to know battlefield first aid.

Not only do companies need to increase their UX efforts, but every developer now needs to know UX, even if it's not his primary job, just as every soldier needs to know battlefield first aid. Understanding UX is as essential today as understanding functions was in the 1980s, objects in the 1990s (the "Generation X Decade") and Web services in the 2000s (the "Millennial Decade"). For a university to bestow a computer science degree without a class in UX constitutes malpractice.

Many developers or architects think they don't need to understand UX. Here's why they say that, and why they're wrong.

Our Projects Are Low-Level, So UX Doesn't Matter

Nonsense. Every project you work on has some connection to a human user somewhere. Even a programmatic Web service needs error reporting, installation and configuration, status and performance monitoring dashboards, and so on. If your project only has small amounts of UX, that's all the more reason it needs to work well.

Marketing Decides Our UX

You're wise to have a good relationship with your marketing department. They certainly feel pain points from the customer and bring them back to you.

At the same time, marketeers are not interaction designers. They might give you the first clue as to what would make the customer happier and more productive—"Customers complain that they're losing work when our app crashes"—but it's up to you to take it from there. How often does it happen? How do you detect and measure it? To fix it: Auto-save? Rollback? How often? Configurable? Imagine asking these questions of your marketing department, and tell me if they're really driving the UX. They're not; you are, even if they sound the initial alarm.

You should also be talking to your tech support department. They feel your customers' pain far more immediately and brutally than the glad-handing marketeers.

We Have a UX Group That Does That Stuff

Some large companies have a UX design group that approves every user interaction. If you have one of these, you've already found that their time is in tight supply, as it is for other ultra-specialists such as ophthalmologists. You can get a brief appointment in six months if you beg. They can't follow up or help you iterate. You need to understand what they tell you in your far-too-limited interactions and implement those principles day-to-day to make the best use of the thimblefuls of their wisdom that you actually get.

Also, their time is (quite rationally) dedicated to your company's primary, outward-facing programs. They don't have time for your internal or second-tier apps. Ironically, because your company values good UX, the apps you work on are held to a higher standard. But your bosses don't give you the skill set or resources to meet these demands, now do they?

UX Is for the Beret-Heads

Also known as *graphical* designers. More accurately, I call them *decorators*. I've written about them before (msdn.microsoft.com/magazine/hh394140). The UX game is almost always lost before it reaches them.

Every developer now needs to understand UX. Take it in college, take one of my classes (in Boston or London this October, see idesign.net), take it in continuing education, but take it. And for at least the next 10 years, make it sweet. ■

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.*

HTML5 & JavaScript | WPF & Silverlight | Windows Forms | Windows Phone | Windows 8

HTML5

Based on industry standards & built with HTML5, jQuery, & CSS3

Adaptive widgets adapt to jQuery Mobile or jQuery UI environments

Effortless application-wide theming with built-in professionally-designed themes

Full cross-browser compatibility

Touch

Windows Store & Windows Phone components designed with a touch-first UX in mind

Support for selecting, dragging, & multi-touch gestures

Supports for Windows Phone Modern UI design & interaction guidelines specified by Microsoft

Windows Store compliant to speed the app submission process

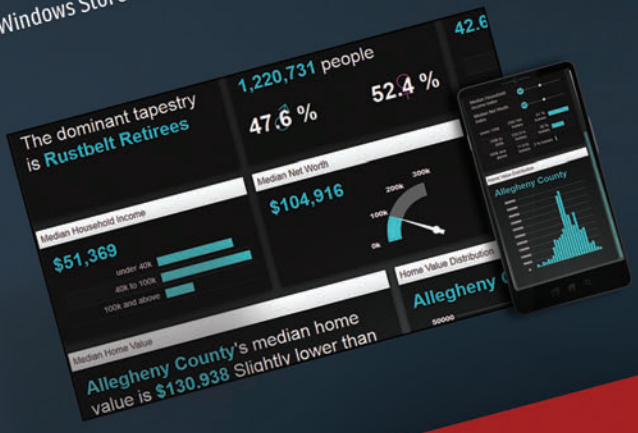
Data Visualization

Cross-platform charts, grids, maps, gauges, & more

Reporting & document-generation with pixel-perfect reporting, PDF, & print-preview controls

Award-winning grids with filtering, hierarchical data-binding, grouping, printing, & exporting to Microsoft Excel

Bring data to life with dynamic dashboards



.NET Tools for the Professional Developer

STUDIO ENTERPRISE

ComponentOne
a division of GrapeCity®

Download your free trial @
www.componentone.com