THE MICROSOFT JOURNAL FOR DEVELOPERS

FEBRUARY 2013 VOL 28 NO 2

msdn magazine

JavaScript API for Office.....20

COLUMNS

Microsoft

Compatible with
Microsoft® Visual Studio® 2012

BRILLIANT UX

At Your Fingertips

Download your free trial
infragistics.com/**EXPERIENCE**

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

FEBRUARY 2013 VOLUME 28 NUMBER 2

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.

MICHAEL DESMOND

Inside Modern Apps

This month *MSDN Magazine* welcomes Rachel Appel and her Modern Apps column. Readers of the magazine and Web site might be familiar with Appel's work. She wrote the popular Web Dev Report column on the *MSDN Magazine* Web site up until September of last year. And she appeared in our Windows 8 Special Edition, published in October, with a look at the unique aspects of the Windows Store application lifecycle.

This month, Modern Apps debuts with a quick rundown of what you need to get started with Windows Store app development, and explores the new features and capabilities that enable developers to create powerful applications for Windows 8 and the Windows Runtime. Going forward, you can expect Appel to dig deeper into the Windows Store app dev experience. In the March issue, look for the column to explore the important topic of data access and storage for Windows Store apps.

Appel's column arrives a few short months after Bruno Terkaly and Ricardo Villalobos came on board as authors of the new Windows Azure Insider column. And just last month Charles Petzold re-branded his column as DirectX Factor, reflecting his focus on the powerful DirectX development infrastructure in the Windows Runtime.

The changes reflect the significant progress we've seen from Microsoft in updating its developer infrastructure. Windows 8 and the Windows Runtime present a compelling target for an incredibly broad range of software developers, and the new columns from Appel and Petzold aim to empower those people—from business devs working with C# to Web programmers working with JavaScript to native coders working with C++. Similarly, major updates in developer tooling for Windows Azure have changed the game in the cloud development space, and Terkaly and Villalobos are here to help guide you through it.

'It's New and Shiny'

As for Rachel Appel, she's been writing for *MSDN Magazine* and its Web site for more than a year now, and was a Microsoft technical evangelist for years before that. A veteran programmer, Appel has been in the software development racket since the late 1980s, first writing back-end enterprise applications in COBOL. She struck out on her own as an independent consultant, trainer and mentor, earning MVP recognition from Microsoft, before joining Microsoft as a technical evangelist. Today, she spends a great deal of her time giving talks at conferences, working with customers and blogging about development issues.

When I asked Appel how a former COBOL programmer ends up helping lead the charge on Windows Store app development, she couldn't resist a joke—"The short answer? It's new and shiny"—before providing a more serious response.

"I enjoy learning about new technologies and the latest in software development," Appel says. "As a tech evangelist, being out in the public allows me to see and work with all kinds of awesome ideas and code."

She goes on to praise some of the built-in features of Windows 8, including Search and Share contracts, the rich sensor platforms and the device APIs. "You can access it all through open, standard HTML5 and ES5 [ECMAScript 5] if you want, or you can use C#/Visual Basic/C++ and XAML, as all languages have parity."

In her community engagements, Appel says she sometimes fields questions about the Windows Runtime and its relationship to the Microsoft .NET Framework. Her message: The .NET Framework is not going anywhere.

"Windows Runtime is not a .NET replacement. Rather, many of the WinRT APIs are wrappers around classic .NET or Win32 libraries, so those underlying frameworks are still available," Appel says. "Yes, you can write WinRT apps in HTML5. And, yes, it's real HTML5."

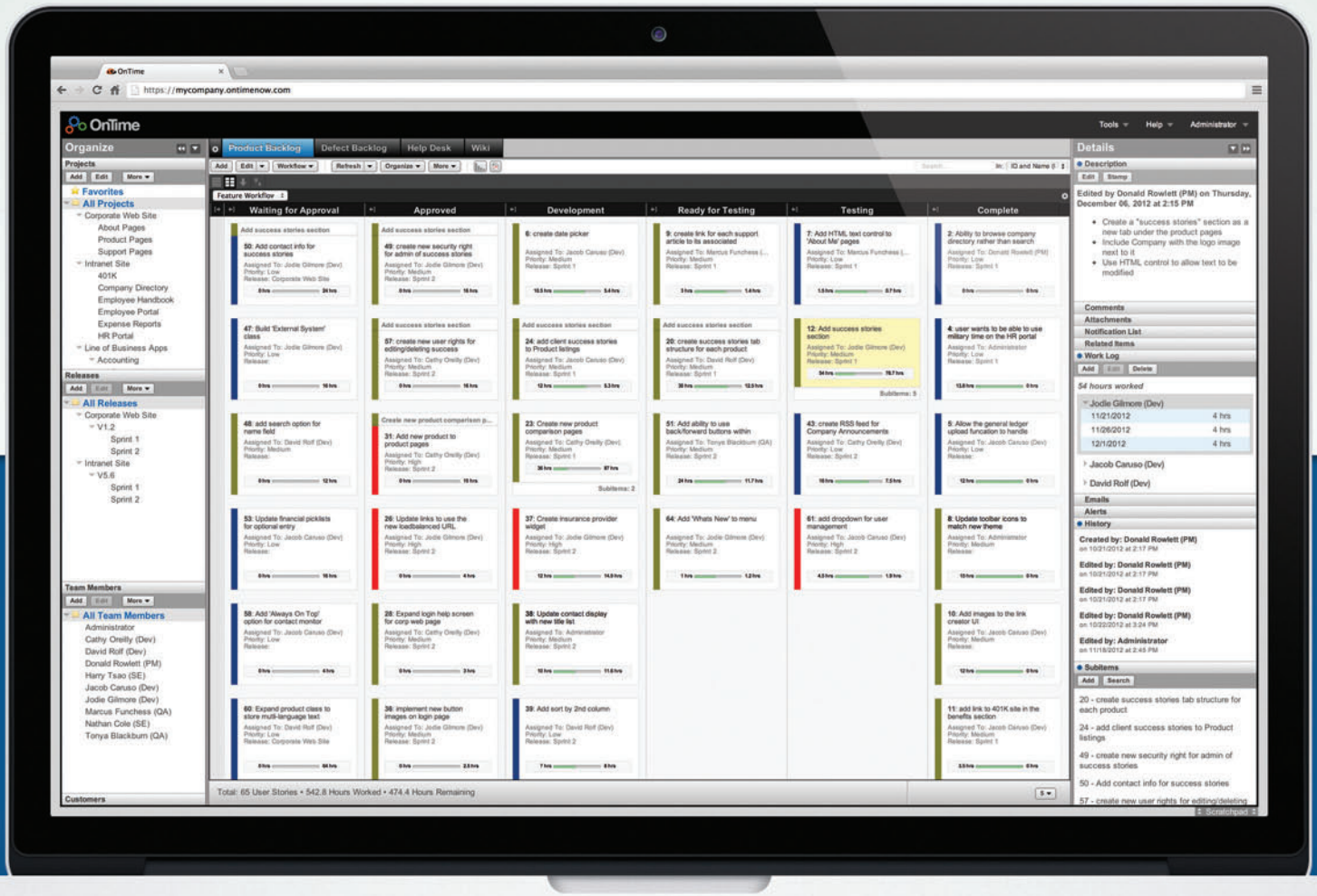
Appel urges readers of her Modern Apps column to also check out the GenerationApp site (`bit.ly/W8GenAppDev`), which features articles and tutorials aimed at developers building Windows 8 and Windows Phone 8 apps.

There's a whole lot going on with the emergence of Windows 8 and the Windows Runtime, and our new columns are designed to address that activity. Is there something you want to see Appel cover in her Modern Apps column? E-mail her at rachel@rachelappel.com.

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

OnTime Scrum

Agile project management & bug tracking software



Introducing **OnTime 13.**

Ditch your sticky notes. The Card View is now here.

The **OnTime Card View** is the ideal planning board tool for Kanban or Scrum teams. It adds a whole new dimension to user story management, bug tracking and workflow automation.

Learn more about Card View and the many other features of OnTime Scrum that your dev team will love.

OnTimeNow.com/MSDN

$834 per month for up to 10 users billed annually

special small-team pricing

$584 per user per month billed annually

for teams of 11+ users

axosoft

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

DINO ESPOSITO

Essential Facebook Programming: The JavaScript SDK

I'll be honest: I don't know exactly what a social strategy might look like and I have no more than a faint idea about the tricks that social experts can play to make your content more popular. Likewise, I never paid much attention to search engine optimization (SEO) in Web sites, but I do believe that social optimization is today much more relevant and rewarding than SEO ever was.

In the past two installments of this column, I discussed how to authenticate users of a Web or Windows application using Facebook and how to post to the social network on behalf of a consenting user. (Those two columns can be found at msdn.microsoft.com/magazine/jj863128 and msdn.microsoft.com/magazine/jj883950.)

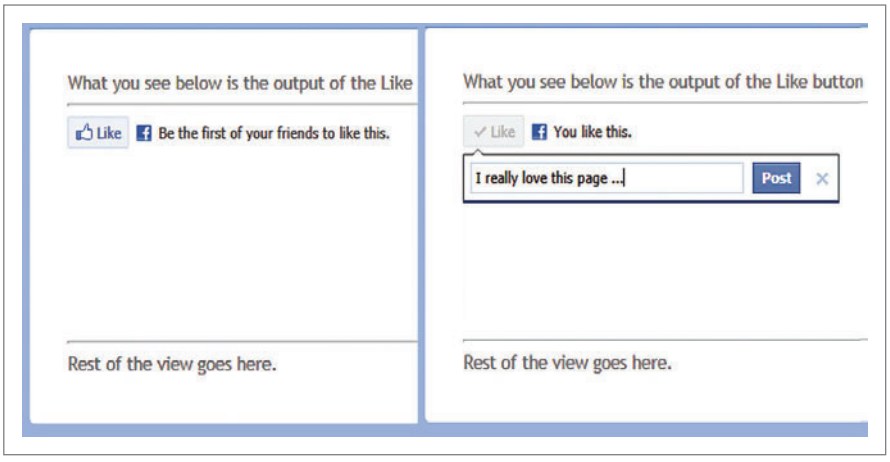


Figure 1 **The Standard Interface of the Like Button**

As far as Facebook is concerned, there's a lot more you can do to add a "social layer" to a Web site. A social layer results from the combination of three main ingredients: identity, community and interaction. In those previous columns, I addressed identity and a bit of interaction. Here, I'm going to explore the principal tools you can leverage to add a true social layer onto your existing Web site. According to Facebook, these tools are collectively known as social plug-ins. A social plug-in is a powerful mix of HTML markup, CSS and JavaScript code. It's your responsibility as a Web developer to fuse these elements into the UX.

The Ubiquitous Like Button

The primary purpose of social plug-ins is to create a bridge between the content of the Web site visited by a user and the user's Facebook page. By hosting one or more social plug-ins, the Web site enables users to share specific content with friends. The most immediate way for a Web site to let users share content via Facebook is the Like button.

By simply clicking the Like button, a user can let friends know that she likes something at a given URL. Hosting the Like button on a page couldn't be simpler; fusing the button to the existing UI may require a few extra steps.

Code download available at archive.msdn.microsoft.com/mag201302CuttingEdge.

There are a few different ways to embed a Like button. The simplest and most direct way is using an iframe element:

```
<iframe src="http://www.facebook.com/plugins/like.php?href=your-site"
  scrolling="no" frameborder="0"
  style="border:solid 1px #000; width:450px; height:80px"></iframe>
```

The href query string parameter refers to the URL you want to like. The URL must be expressed in a fully qualified manner, something like http://www.contoso.com/cool.

Most of the markup is aimed at styling the iframe. Usually, you don't want the iframe to scroll or be framed. You also want it to take up an appropriate area. The preceding markup produces the output in **Figure 1**.

If the height of the iframe is too small (less than 25 pixels or so), you lose the panel containing the button to post an additional comment. If you're using the Like button within a horizontal menu bar, then the height is a critical issue. Otherwise, giving users a chance to also post their own comment augments the penetration of the feature.

There are several parameters you can leverage to customize the look, feel and functionality of the button. **Figure 2** lists the options available. All will be assigned through an additional query string parameter.

While **Figure 1** shows the standard layout, **Figure 3** shows the button_count and box_count layouts.

Another parameter you should look into is the ref parameter. It helps you track use of the Like button in your Web site. By giving each Like button (even in different pages of the site) a different ref value—a plain alphanumeric string—you can easily track referrers in the server log that can be traced back to that specific Like button. In particular, for any click back from Facebook to your site, you'll

receive a referrer URL with two extra parameters. The fb_ref query string parameter is just your original ref string; the fb_source query string parameter is a string that gives you information about the context from within Facebook where the click originated.

Localizing the Like Button

Even though the Like button can be considered universal, there still might be situations in which you want to translate the Like plug-in to a given language. As shown in **Figure 2**, all you need to do is add the locale parameter to the query string and set it to a custom string that indicates the desired language and culture.

To make things a bit trickier, you can't express culture using the canonical xx-XX format where xx indicates the language and XX the culture. In the Microsoft .NET Framework, you get this string from the following expression:

```
var name = Thread.CurrentThread.CurrentUICulture.Name;
```

For this string to be usable with Facebook, you need to replace the dash with an underscore. Also note that the sole language token isn't sufficient in itself and the whole locale setting will be ignored. This point leads me to another interesting consideration: What's the default behavior of the Like button as far as the language is concerned?

Facebook, as well as Twitter, defaults to the language the user has chosen as the primary language in her profile. If the user isn't currently logged in, then the UI is based on the language settings detected on the browser.

Introducing the JavaScript SDK

In general, you can configure Facebook's plug-ins in a few different ways: using a plain URL from a custom hyperlink; using an iframe (as shown in this article); using HTML5 markup; and using the eXtended Facebook Markup Language (XFBML). HTML5 and XFBML are equivalent in many ways; HTML5 is just a more-recent syntax supported for completeness. Both HTML5 and XFBML require the use of the Facebook JavaScript SDK.

Most sophisticated social plug-ins are only available through HTML5 and XFBML markup. This is the case for the Send button for sending content directly to friends and the Comments box to see the comments on a given post. Other plug-ins such as Like, the Like box and the Activity feed (the small list of notifications from all friends you usually have on the top-right corner of your page) also can be quickly configured and embedded via an iframe.

Figure 2 **Customizing Look, Feel and Functionality**

Parameter	Description
action	Indicates the text of the button and the subsequent action to be taken. It can be Like or Recommend. Default is Like.
colorscheme	Styles the button differently. It can be light or dark. Default is light.
font	Sets the desired font for the plug-in. Possible options include Arial, Tahoma, Trebuchet MS and a few others.
layout	Changes the layout of the button. Possible values are standard (as in Figure 1), button_count and box_count (as in Figure 3).
locale	Indicates the language of the UI. It must be a string in the format xx_XX. Note the use of the underscore to separate the two parts of a culture name.
show_faces	Boolean flag to indicate whether you want the picture of the user rendered once a user has liked the content.
width	Indicates the width of the plug-in. The minimum width also depends on the layout used.

Plug-ins not implemented through iframes or direct URLs require the use of the Facebook JavaScript SDK. As you can imagine, the SDK is a client front-end for a number of Facebook endpoints for you to perform tasks such as authentication, posting and retrieving comments, likes, and other actions.

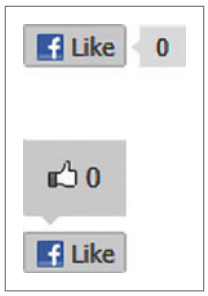


Figure 3 **Additional Layout Formats (Button Count and Box Count) for the Like Button**

In order to use the JavaScript SDK, even before you download it, you must have an app ID. I thoroughly discussed this point in previous columns, but in a nutshell, for any interaction with the Facebook site that goes beyond basic operations such as Like, you need to register an app and get a unique ID. The app plays the role of the connector between your client (for example, a Web site) and the Facebook back end. The Facebook App Dashboard for registering an app is available at bit.ly/mly4xs.

The second step consists of adding some code to your Web pages that downloads the SDK. The URL to invoke is: /connect.facebook.net/xx_XX/all.js.

> Most sophisticated social plug-ins are only available through HTML5 and XFBML markup.

The xx_XX in the URL is a placeholder for the desired locale. An option is linking this URL from within a static script tag. Because the size of the file is far more than 180KB, it might not be a good idea to download it synchronously through a static script tag. The file gets cached soon and most of the successive requests for it will receive an HTTP 304 "not modified" status code; however, Facebook recommends you download the file asynchronously. This is a pattern common to all script blocks required for social interactions—for example, it works the same for Twitter plug-ins. Here's the code you need:

```
<script type="text/javascript">
  (function (d, s, id) {
    var js, fjs = d.getElementsByTagName(s)[0];
    if (d.getElementById(id)) return;
    js = d.createElement(s);
    js.id = id;
    js.async = true;
    js.src = "//connect.facebook.net/en_US/all.js#xfbml=1&appId=xxx";
    fjs.parentNode.insertBefore(js, fjs);
  } (document, 'script', 'facebook-jssdk'));
</script>
```

The code is defined as a JavaScript immediate function and runs as soon as it's found. Note that you place this code preferably right after the opening body tag and you should fill the appId parameter with your app ID. (Note that many developers advocate placing scripts at the bottom of the body section in order to prevent any blocking of the rendering of the page.)

Telerik DevCraft

The all-in-one toolset for professional developers targeting Microsoft platforms.

- Create web, mobile and desktop applications that impress
- Cover any .NET platform and any device
- Code faster and smarter without cutting corners

www.telerik.com/all-in-one

telerik

At this point, you're ready to use HTML5 and XFBML tags to integrate Facebook plug-ins in your pages. Let's start with a look at the Login button.

The Login Button

The SDK contains a method on the root FB object through which you can programmatically trigger the login process. The simplest option consists of adding your own link or button and binding its click handler to the following code:

```
FB.login(function(response) {
  if (response.authResponse) {
    // Display some wait message
    // ...
    FB.api('/me', function(response) {
      $("#username").html('Welcome, ' + response.name + '.');
    });
  }
});
```

This code is far simpler than any other analogous code in the past two columns. In this way, authenticating users against Facebook is a breeze (see **Figure 4**).

The Login plug-in can make the login process even easier. In addition to linking the SDK, all you need is the following HTML5 markup (or analogous XFBML markup as demonstrated on the Facebook site):

```
<div class="fb-login-button"
  data-show-faces="true"
  data-width="200"
  data-max-rows="1"></div>
```

The data-* attributes let you configure the appearance and behavior of the button. The blue button in the page of **Figure 4** gives an idea of the standard look and feel. In particular, the data-show-faces attribute enables you to display the pictures of users (and some of their friends) that used your app to connect to Facebook. The data-max-rows attribute determines the number of rows (given the width of the plug-in) to be filled with faces.

It should also be noted that if data-show-faces is on and the user is already logged in, then no login button is shown. The user can't log out from Facebook through your app. If data-show-faces is false, then the login button stays visible all the time and it doesn't react if clicked.

As a Web developer, you should see the profound difference between using the JavaScript SDK or the Login plug-in and server-side code and the Facebook C# SDK. If you work the client side, then you're essentially targeting Facebook, and login is probably the necessary first step in order to do more specific work. Facebook is the goal here.

> Provided that it's not already so, authentication via social networks will become a must-have feature for all sites needing authentication.

C# code is preferable if you're using Facebook as just one way of authenticating users and still need to handle the authentication cookie of ASP.NET and the standard membership system. Facebook is the means here.

Coming Up

Provided that it's not already so, authentication via social networks will become a must-have feature for all sites needing authentication. This means the C# SDK probably remains the most flexible approach. In this regard, the new social classes in ASP.NET MVC 4 are just icing on the cake. In terms of Web site development, I likewise see no reason for not populating page layouts with social buttons and plug-ins to tweet or post immediate content. Next time, I'll be back with Facebook programming covering more advanced social plug-ins such as Comments and the Like box. Meanwhile, for more information on Facebook social plug-ins, you can have a look at bit.ly/fAU7Xe. ■

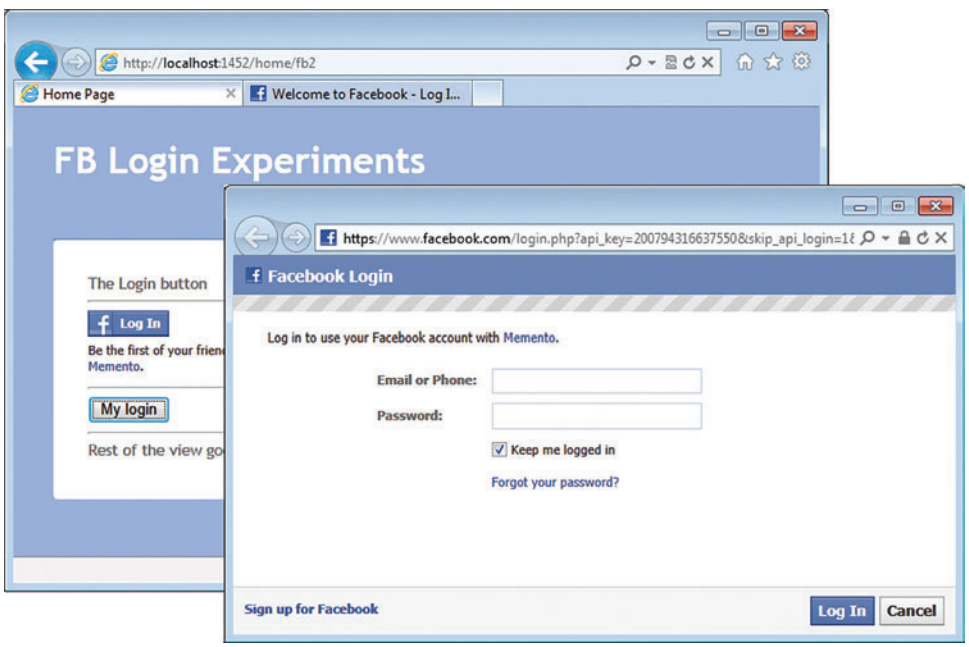


Figure 4 **Authenticating Users via JavaScript**

Dino Esposito *is the author of "Architecting Mobile Solutions for the Enterprise" (Microsoft Press, 2012) and "Programming ASP.NET MVC 3" (Microsoft Press, 2011), and coauthor of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2008). Based in Italy, Esposito is a frequent speaker at industry events worldwide. Follow him on Twitter at twitter.com/despos.*

Thanks to the following technical experts for reviewing this article: *Christopher Bennage and Scott Densmore*

Globalize Your Business

Melissa Data can help you globalize your applications as you expand operations to other countries or reach new customers in emerging markets. As a world leading data quality vendor, we offer solutions to verify, correct and standardize addresses in over 240 countries. Eliminate returns, cut postage expenses, prevent fraud and keep your customers happy by verifying their address before you send a package.

- Reduce address correction fees – save up to $10 per package
- Efficiently validate and correct addresses every time you ship
- Maintain high customer satisfaction

Accurate data. Delivered.

- ✔ **Address Verification**
- ✔ **ID Verification**
- ✔ **Email Verfication**
- ✔ **GeoCoding**
- ✔ **IP Location**
- ✔ **Name Parsing**
- ✔ **Phone Verification**
- ✔ **Record Matching**

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

MELISSA DATA®
Your Partner in Data Quality

C# | VB.NET | SQL Server 2012-Data Quality Services | SQL Server® SSIS

KENNY KERR

Creating Desktop Apps with Visual C++ 2012

With all the hype over Windows 8 and what are now known as Windows Store apps, I've received some questions about the relevance of desktop apps and whether Standard C++ is still a viable choice going forward. These questions are sometimes hard to answer, but what I can tell you is that the Visual C++ 2012 compiler is more committed than ever to Standard C++ and it remains the best toolchain, in my humble opinion, for building great desktop apps for Windows whether you're targeting Windows 7, Windows 8 or even Windows XP.

A follow-up question I inevitably receive is how best to approach desktop app development on Windows and where to begin. Well, in this month's column, I'm going to explore the fundamentals of creating desktop apps with Visual C++. When I was first introduced to Windows programming by Jeff Prosise (bit.ly/WmoRuR), Microsoft Foundation Classes (MFC) was a promising new way to build apps. While MFC is still available, it really is showing its age, and a need for modern and flexible alternatives has driven programmers to search for new approaches. This issue has been compounded by a shift away from USER and GDI (msdn.com/library/ms724515) resources and toward Direct3D as the primary foundation by which content is rendered on the screen.

For years I've been promoting the Active Template Library (ATL) and its extension, the Windows Template Library (WTL), as great choices for building apps. However, even these libraries are now showing signs of aging. With the shift away from USER and GDI resources, there's even less reason to use them. So where to begin? With the Windows API, of course. I'll show you that creating a desktop window without any library at all isn't actually as daunting as it might seem at first. I'll then show you how you can give it a bit more of a C++ flavor, if you so desire, with a little help from ATL and WTL. ATL and WTL make a lot more sense once you have a good idea of how it all works behind the templates and macros.

The Windows API

The trouble with using the Windows API to create a desktop window is that there are myriad ways you could go about writing it—far too many choices, really. Still, there's a straightforward way to create a window, and it starts with the master include file for Windows:

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

You can then define the standard entry point for apps:

```
int __stdcall wWinMain(HINSTANCE module, HINSTANCE, PWSTR, int)
```

If you're writing a console app, then you can just continue to use the standard C++ main entry point function, but I'll assume that you don't want a console box popping up every time your app starts. The wWinMain function is steeped in history. The __stdcall calling convention clarifies matters on the confusing x86 architecture, which provides a handful of calling conventions. If you're targeting x64 or ARM, then it doesn't matter because the Visual C++ compiler only implements a single calling convention on those architectures—but it doesn't hurt, either.

> I'll show you that creating a desktop window without any library at all isn't actually as daunting as it might seem at first.

The two HINSTANCE parameters are particularly shrouded in history. In the 16-bit days of Windows, the second HINSTANCE was the handle to any previous instance of the app. This allowed an app to communicate with any previous instance of itself or even to switch back to the previous instance if the user had accidentally started it again. Today, this second parameter is always a nullptr. You may also have noticed that I named the first parameter "module" rather than "instance." Again, in 16-bit Windows, instances and modules were two separate things. All apps would share the module containing code segments but would be given unique instances containing the data segments. The current and previous HINSTANCE parameters should now make more sense. 32-bit Windows introduced separate address spaces and along with that the necessity for each process to map its own instance/module, now one and the same. Today, this is just the base address of the executable. The Visual C++ linker actually exposes this address through a pseudo variable, which you can access by declaring it as follows:

```
extern "C" IMAGE_DOS_HEADER __ImageBase;
```

The address of __ImageBase will be the same value as the HINSTANCE parameter. This is in fact the way that the C Run-Time Library (CRT) gets the address of the module to pass to your wWinMain function in the first place. It's a convenient shortcut if you don't want to pass this wWinMain parameter around your app. Keep in mind, though, that this variable points to the current module whether it's a DLL or an executable and is thus useful for loading module-specific resources unambiguously.

Aspose.Total just got BIGGER

Aspose.Diagram

Working with Visio files? Easily create, modify and convert diagrams in your applications.

Supported Files

VSD VTX
VSS VDW
VST VDX
VSX

NEW

Aspose.OCR

Extract text from images. Supports popular fonts and styles. Scan a whole image or part of an image.

Supported Files

BMP
TIFF

NEW

Aspose.Imaging

Add advanced drawing features to your applications, plus support for PSD files.

Supported Files

PSD BMP
TIFF PNG
JPEG
GIF

NEW

Already own Aspose.Total for .NET?

These are yours for FREE!

Free Evaluations at www.aspose.com

EU Sales: +44 (0) 141 416 1112
sales.europe@aspose.com
AU Sales: +61 2 8003 5926
sales.asiapacific@aspose.com

US Sales: 1.888.277.6734
sales@aspose.com

ASPOSE
Your File Format Experts

The next parameter provides any command-line arguments, and the last parameter is a value that should be passed to the ShowWindow function for the app's main window, assuming you're initially calling ShowWindow. The irony is that it will almost always be ignored. This goes back to the way in which an app is launched via CreateProcess and friends to allow a shortcut—or some other app—to define whether an app's main window is initially minimized, maximized or shown normally.

Inside the wWinMain function, the app needs to register a window class. The window class is described by a WNDCLASS structure and registered with the RegisterClass function. This registration is stored in a table using a pair made up of the module pointer and class name, allowing the CreateWindow function to look up the class information when it's time to create the window:

```
WNDCLASS wc = {};
wc.hCursor = LoadCursor(nullptr, IDC_ARROW);
wc.hInstance = module;
wc.lpszClassName = L"window";

wc.lpfnWndProc = []
 (HWND window, UINT message, WPARAM wparam, LPARAM lparam) -> LRESULT
{
  ...
};

VERIFY(RegisterClass(&wc));
```

To keep the examples brief, I'll just use the common VERIFY macro as a placeholder to indicate where you'll need to add some error handling to manage any failures reported by the various API functions. Just consider these as placeholders for your preferred error-handling policy.

Once the window appears, it's important that your app starts dispatching messages as soon as possible—otherwise your app will appear unresponsive.

The earlier code is the minimum that's required to describe a standard window. The WNDCLASS structure is initialized with an empty pair of curly brackets. This ensures that all the structure's members are initialized to zero or nullptr. The only members that must be set are hCursor to indicate which mouse pointer, or cursor, to use when the mouse is over the window; hInstance and lpszClassName to identify the window class within the process; and lpfnWndProc to point to the window procedure that will process messages sent to the window. In this case, I'm using a lambda expression to keep everything inline, so to speak. I'll get back to the window procedure in a moment. The next step is to create the window:

```
VERIFY(CreateWindow(wc.lpszClassName, L"Title",
  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
  CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
  nullptr, nullptr, module, nullptr));
```

The CreateWindow function expects quite a few parameters, but most of them are just defaults. The first and second-to-last parameters, as I mentioned, together represent the key that the RegisterClass function creates to let CreateWindow find the window class information. The second parameter indicates the text that will be displayed in the window's title bar. The third indicates the window's style. The WS_OVERLAPPEDWINDOW constant is a commonly used style describing a regular top-level window with a title bar with buttons, resizable borders and so on. Combining this with the WS_VISIBLE constant instructs CreateWindow to go ahead and show the window. If you omit WS_VISIBLE, then you'll need to call the ShowWindow function before your window will make its debut on the desktop.

The next four parameters indicate the window's initial position and size, and the CW_USEDEFAULT constant used in each case just tells Windows to choose appropriate defaults. The next two parameters provide the handle to the window's parent window and menu, respectively (and neither are needed). The final parameter provides the option of passing a pointer-sized value to the window procedure during creation. If all goes well, a window appears on the desktop and a window handle is returned. If things go south, then nullptr is returned instead and the GetLastError function may be called to find out why. With all the talk about the hardships of using the Windows API, it turns out that creating a window is actually quite simple and boils down to this:

```
WNDCLASS wc = { ... };
RegisterClass(&wc);
CreateWindow( ... );
```

Once the window appears, it's important that your app starts dispatching messages as soon as possible—otherwise your app will appear unresponsive. Windows is fundamentally an event-driven, message-based OS. This is particularly true of the desktop. While Windows creates and manages the queue of messages, it's the app's responsibility to dequeue and dispatch them, because messages are sent to a window's thread rather than directly to the window. This provides a great deal of flexibility, but a simple message loop need not be complicated, as shown here:

```
MSG message;
BOOL result;

while (result = GetMessage(&message, 0, 0, 0))
{
  if (-1 != result)
  {
    DispatchMessage(&message);
  }
}
```

Perhaps not surprisingly, this seemingly simple message loop is often implemented incorrectly. This stems from the fact that the GetMessage function is prototyped to return a BOOL value, but in fact, this is really just an int. GetMessage dequeues, or retrieves, a message from the calling thread's message queue. This may be for any window or no window at all, but in our case, the thread is only pumping messages for a single window. If the WM_QUIT message is dequeued, then GetMessage will return zero, indicating that the window has disappeared and is done processing messages and that the app should terminate. If something goes terribly wrong, then GetMessage might return -1 and you can again call GetLastError to get more information. Otherwise, any nonzero return value from GetMessage indicates that a message was dequeued and is ready to be dispatched to the window. Naturally, this is the purpose of the DispatchMessage function. Of course, there are many variants

Figure 1 **A Static Table of Message Handlers**

```
static message_handler s_handlers[] =
{
  {
    WM_PAINT, [] (HWND window, WPARAM, LPARAM) -> LRESULT
    {
      PAINTSTRUCT ps;
      VERIFY(BeginPaint(window, &ps));

      // Dress up some pixels here!

      EndPaint(window, &ps);
      return 0;
    }
  },
  {
    WM_DESTROY, [] (HWND, WPARAM, LPARAM) -> LRESULT
    {
      PostQuitMessage(0);
      return 0;
    }
  }
};
```

to the message loop, and having the ability to construct your own affords you many choices for how your app will behave, what input it will accept and how it will be translated. Apart from the MSG pointer, the remaining parameters to GetMessage can be used to optionally filter messages.

The window procedure will start receiving messages before the CreateWindow function even returns, so it had better be ready and waiting. But what does that look like? A window requires a message map or table. This could literally be a chain of if-else statements or a big switch statement inside the window procedure. This does, however, quickly become unwieldy, and much effort has been spent in different libraries and frameworks to try to manage this somehow. In reality, it doesn't have to be anything fancy, and a simple static table will suffice in many cases. First, it helps to know what a window message consists of. Most importantly, there's a constant—such as WM_PAINT or WM_SIZE—that uniquely identifies the message. Two arguments, so to speak, are provided for every message, and these are called WPARAM and LPARAM, respectively. Depending on the message, these might not provide any information. Finally, Windows expects the handling of certain messages to return a value, and this is called the LRESULT. Most messages that your app handles, however, won't return a value and should instead return zero.

Given this definition, we can build a simple table for message handling using these types as building blocks:

```
typedef LRESULT (* message_callback)(HWND, WPARAM, LPARAM);

struct message_handler
{
  UINT message;
  message_callback handler;
};
```

At a minimum, we can then create a static table of message handlers, as shown in **Figure 1**.

The WM_PAINT message arrives when the window needs painting. This happens far less often than it did in earlier versions of Windows thanks to advances in rendering and composition of the desktop. The BeginPaint and EndPaint functions are relics of the GDI but are still needed even if you're drawing with an entirely different rendering engine. This is because they tell Windows that you're done painting by validating the window's drawing surface. Without these calls, Windows wouldn't consider the WM_PAINT message answered and your window would receive a steady stream of WM_PAINT messages unnecessarily.

The WM_DESTROY message arrives after the window has disappeared, letting you know that the window is being destroyed. This is usually an indicator that the app should terminate, but the GetMessage function inside the message loop is still waiting for the WM_QUIT message. Queuing this message is the job of the PostQuitMessage function. Its single parameter accepts a value that's passed along via WM_QUIT's WPARAM, as a way to return different exit codes when terminating the app.

The final piece of the puzzle is to implement the actual window procedure. I omitted the body of the lambda that I used to prepare the WNDCLASS structure previously, but given what you now know, it shouldn't be hard to figure out what it might look like:

```
wc.lpfnWndProc =
  [] (HWND window, UINT message,
      WPARAM wparam, LPARAM lparam) -> LRESULT
{
  for (auto h = s_handlers; h != s_handlers +
    _countof(s_handlers); ++h)
  {
    if (message == h->message)
    {
      return h->handler(window, wparam, lparam);
    }
  }

  return DefWindowProc(window, message, wparam, lparam);
};
```

The for loop looks for a matching handler. Fortunately, Windows provides default handling for messages that you choose not to process yourself. This is the job of the DefWindowProc function.

And that's it—if you've gotten this far, you've successfully created a desktop window using the Windows API!

The ATL Way

The trouble with these Windows API functions is that they were designed long before C++ became the smash hit that it is today, and thus weren't designed to easily accommodate an object-oriented view of the world. Still, with enough clever coding, this C-style API can be transformed

Figure 2 **Expressing a Window in ATL**

```
class Window : public CWindowImpl<Window, CWindow,
  CWinTraits<WS_OVERLAPPEDWINDOW | WS_VISIBLE>>
{
  BEGIN_MSG_MAP(Window)
    MESSAGE_HANDLER(WM_PAINT, PaintHandler)
    MESSAGE_HANDLER(WM_DESTROY, DestroyHandler)
  END_MSG_MAP()

  LRESULT PaintHandler(UINT, WPARAM, LPARAM, BOOL &)
  {
    PAINTSTRUCT ps;
    VERIFY(BeginPaint(&ps));

    // Dress up some pixels here!

    EndPaint(&ps);
    return 0;
  }

  LRESULT DestroyHandler(UINT, WPARAM, LPARAM, BOOL &)
  {
    PostQuitMessage(0);
    return 0;
  }
};
```

WPF lives!

XCEED Business Suite for WPF

The essential set of WPF controls for all your line-of-business solutions. Includes the industry-leading Xceed DataGrid for WPF.
A total of 85 tools!

XCEED
MULTI-TALENTED COMPONENTS

xceed.com

into something a little more suited to the average C++ programmer. ATL provides a library of class templates and macros that do just that, so if you need to manage more than a handful of window classes or still rely on USER and GDI resources for your window's implementation, there's really no reason not to use ATL. The window from the previous section can be expressed with ATL as shown in **Figure 2**.

The CWindowImpl class provides the necessary routing of messages. CWindow is a base class that provides a great many member function wrappers, mainly so you don't need to provide the window handle explicitly on every function call. You can see this in action with the BeginPaint and EndPaint function calls in this example. The CWinTraits template provides the window style constants that will be used during creation.

> Although it's simple to drop into your app and use, WTL packs a lot of power if you have sophisticated message filtering and routing needs.

The macros harken back to MFC and work with CWindowImpl to match incoming messages to the appropriate member functions for handling. Each handler is provided with the message constant as its first argument. This can be useful if you need to handle a variety of messages with a single member function. The final parameter defaults to TRUE and lets the handler decide at run time whether it actually wants to process the message or let Windows—or even some other handler—take care of it. These macros, along with CWindowImpl, are quite powerful and let you handle reflected messages, chain message maps together and so on.

To create the window, you must use the Create member function that your window inherits from CWindowImpl, and this in turn will call the good old RegisterClass and CreateWindow functions on your behalf:

```
Window window;
VERIFY(window.Create(nullptr, 0, L"Title"));
```

At this point, the thread again needs to quickly begin dispatching messages, and the Windows API message loop from the previous section will suffice. The ATL approach certainly comes in handy if you need to manage multiple windows on a single thread, but with a single top-level window, it's much the same as the Windows API approach from the previous section.

WTL: An Extra Dose of ATL

While ATL was designed primarily to simplify the development of COM servers and only provides a simple—yet extremely effective—window-handling model, WTL consists of a slew of additional class templates and macros specifically designed to support the creation of more-complex windows based on USER and GDI resources. WTL is now available on SourceForge (wtl.sourceforge.net), but for a new app using a modern rendering engine, it doesn't provide a great deal of value. Still, there are a handful of useful helpers. From the WTL atlapp.h header, you can use its message loop implementation to replace the hand-rolled version I described earlier:

```
CMessageLoop loop;
loop.Run();
```

Although it's simple to drop into your app and use, WTL packs a lot of power if you have sophisticated message filtering and routing needs. WTL also provides atlcrack.h with macros designed to replace the generic MESSAGE_HANDLER macro provided by ATL. These are merely conveniences, but they do make it easier to get up and running with a new message because they take care of cracking open the message, so to speak, and avoid any guesswork in figuring out how to interpret WPARAM and LPARAM. A good example is WM_SIZE, which packs the window's new client area as the low- and high-order words of its LPARAM. With ATL, this might look as follows:

```
BEGIN_MSG_MAP(Window)
  ...
  MESSAGE_HANDLER(WM_SIZE, SizeHandler)
END_MSG_MAP()

LRESULT SizeHandler(UINT, WPARAM, LPARAM lparam, BOOL &)
{
  auto width = LOWORD(lparam);
  auto height = HIWORD(lparam);

  // Handle the new size here ...

  return 0;
}
```

With the help of WTL, this is a little simpler:

```
BEGIN_MSG_MAP(Window)
  ...
  MSG_WM_SIZE(SizeHandler)
END_MSG_MAP()

void SizeHandler(UINT, SIZE size)
{
  auto width = size.cx;
  auto height = size.cy;

  // Handle the new size here ...
}
```

Notice the new MSG_WM_SIZE macro that replaced the generic MESSAGE_HANDLER macro in the original message map. The member function handling the message is also simpler. As you can see, there aren't any unnecessary parameters or a return value. The first parameter is just the WPARAM, which you can inspect if you need to know what caused the change in size.

The beauty of ATL and WTL is that they're just provided as a set of header files that you can include at your discretion. You use what you need and ignore the rest. However, as I've shown you here, you can get quite far without relying on any of these libraries and just write your app using the Windows API. Join me next time, when I'll show you a modern approach for actually rendering the pixels in your app's window. ■

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

Thanks to the following technical expert for reviewing this article: *Worachai Chaoweeraprasit*

Exploring the New JavaScript API for Office

Stephen Oliver and Eric Schmidt

This article is the first in a series of in-depth looks at the JavaScript API for Office, newly introduced in Microsoft Office 2013. It presupposes that you're familiar with apps for Office. If not, the MSDN documentation page, "Overview of apps for Office" (bit.ly/12nBWHG), provides a broad overview of and general introduction to the API.

This article and the others in this series, while not exhaustive, go deep into the API, touching on key aspects that will give you a solid, richer understanding of how the apps for Office API works.

In this first article, we review the apps for Office object model. Part 2 will focus on the core task of how to access Office file content and will review the event model. Part 3 will consider the concept of data binding and examine the basics of working with custom XML parts. Finally, Part 4 will close the series with a focused look at mail apps.

Throughout this series, we often make reference to the apps for Office API documentation. You can find the official documentation, code samples and community resources at the Apps for Office and SharePoint Developer Center on MSDN (dev.office.com).

This article discusses:

- Overview of the JavaScript API for Office
- The asynchronous programming pattern
- Object model hierarchy
- Testing for host application support

Technologies discussed:

JavaScript API for Office

Overview of the JavaScript API for Office

The JavaScript API for Office comprises a complete object model. The API is contained within a set of JavaScript files, starting with the office.js file. An app must include a reference to the office.js file to use the JavaScript API for Office. On load, the office.js file loads the other required scripts that it needs to operate, including the scripts needed for the host environment and the locale strings. Fortunately, you can add a reference to the office.js file using a content delivery network (CDN), so you don't need to deploy a copy of the office.js file along with your app. Here's an example:

```
<!-- When deploying an app, you should always
  load the CDN version of the office.js file. -->
<script src=
  "https://appsforoffice.microsoft.com/lib/1.0/hosted/office.js">
</script>
```

The object model was designed around several goals:

1. **"Write once, run everywhere."** It had to be extensible—not tied to a specific host application, but built around capabilities available in multiple host applications. Apps access host-specific functionality in a consistent way.
2. **Cross-platform.** Compatibility ranked high on this list, too; thus, the object model isn't tied to a specific version of Office. As well, the same code works on the Web App versions of the Office client applications, where supported. For example, an app for Excel can work on the Excel Web App just as well as in the Excel client application.

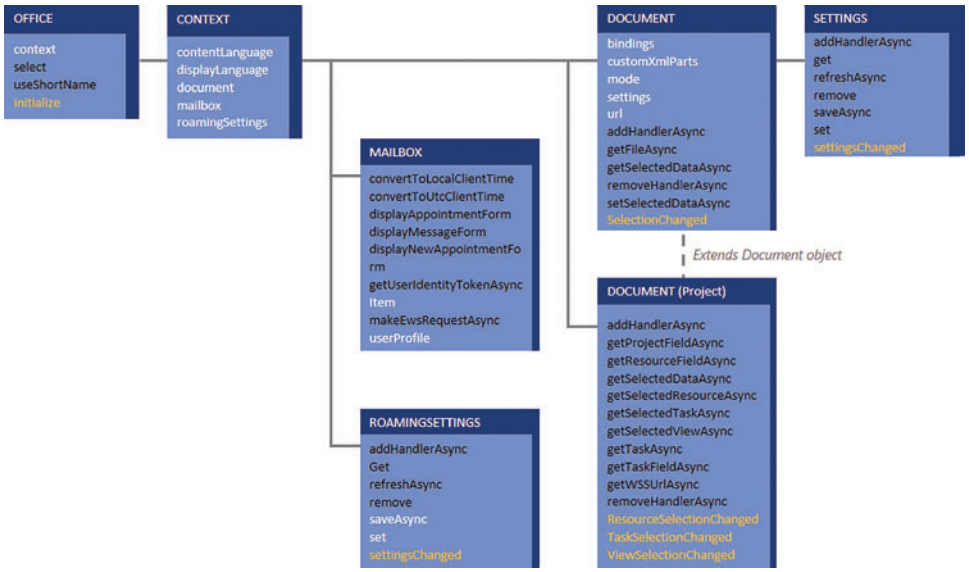


Figure 1 **The Object Model Hierarchy in the JavaScript API for Office**

3. **Performance and security.** It needed to be maximized for performance, so that apps can be as unobtrusive to users as possible. Also, the JavaScript API was designed to interact directly with document content without having to automate the Office applications, improving the stability and security of the solutions.

Another key goal for the JavaScript API was to attract Web developers to the Office platform. Thus, the object model was built with modern Web programming in mind. You can leverage your current skills and knowledge of other JavaScript libraries, such as jQuery, when creating apps in conjunction with the JavaScript API for Office.

The Asynchronous Programming Pattern

As mentioned, performance was a key goal in the design of the apps for Office API. One of the ways that the designers enhanced the performance of the API was through the heavy use of asynchronous functions.

The use of asynchronous functions avoids blocking an app during execution in the event a function takes a while to return. The asynchronous function is invoked, but program execution doesn't wait for the function to return. Instead, it continues while the asynchronous function is still executing. This allows the user to continue to use the Office document while the app is potentially still working.

Some key points for understanding the asynchronous design in the apps for Office API covered in this section are:

- The common signature of asynchronous functions in the apps for Office API
- The use of optional parameters in asynchronous functions
- The role of the AsyncResult object in asynchronous functions

We'll discuss each in turn.

Common Signature of Asynchronous Functions in the Apps for Office API All asynchronous functions in the apps for Office API have the same naming convention and the same basic signature. Every asynchronous function name ends in "Async," for example, like this: Document.getSelectedDataAsync.

The signature for all asynchronous functions adheres to the following basic pattern:

```
functionNameAsync(
  requiredParameters,
  [, options], [callback]);
```

The required parameters are followed by two other parameters: an object that holds optional parameters and a callback function, both of which are always optional.

Optional Parameters in Asynchronous Functions The optional JavaScript object in the signature of asynchronous functions is a collection of key/value pairs, separated by a colon, where the key is the name of the parameter and the value is the data that you want to use for that parameter. The order of the key/value pairs doesn't matter as long as the parameter name is correct. The MSDN documentation for each asynchronous function details what parameters are available to use in the options object for that particular function.

For example, the Document.setSelectedDataAsync method has the same basic signature common to all asynchronous functions in the apps for Office:

```
Office.context.document.setSelectedDataAsync(
  data [, options], callback);
```

> One of the ways that the designers enhanced the performance of the API was through the heavy use of asynchronous functions.

Like all asynchronous functions in the API, Document.setSelectedDataAsync has an options object that holds optional parameters, but the parameters for its options object are different from those for other asynchronous functions in the API, because the point of this function is to set data. So the optional parameters for Document.setSelectedDataAsync are related to setting data:

- coercionType: A CoercionType enumeration that specifies the format for the data you insert (text, HTML, OOXML, table or matrix)
- asyncContext: A user-defined object that's returned unchanged in the AsyncResult object that's passed in to the callback function as its only parameter

That same concept applies to all the other asynchronous functions.

You can either supply the object that contains the optional parameters as an object literal inline in the asynchronous function call, or

DevExpress

You used to think "Impossible"

Your Apps, Any Device

Now you think—game on!! The new tools in 12.2 help you envision and create engaging applications for the Web that can be accessed by mobile users on the go. And, with our Windows 8 XAML and JS tools you will begin to create highly interactive applications that address your customer needs today and build next generation touch enabled solutions for tomorrow.

Download your 30-day trial at www.DevExpress.com

DXv2

The next generation of inspiring tools. **Today.**

u | dxp | dxt | asp | win | wpf | sl | cr | vcl

Figure 2 **Availability of Capabilities in the JavaScript API for Office by Host Application**

Capability	Word	Excel/Excel Web App	PowerPoint	Outlook/Outlook Web App	Project
Get/set data as text, table, matrix	All	All	Text only		Text only
Settings	All	All	All	(RoamingSettings)	
Get file	All		Compressed only		
Bindings	All	All			
Custom XML Parts	All				
HTML and OOXML	All				
Mailbox				All	

create an object first and then pass that object in for the parameter. Following are two code samples that show both ways of supplying the options object using the Document.setSelectedDataAsync function.

Passing the options parameter inline:

```
function setData(data) {
  Office.context.document.setSelectedDataAsync(data, {
  coercionType: Office.CoercionType.Text }
  );
}
```

Passing the options parameter in a JavaScript object:

```
function setData(data) {
  var options = { coercionType: Office.CoercionType.Text };
  Office.context.document.setSelectedDataAsync(data, options );
}
```

The Role of the AsyncResult Object in Asynchronous Functions The third parameter in the common signature for asynchronous functions in the JavaScript API for Office is the optional callback parameter. The callback parameter is exactly as it sounds: a function you provide that's invoked when the asynchronous operation completes. Of course, you can provide either a named function or an anonymous function inline in the call to the asynchronous function. The key thing to note here is the role of the AsyncResult object with respect to the callback function.

When the runtime invokes your callback, it passes in an AsyncResult object as the only argument for the callback. The AsyncResult object contains information about the asynchronous operation, such as: whether or not the operation succeeded; what errors, if any, occurred; and the return value, if any, of the asynchronous function. In fact, in all asynchronous functions that return some kind of data or object, the AsyncResult is *the only way you can get at the returned value*. You do this using the AsyncResult.value property.

For example, the following code snippet gets the size of the document and displays it in the specified HTML element on the app UI. In order to get the file size, you first get the file object that the Document.getFileAsync method returns through the AsyncResult.value property. Here's how to do this:

```
function getFileData(elementId) {
  Office.context.document.getFileAsync(Office.FileType.Text,
  function (asyncResult) {
    if (asyncResult.status === 'succeeded') {
      var myFile = asyncResult.value;
      $(elementId).val(myFile.size);
    }
  });
}
```

The getFileData function calls the Document.getFileAsync method, specifying that it should return the file content as text. It then uses the value property of the AsyncResult object passed in to the anonymous function callback to get a reference to the File object. Then it displays the size of the file in the specified element using the size property of the File object. In a similar way, you'll use the AsyncResult.value property to get the return value of any asynchronous function in the apps for Office API.

You can read more about the Document.getFileAsync method in the next article of this series.

Object Model Hierarchy

The JavaScript API for Office aims to provide compatibility across versions of Office and symmetry across different host applications. To support these goals, the JavaScript API has a lean object model with a distinct hierarchy that isn't directly tied to any specific host application. Instead, the object model hosts a targeted set of capabilities for interacting with Office documents, scoped to the type of app (task pane, content or mail app) using them.

Figure 1 provides an abbreviated overview of the top-level hierarchy of objects in the JavaScript API for Office (note that the entire object model isn't shown). Specifically, the diagram demonstrates the relationships between the Office, Context, Document, Settings, Mailbox and RoamingSettings objects.

Each host application (Word, Excel, Excel Web App, PowerPoint, Project, Outlook and Outlook Web App) can use a subset of the capabilities included in the API. For example, roughly 40 percent of the object model pertains solely to mail apps that can only be used in Outlook and the Outlook Web App. Another portion of the object model allows interaction with Custom XML Parts, which is only available in Word 2013.

Figure 2 shows the capabilities available to specific host applications.

Shared Objects in the Object Model The JavaScript API for Office has a definitive entry point, the Office object, which is available to all types of apps and in all of the host applications. The

Figure 3 **Storing a Reference to the Document Object When the App Initializes**

```
// Add a handler to the initialize event of the Office object
Office.initialize = function (reason) {
  $(document).ready(function () {
    app.get_Document(Office.context.document);

    // Other initialization logic goes here
  })
}

// Use a self-executing anonymous function to encapsulate the
// functionality that the app uses
var app = (function () {

  var _document;
  function get_Document(officeDocument) {
    _document = officeDocument;
  }

  // Other fields and functions associated with the app

  return {
    get_Document: get_Document
    // Other exposed members
  };
})()
```

Office object represents a specific instance of an app inserted into a document, workbook, presentation, project, e-mail message or appointment. It can access bindings between the app and the document using the *select* method. (We'll discuss bindings in greater depth in a future article.) Most importantly, the Office object exposes the initialize event for the app, which allows you to build initialization logic for the app (more on that in a future article). Finally, the Office object contains a reference to the Context object for the app.

The Context object, which is also available to all types of apps and in all of the host applications, exposes information about the runtime environment that's hosting the app. In addition to storing the language settings for the app, the Context object provides the entry point to runtime capabilities in the JavaScript API for Office that are specific to the host in which the app was activated.

For example, you can access the document (Document object) associated with the app through the Context.document property. However, this property returns a value only when called from within a host application that supports it, that is, from within a task pane or content app. If we attempt to access the Context.document property from a mail app, we'll get an "undefined object" error. Likewise with the Context.mailbox property: In a mail app, it returns the mailbox (Mailbox object) opened in the host application. In a task pane app, it's undefined.

Support for Task Pane and Content Apps in the Object Model For task pane and content apps, the Document object represents the document, workbook, presentation or project into which the app has been inserted. The Document object provides the highest degree of access to the file's content—in essence, it's the primary point of contact between an app and an Office document.

Almost all of the techniques for accessing the content in the Office document require use of the Document object. For this reason, you might want to capture a reference to the Document object when the app initializes as shown in **Figure 3**.

When an app is activated within a Project file, the Document object exposes additional, specific capabilities targeted for Project files. Through the Document object, an app can get data for specific tasks, views, fields and resources in the project. An app can also add event listeners to monitor when the user changes the selected view, task or resource selected in the project. (We'll talk more about using the Document object in an app for Project in the next article.)

Also exposed by the Document object is the Settings object, which represents the "property bag" for an app. An app can store and persist custom properties across app sessions in the same document using the Settings object. The properties travel with the document: If you share an Office file that contains an app with someone else, the custom properties stored in the app will be available when the other person reads the file.

Storing and retrieving settings using the property bag is simple. The Settings.set method creates the setting in memory as a key/

value pair. To get the properties out of the property bag, we use the Settings.get method, passing in the setting's name (key), to get the value. Both the set and get methods are synchronous. To store the settings across sessions, we need to call the Settings.saveAsync method, which saves all of the custom properties contained in the app when the document is saved.

The code sample, "Apps for Office: Persist custom settings" (bit.ly/UEiZff), provides additional examples of how to use the Settings object and how to store data in an app.

Support for Mail Apps in the Object Model For mail apps, the Mailbox object provides the entry point of data access for mail-app-specific functionality. As the name implies, the Mailbox object corresponds to the mailbox of the current user and travels to wherever users read their e-mail—either in the Outlook client application or in the Outlook Web App. In addition to providing access to individual e-mail messages and appointments (through the Mailbox.item property), the Mailbox object allows the app to create new appointments, access the profile of the local user and even get the user's local time.

Like the Document object for content and task pane apps, you might want to capture a reference to the Mailbox object when the app initializes, as shown in **Figure 4**.

The RoamingSettings object, which is also available only in mail apps, is similar to the Settings object for document-centric apps (task pane and content apps). It allows apps to persist custom properties as name/value pairs across sessions. However, unlike the Settings object, which saves the custom properties within the host Office file, the RoamingSettings object saves the custom settings to the current user's mailbox. This makes the custom properties available to the app no matter what message the user is looking at or how the user has accessed his mailbox (in Outlook or the Outlook Web App).

For more information about the object model hierarchy in the JavaScript API for Office, see the MSDN documentation page, "Understanding the JavaScript API for Office" (bit.ly/UV2POY).

Figure 4 **Storing a Reference to the Mailbox Object in a Global Variable When the App Initializes**

```
// Add a handler to the initialize event of the Office object
Office.initialize = function (reason) {
  $(document).ready(function () {
    app.get_Mailbox(Office.context.mailbox);

    // Other initialization logic goes here
  })
}

// Use a self-executing anonymous function to encapsulate the
// functionality that the app uses
var app = (function () {

  var _mailbox;
  function get_Mailbox(mailbox) {
    _mailbox = mailbox;
  }

  // Other fields and functions associated with the app

  return {
    get_Mailbox: get_Mailbox
    // Other exposed members
  };
})()
```

Testing Whether a Capability Can Be Used in a Host Application

As we alluded to earlier, one of the strengths of the JavaScript API for Office is the "develop once, host many places" nature of apps for Office. For example, the same task pane app can be activated within Word, Excel, Project and PowerPoint (provided that its manifest allows all of those capabilities).

Yet, because not all apps have access to the exact same list of capabilities, an app could be inserted into a host application that doesn't allow the capabilities that the app requires. For example, Project currently doesn't provide access to the Settings object. An app that tries to access the Settings object when inserted into Project will raise an "undefined object" error.

Thus, developers must include logic in their apps for testing the availability of the capabilities they need. In the example with Project, the best technique for detecting capabilities in a host application is through a simple *if* block:

```
// Test for Settings object in host application
if (Office.context.document.settings) {

  // Provide implementation that uses the Settings object

}
else {

  // Use some other technique for saving custom properties,
  // like localStorage, sessionStorage or cookies

}
```

For more information about how to detect whether a member is available in the host application, see the MSDN documentation page, "How to: Determine host application support for specific API members," at bit.ly/TR5ZIB.

To summarize this article, we've discussed the meat and potatoes of the JavaScript API for Office. We described the object model hierarchy at a high level and discussed the asynchronous pattern as it's implemented in the object model. We also described how to test whether a capability is supported in a host application.

In the next article in this series, we'll take a closer look at the simplest, yet most powerful ways for working with data in an app for Office. We'll describe how to get and set selected data in more depth. We'll look at getting all of the file content and how to parse it. Also, we'll discuss apps in Project and how to read task, resource and view data. Finally, we'll review the event model in the JavaScript API for Office: what events you can code against and how to handle the results. ■

Stephen Oliver *is a programming writer in the Office Division and a Microsoft Certified Professional Developer (SharePoint 2010). He writes the developer documentation for the Excel Services and Word Automation Services, along with PowerPoint Automation Services developer documentation. He helped curate and design the Excel Mashup site at ExcelMashup.com.*

Eric Schmidt *is a programming writer in the Office Division. He has created several code samples for apps for Office, including the popular "Persist custom settings" code sample. In addition, he has written articles and created videos about other products and technologies within Office programmability.*

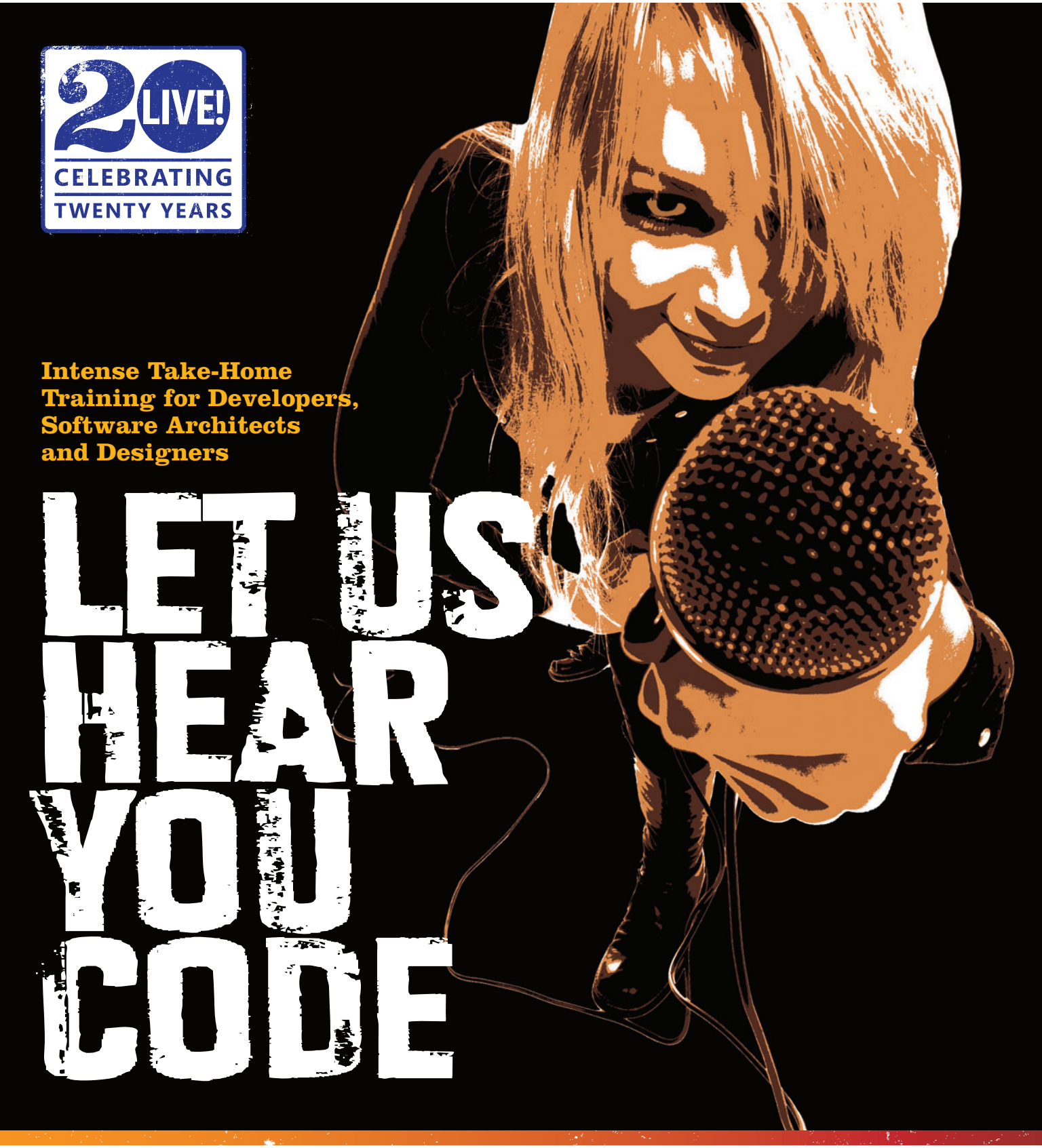
Thanks to the following technical experts for reviewing this article: *Mark Brewster, Shilpa Kothari and Juan Balmori Labra*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

20 LIVE! CELEBRATING TWENTY YEARS



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

LET US HEAR YOU CODE

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA

LAS VEGAS | MARCH 25-29, 2013

MGM Grand Hotel & Casino

EARLY BIRD SPECIAL

Register before February 27 and save $300!

Use Promo Code LVFEB4

Everyone knows all the *really* cool stuff happens behind the scenes. Get an all-access look at the Microsoft Platform and practical, unbiased, developer training at Visual Studio Live! Las Vegas.

Topics will include:

- ASP.NET
- Azure / Cloud Computing
- Cross-Platform Mobile
- Data Management
- HTML5 / JavaScript
- Developer Deep Dive: SharePoint / Office 365
- Developer Deep Dive: SQL Server
- Windows 8 / WinRT
- WPF / Silverlight
- Visual Studio 2012 / .NET 4.5

TURN THE PAGE FOR MORE EVENT DETAILS

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

CODE WITH .NET ROCKSTARS AND LEARN HOW TO MAXIMIZE THE DEVELOPMENT CAPABILITIES OF VISUAL STUDIO AND .NET DURING 5 ACTION-PACKED DAYS OF PRE- AND POST-CONFERENCE WORKSHOPS, 70+ SESSIONS LED BY EXPERT INSTRUCTORS AND KEYNOTES BY INDUSTRY HEAVYWEIGHTS.

BONUS LAS VEGAS CONTENT!

DEVELOPER DEEP DIVES ON SHAREPOINT AND SQL SERVER – BROUGHT TO YOU BY:

SharePoint LIVE!
TRAINING FOR COLLABORATION

SQL Server LIVE!
TRAINING FOR DBAs AND IT PROS

AGENDA AT-A-GLANCE

Windows 8/ WinRT | WPF/ Silverlight | ASP.NET | Visual Studio 2012/ .NET 4.5 | SharePoint / Office

START TIME	END TIME	Visual Studio Live! Pre-Conference Workshops: Monday, March 25,
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries
9:00 AM	6:00 PM	**MW01** - Workshop: Build a Windows 8 Application in a Day - *Rockford Lhotka*
		MW02 - Workshop: Services - Using WCF and ASP.NET Web API - *Miguel Castro*

START TIME	END TIME	Visual Studio Live! Day 1: Tuesday, March 26, 2013
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries
8:00 AM	9:00 AM	Keynote: To Be Announced
9:15 AM	10:30 AM	**T01** - A Primer in Windows 8 Development with WinJS - *Philip Japikse*
		T02 - jQuery Fundamentals - *Robert Boedigheimer*
10:45 AM	12:00 PM	**T06** - Windows 8 Style Apps - Design Essentials - *Billy Hollis*
		T07 - Hate JavaScript? Try TypeScript. - *Ben Hoelting*
12:00 PM	2:30 PM	Lunch & Expo Hall
1:15 PM	2:15 PM	**T11** - Chalk Talk: MVVM in Practice aka ""Code behind""-free XAML - *Tiberiu Covaci*
		T12 - Chalk Talk: Neural Networks for Developers - *James McCaffrey*
2:30 PM	3:45 PM	**T15** - New XAML controls in Windows 8 - *Billy Hollis*
		T16 - Tips for building Multi-Touch Enabled Web Sites - *Ben Hoelting*
3:45 PM	4:15 PM	Networking Break
4:15 PM	5:30 PM	**T20** - Make your App Alive with Tiles and Notifications - *Ben Dewey*
		T21 - Build Speedy Azure Applications with HTML 5 and Web Sockets Today - *Rick Garibay*
5:30 PM	7:00 PM	Welcome Reception

START TIME	END TIME	Visual Studio Live! Day 2: Wednesday, March 27, 2013
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries
8:00 AM	9:00 AM	Keynote: To Be Announced
9:15 AM	10:30 AM	**W01** - Building Your First Windows Phone 8 Application - *Brian Peek*
		W02 - Azure - *Vishwas Lele*
10:45 AM	12:00 PM	**W06** - Cross Win8/WP8 Apps - *Ben Dewey*
		W07 - Azure - *Vishwas Lele*
12:00 PM	2:30 PM	Birds-of-a-Feather Lunch & Expo Hall
1:15 PM	2:15 PM	**W11** - Chalk Talk: Moving Web Apps to the Cloud - *Eric D. Boyd*
		W12 - Chalk Talk: Improving Web Performance - *Robert Boedigheimer*
2:30 PM	3:45 PM	**W15** - Designing Your Windows Phone Apps for Multitasking and Background Processing - *Nick Landry*
		W16 - IaaS in Windows Azure with Virtual Machines - *Eric D. Boyd*
3:45 PM	4:15 PM	Sponsored Break - Exhibitor Raffle
4:15 PM	5:30 PM	**W20** - Building a Windows Runtime Component with C# - *Brian Peek*
		W21 - Bringing Open Source to Windows Azure: A Match Made in Heaven - *Jesus Rodriguez*
6:30 PM	8:30 PM	Evening Event

START TIME	END TIME	Visual Studio Live! Day 3: Thursday, March 28, 2013
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries
8:00 AM	9:15 AM	**TH01** - Building Extensible XAML Client Apps - *Brian Noyes*
		TH02 - JavaScript, Meet Cloud: Node.js on Windows Azure - *Sasha Goldshtein*
9:30 AM	10:45 AM	**TH06** - Migrating from WPF or Silverlight to WinRT - *Rockford Lhotka*
		TH07 - Using Windows Azure to Build the Next Generation of Mobile Applications - *Jesus Rodriguez*
11:00 AM	12:15 PM	**TH11** - Managing the .NET Compiler - *Jason Bock*
		TH12 - Cloud Backends for Your Mobile Apps: Windows Azure Mobile Services and Parse - *Sasha Goldshtein*
12:15 PM	1:30 PM	Lunch
1:30 PM	2:45 PM	**TH16** - Understanding Dependency Injection and Those Pesky Containers - *Miguel Castro*
		TH17 - Using Windows Azure for Solving Identity Management Challenges - *Michael Collier*
3:00 PM	4:15 PM	**TH21** - Static Analysis in .NET - *Jason Bock*
		TH22 - Elevating Windows Azure Deployments - *Michael Collier*

START TIME	END TIME	Visual Studio Live! Post-Conference Workshops: Friday, March 29,
7:30 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries
8:00 AM	5:00 PM	**FW02** - Workshop: Happy ALM with Visual Studio 2012 and Team Foundation Server 2012 - *Brian Randell*

Speakers and sessions subject to change

LAS VEGAS | MARCH 25-29, 2013

Azure / Cloud Computing	Data Management	HTML5 / JavaScript	Cross-Platform Mobile	SQL Server

2013 (Separate entry fee required)

MW03 - Workshop: HTML5 + Cloud - Reach Everyone, Everywhere - *Eric D. Boyd*	**MW04** - Workshop: SharePoint 2013 Developer Boot Camp - *Andrew Connell*	
T03 - Busy Developer's Guide to MongoDB - *Ted Neward*	**T04** - Mastering Visual Studio 2012 - *Deborah Kurata*	**T05** - Building Your First SharePoint 2013 Application Using Visual Studio 2012 - *Darrin Bishop*
T08 - Busy Developer's Guide to Cassandra - *Ted Neward*	**T09** - IntelliTrace, What is it and How Can I Use it to My Benefits? - *Marcel de Vries*	**T10** - Getting Started with Microsoft Office 365 SharePoint Online Development
T13 - Chalk Talk: Building a URL Shortening Service with Node.js - *Rick Garibay*	**T14** - Chalk Talk: Acing Application Lifecycle Management in SharePoint	
T17 - What's New in ASP.NET 4.5 - *Adam Tuliper*	**T18** - Team Foundation Server 2012 Builds: Understand, Configure, and Customize - *Benjamin Day*	**T19** - Unit Testing in SharePoint - *Jim Wooley*
T22 - 25 Tips and Tricks for the ASP.NET Developer - *Adam Tuliper*	**T23** - Design for Testability: Mocks, Stubs, Refactoring, and User Interfaces - *Benjamin Day*	**T24** - Better Together - SharePoint 2013 and Mobile Development - *Darrin Bishop*
W03 - Controlling ASP.NET MVC4 - *Philip Japikse*	**W04** - Modern ALM and the DevOps Story - *Brian Randell*	**W05** - Developing and Extending Enterprise Content Management Features with SharePoint 2013 - *Paul Swider*
W08 - MVC For WebForms Developers: Comparing and Contrasting - *Miguel Castro*	**W09** - OData - *Sergey Barskiy*	**W10** - Use 2012 (and Beyond) Technology with SharePoint 2010 - *Ryan McIntyre*
W13 - Chalk Talk: NoSQL for the SQL Guy - *Ted Neward*	**W14** - Demystifying the Microsoft UI Technology Roadmap - *Brian Noyes*	
W17 - ASP.NET MVC - AJAX in your Views - *Walt Ritscher*	**W18** - Code First and Entity Framework - *Sergey Barskiy*	**W19** - Build Modern Collaborative Solutions with Office 2013, "Napa" Office 365 Development Tools, and SharePoint 2013 - *Brian Randell*
W22 - Patterns for Parallel Programming - *Tiberiu Covaci*	**W23** - LINQ performance and Scalability - *Jim Wooley*	**W24** - Intro to Windows Azure SQL Database and What's New - *Eric D. Boyd*
TH03 - To Be Announced	**TH04** - Sharing up to 80% of code building Mobile apps for iOS, Android, WP 8 and Windows 8 - *Marcel de Vries*	**TH05** - SQL Server Data Tools - *Leonard Lobel*
TH08 - ASP.NET MVC - Routing in the spotlight - *Walt Ritscher*	**TH09** - iOS Development Survival Guide for the .NET Guy - *Nick Landry*	**TH10** - Programming the T-SQL Enhancements in SQL Server 2012 - *Leonard Lobel*
TH13 - From 0 to Web Site in 60 Minutes with Web Matrix - *Mark Rosenberg*	**TH14** - To Be Announced	**TH15** - Getting to know the BI Semantic Model - *Andrew Brust*
TH18 - Creating Web Sites Using Visual Studio LightSwitch - *Michael Washington*	**TH19** - Building Multi-Platform Mobile Apps with Push Notifications - *Nick Landry*	**TH20** - Big Data-BI Fusion: Microsoft HDInsight & MS BI - *Andrew Brust*
TH23 - Building Single Page Web Applications with HTML5, ASP.NET MVC4 and Web API - *Marcel de Vries*	**TH24** - Create HTML 5 Mobile websites with Visual Studio LightSwitch - *Michael Washington*	**TH25** - Optimizing Stored Procedures - *Mark Rosenberg*

2013 (Separate entry fee required)

FW02 - Workshop: SQL Server 2012 - *Andrew Brust & Leonard Lobel*

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

linkedin.com – Join the "VSLive" group!

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

Register at vslive.com/lasvegas

Use Promo Code LVFEB4

Async Causality Chain Tracking

Andrew Stasyuk

With the advent of C# 5, Visual Basic .NET 11, the Microsoft .NET Framework 4.5 and .NET for Windows Store apps, the asynchronous programming experience has been streamlined greatly. New async and await keywords (Async and Await in Visual Basic) allow developers to maintain the same abstraction they were used to when writing synchronous code.

A lot of effort was put into Visual Studio 2012 to improve asynchronous debugging with tools such as Parallel Stacks, Parallel Tasks, Parallel Watch and the Concurrency Visualizer. However, in terms of being on par with the synchronous code debugging experience, we're not quite there yet.

One of the more prominent issues that breaks the abstraction and reveals internal plumbing behind the async/await façade is the lack of call stack information in the debugger. In this article, I'm going to provide means to bridge this gap and improve the asynchronous debugging experience in your .NET 4.5 or Windows Store app.

Let's settle on essential terminology first.

This article discusses:

- Causality chains versus return stacks
- Asynchronous debugging with existing tools
- Preserving causality chains in classic and Windows Store apps
- Using EventSource and EventListener
- Emulating async-local storage
- Comparison of causality tracking approaches and caveats

Technologies discussed:

Visual Studio 2012, Microsoft .NET Framework 4.5, .NET for Windows Store Apps

Code download available at:

archive.msdn.microsoft.com/mag201302Causality

Definition of a Call Stack

MSDN documentation (bit.ly/Tukvkm) used to define call stack as "the series of method calls leading from the beginning of the program to the statement currently being executed at run time." This notion was perfectly valid for the single-threaded, synchronous programming model, but now that parallelism and asynchrony are gaining momentum, more precise taxonomy is necessary.

For the purpose of this article, it's important to distinguish the causality chain from the return stack. Within the synchronous paradigm, these two terms are mostly identical (I'll mention the exceptional case later). In asynchronous code, the aforementioned definition describes a causality chain.

On the other hand, the statement currently being executed, when finished, will lead to a series of methods continuing their execution. This series constitutes the return stack. Alternatively, for readers familiar with the continuation passing style (Eric Lippert has a fabulous series on this topic, starting at bit.ly/d9VODc), the return stack might be defined as a series of continuations that are registered to execute, should the currently executing method complete.

In a nutshell, the causality chain answers the question, "How did I get here?" while return stack is the answer for, "Where do I go next?" For example, if you've got a deadlock in your application, you might be able to find out what caused it from the former, while the latter would let you know what the consequences are. Note that while a causality chain always tracks back to the program entry point, the return stack is cut off at the point where the result

of asynchronous operation is not observed (for example, async void methods or work scheduled via ThreadPool.QueueUserWorkItem).

There's also a notion of stack trace being a copy of a synchronous call stack preserved for diagnostics; I'll use these two terms interchangeably.

Be aware that there are several unspoken assumptions in the preceding definitions:

- "Method calls" referred to in the first definition generally imply "methods that have not completed yet," which bear the physical meaning of "being on stack" in the synchronous programming model. However, while we're generally not interested in methods that have already returned, it's not always possible to distinguish them during asynchronous debugging. In this case, there's no physical notion of "being on stack" and all continuations are equally valid elements of a causality chain.
- Even in synchronous code, a causality chain and return stack aren't always identical. One particular case when a method might be present in one, but missing from the other, is a tail call. Though not directly expressible in C# and Visual Basic .NET, it may be coded in Intermediate Language (IL) ("tail." prefix) or produced by the just-in-time (JIT) compiler (especially in a 64-bit process).
- Last, but not least, causality chains and return stacks can be nonlinear. That is, in the most general case, they're directed graphs having current statement as a sink (causality graph) or source (return graph). Nonlinearity in asynchronous code is due to forks (parallel asynchronous operations originating from one) and joins (continuation scheduled to run upon completion of a set of parallel asynchronous operations). For the purpose of this article, and due to platform limitations (explained later), I'll consider only linear causality chains and return stacks, which are subsets of corresponding graphs.

Luckily, if asynchrony is introduced into a program by using async and await keywords with no forks or joins, and all async methods are awaited, the causality chain is still identical to the return stack, just as in synchronous code. In this case, both of them are equally useful in orienting yourself in the control flow.

On the other hand, causality chains are rarely equal to return stacks in programs employing explicitly scheduled continuations, a notable example being Task Parallel Library (TPL) dataflow. This is due to the nature of data flowing from a source block to a target block, never returning to the former.

Existing Tools

Consider a quick example:

```
static void Main()
{
  OperationAsync().Wait();
}

async static Task OperationAsync()
{
  await Task.Delay(1000);
  Console.WriteLine("Where is my call stack?");
}
```

By extrapolating the abstraction developers were used to in synchronous debugging, they would expect to see the following causality chain/return stack when execution is paused at the Console.WriteLine method:

```
ConsoleSample.exe!ConsoleSample.Program.OperationAsync() Line 19
ConsoleSample.exe!ConsoleSample.Program.Main() Line 13
```

But if you try this, you'll find that in the Call Stack window the Main method is missing, while the stack trace starts directly in the OperationAsync method preceded by [Resuming Async Method]. Parallel Stacks has both methods; however, it doesn't show that Main calls OperationAsync. Parallel Tasks doesn't help either, showing "No tasks to display."

Note: At this point the debugger is aware of the Main method being part of the call stack—you might have noticed that by the gray background behind the call to OperationAsync. The CLR and Windows Runtime (WinRT) have to know where to continue execution after the topmost stack frame returns; thus, they do indeed store return stacks. In this article, though, I'll only delve into causality tracking, leaving return stacks as a topic for another article.

Preserving Causality Chains

In fact, causality chains are never stored by the runtime. Even call stacks that you see when debugging synchronous code are, in essence, return stacks—as was just said, they're necessary for the CLR and Windows Runtime to know which methods to execute after the topmost frame returns. The runtime doesn't need to know what caused a particular method to execute.

To be able to view causality chains during live and post-mortem debugging, you have to explicitly preserve them along the way. Presumably, this would require storing (synchronous) stack trace information at every point where continuation is scheduled and restoring this data when continuation starts to execute. These stack trace segments could then be stitched together to form a causality chain.

We're more interested in transferring causality information across await constructs, as this is where abstraction of similarity with synchronous code breaks. Let's see how and when this data can be captured.

As Stephen Toub points out (bit.ly/yF8eGu), provided that FooAsync returns a Task, the following code:

```
await FooAsync();
RestOfMethod();
```

is transformed by the compiler to a rough equivalent of this:

```
var t = FooAsync();
var currentContext = SynchronizationContext.Current;
t.ContinueWith(delegate
{
  if (currentContext == null)
    RestOfMethod();
  else
    currentContext.Post(delegate { RestOfMethod(); }, null);
}, TaskScheduler.Current);
```

From looking at the expanded code, it appears there are at least two extension points that might allow for capturing causality information: TaskScheduler and SynchronizationContext. Indeed, both offer similar pairs of virtual methods where it should be possible to capture call stack segments at the right moments: QueueTask/TryDequeue on TaskScheduler and Post/OperationStarted on SynchronizationContext.

Unfortunately, you can only substitute default TaskScheduler when explicitly scheduling a delegate via the TPL API, such

as Task.Run, Task.ContinueWith, TaskFactory.StartNew and so on. This means that whenever continuation is scheduled outside a running task, the default TaskScheduler will be in force. Thus, the TaskScheduler-based approach won't be able to capture necessary information.

As for SynchronizationContext, although it's possible to override the default instance of this class for the current thread by calling the SynchronizationContext.SetSynchronizationContext method, this has to be done for every thread in the application. Thus, you'd have to be able to control thread lifetime, which is infeasible if you aren't planning to re-implement a thread pool. Moreover, Windows Forms, Windows Presentation Foundation (WPF) and ASP.NET all provide their own implementations of SynchronizationContext in addition to SynchronizationContext.Default, which schedules work to the thread pool. Hence, your implementation would have to behave differently depending on the origin of the thread in which it's working.

Also note that when awaiting a custom awaitable, it's entirely up to implementation whether to use SynchronizationContext to schedule a continuation.

Luckily, there are two extension points suitable for our scenario: subscribing to TPL events without having to modify the existing codebase, or explicitly opting in by slightly modifying every await expression in the application. The first approach only works in desktop .NET applications, while the second can accommodate Windows Store apps. I'll detail both in the following sections.

Introducing EventSource

The .NET Framework supports Event Tracing for Windows (ETW), having defined event providers for practically every aspect of the runtime (bit.ly/VDfrtP). Particularly, TPL fires events that allow you to track Task lifetime. Although not all of these events are documented, you can obtain their definitions yourself by delving into mscorlib.dll with a tool such as ILSpy or Reflector or peeking into framework reference source (bit.ly/HRU3) and searching for the TplEtwProvider class. Of course, the usual reflection disclaimer applies: If the API isn't documented, there's no guarantee that empirically observed behavior will be retained in the next release.

TplEtwProvider inherits from System.Diagnostics.Tracing.EventSource, which was introduced in the .NET Framework 4.5 and is now a recommended way to fire ETW events in your application (previously you had to deal with manual ETW manifest generation). In addition, EventSource allows for consumption of events in process, by subscribing to them via EventListener, also new in the .NET Framework 4.5 (more on this momentarily).

The event provider can be identified by either a name or GUID. Each particular event type is in turn identified by event ID and, optionally, a keyword to distinguish from other unrelated types of events fired by this provider (TplEtwProvider doesn't use keywords). There are optional Task and Opcode parameters that you might find useful for filtering, but I'll rely solely on event ID. Each event also defines the level of verbosity.

TPL events have a variety of uses besides causality chains, such as tracking of tasks in-flight, telemetry and so on. They don't fire for custom awaitables, though.

Introducing EventListener

In the .NET Framework 4, in order to capture ETW events, you had to be running an out-of-process ETW listener, such as Windows Performance Recorder or Vance Morrison's PerfView, and then correlate captured data with the state you observed in the debugger. This posed additional problems, as data was stored outside process memory space and crash dumps didn't include it, which made this solution less suitable for post-mortem debugging. For example, if you rely on Windows Error Reporting to provide dumps, you won't get any ETW traces and thus causality information will be missing.

However, starting in the .NET Framework 4.5, it's possible to subscribe to TPL events (and other events fired by EventSource inheritors) via System.Diagnostics.Tracing.EventListener (bit.ly/XJelwF). This allows the capture and preservation of stack trace segments *in the process* memory space. Therefore, a mini-dump with heap should be enough to extract causality information. In this article, I'll only detail EventListener-based subscriptions.

It's worth mentioning that the advantage of an out-of-process listener is that you can always get the call stacks by listening to the Stack ETW Events (either relying on an existing tool or doing tedious stack walking and module address tracking yourself). When subscribing to the events using EventListener, you can't get call stack information in Windows Store apps, because the StackTrace API is prohibited. (An approach that works for Windows Store apps is described later.)

In order to subscribe to events, you have to inherit from EventListener, override the OnEventSourceCreated method and make sure that an instance of your listener gets created in every AppDomain of your program (subscription is per application domain). After EventListener is instantiated, this method will be called to notify the listener of event sources that are being created. It will also provide notifications for all event sources that existed before the listener was created. After filtering event sources either by name or GUID (performance-wise, comparing GUIDs is a better idea), a call to EnableEvents subscribes the listener to the source:

```
private static readonly Guid tplGuid =
  new Guid("2e5dba47-a3d2-4d16-8ee0-6671ffdcd7b5");

protected override void OnEventSourceCreated(EventSource eventSource)
{
  if (eventSource.Guid == tplGuid)
    EnableEvents(eventSource, EventLevel.LogAlways);
}
```

To process events, you need to implement abstract method OnEventWritten. For the purpose of preserving and restoring stack trace segments, you need to capture the call stack right before an asynchronous operation is scheduled, and then, when it starts execution, associate a stored stack trace segment with it. To correlate these two events, you can use the TaskID parameter. Parameters passed to a corresponding event-firing method in an event source are boxed into a read-only object collection and passed in as the Payload property of EventWrittenEventArgs.

Interestingly, there are special fast paths for EventSource events that are consumed as ETW (not via EventListener), where boxing doesn't occur for their arguments. This does provide a performance improvement, but it's mostly zeroed out due to cross-process machinery.

In the OnEventWritten method, you need to distinguish between event sources (in case you subscribe to more than one) and identify

ComponentOne Studio Enterprise | from $1,315.60

ComponentOne a division of GrapeCity

.NET Tools for the Smart Developer: Windows, Web, and XAML.

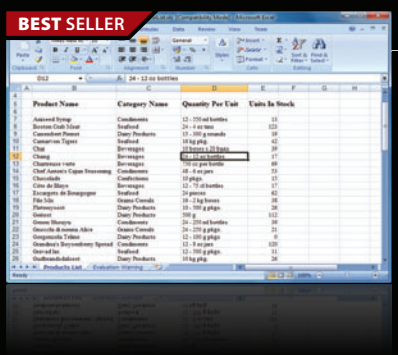
- Hundreds of UI controls for all .NET platforms including grids, charts, reports and schedulers
- Supports Visual Studio 2012 and Windows 8
- Now includes Windows 8 Studios for WinRT XAML and WinJS
- New Cosmopolitan (Windows 8 UI) theme provides a modern look and feel
- Royalty-free deployment and distribution

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



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
- Add charting, email, spell checking, barcode creation, OCR, diagramming, imaging, project management and file format management to your .NET applications
- Common uses also include mail merge, adding barcodes to documents, building dynamic Excel reports on the fly and extracting text from PDF files

Code Compare Pro | from $48.95

devart

An advanced visual file comparison tool with Visual Studio integration.

- Code oriented comparison, including syntax highlighting, unique structure and lexical comparison algorithms, for the most popular programming languages
- Smooth Visual Studio integration to develop and merge within one environment in the context of current solution, using native IDE editors
- Three-way file merge, folder comparison and synchronization

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 1 **Handling of TPL Events in the OnEventWritten Method**

```
protected override void OnEventWritten(EventWrittenEventArgs eventData)
{
  if (eventData.EventSource.Guid == tplGuid)
  {
    int taskId;
    switch (eventData.EventId)
    {
      case 7: // Task scheduled
        taskId = (int)eventData.Payload[2];
        stackStorage.StoreStack(taskId);
        break;
      case 10: // Task wait begin
        taskId = (int)eventData.Payload[2];
        bool waitBehaviorIsSynchronous =
          (int)eventData.Payload[3] == 1;
        if (!waitBehaviorIsSynchronous)
          stackStorage.StoreStack(taskId);
        break;
      case 11: // Task wait end
        taskId = (int)eventData.Payload[2];
        stackStorage.RestoreStack(taskId);
        break;
    }
  }
}
```

the event itself. The stack trace will be captured (stored) when TaskScheduled or TaskWaitBegin events fire, and associated with a newly started asynchronous operation (restored) in TaskWaitEnd. You also need to pass in taskId as the correlation identifier. **Figure 1** shows the outline of how the events will be handled.

Note: Explicit values ("magic numbers") in code are a bad programming practice and are used here only for brevity. The accompanying sample code project has them conveniently structured in constants and enumerations to avoid duplication and risk of typos.

Note that in TaskWaitBegin, I check for TaskWaitBehavior being synchronous, which happens when a task being awaited is executed synchronously or has already completed. In this case, a synchronous call stack is still in place, so it doesn't need to be stored explicitly.

Async-Local Storage

Whatever data structure you choose to preserve call stack segments needs the following quality: Stored value (causality chain) should be preserved for every asynchronous operation, following control flow along the way across await boundaries and continuations, bearing in mind that continuations may execute on different threads.

This suggests a thread-local-like variable that would preserve its value pertaining to the current asynchronous operation (a chain of continuations), instead of a particular thread. It can be roughly named "async-local storage."

The CLR already has a data structure called ExecutionContext that's captured on one thread and restored on the other (where continuation gets to execute), thus being passed along with control flow. This is essentially a container that stores other contexts (SynchronizationContext, CallContext and so on) that might be needed to continue execution in exactly the same environment, where they were interrupted. Stephen Toub has the details at bit.ly/M0amHk. Most importantly, you can store arbitrary data in CallContext (by calling its static methods LogicalSetData and LogicalGetData), which seems to suit the aforementioned purpose.

Bear in mind that CallContext (actually, internally there are two of them: LogicalCallContext and IllogicalCallContext) is a heavy object, designed to flow across remoting boundaries. When no custom data is stored, the runtime doesn't initialize the contexts, sparing the cost of maintaining them with the control flow. As soon as you call the CallContext.LogicalSetData method, a mutable ExecutionContext and several Hashtables have to be created and passed along or cloned from then on.

Unfortunately, ExecutionContext (together with all its constituents) is captured before the described TPL events fire and restored shortly afterward. Thus, any custom data saved in CallContext in between is discarded after ExecutionContext is restored, which makes it unsuitable for our particular purpose.

In addition, the CallContext class isn't available in the .NET for Windows Store apps subset, so an alternative is needed for this scenario.

One way to build an async-local storage that would work around these problems is to maintain the value in thread-local storage (TLS) while the synchronous portion of code is executing. Then, when the TaskWaitStart event fires, store the value in a shared (non-TLS) dictionary, keyed by the TaskID. When the counterpart event, TaskWaitEnd, fires, remove the preserved value from the dictionary and save it back to TLS, possibly on a different thread.

As you might know, values stored in TLS are preserved even after a thread is returned to the thread pool and gets new work to execute. So, at some point, the value has to be removed from TLS (otherwise, some other asynchronous operation executing on this thread later might access the value stored by the previous operation as if it were its own). You can't do this in the TaskWaitBegin event handler because, in case of nested awaits, TaskWaitBegin and TaskWaitEnd events occur multiple times, once per await, and a stored value might be needed in between, such as in the following snippet:

```
async Task OuterAsync()
{
  await InnerAsync();
}
async Task InnerAsync()
{
  await Task.Delay(1000);
}
```

Instead, it's safe to consider that the value in TLS is eligible to be cleared when the current asynchronous operation is no longer being executed on a thread. Because the CLR doesn't have an in-process event that would notify of a thread being recycled back to the thread pool (there's an ETW one—bit.ly/ZfAWrb), for this purpose I'll use ThreadPoolDequeueWork fired by FrameworkEventSource (also undocumented), which occurs when a new operation is started on a thread pool thread. This leaves out non-pooled threads, for which you'd have to manually clean the TLS, such as when a UI thread returns to the message loop.

For a working implementation of this concept together with stack segments capturing and concatenation, please refer to the StackStorage class in the accompanying source code download. There's also a cleaner abstraction, AsyncLocal<T>, which allows you to store any value and transfer it with the control flow to subsequent asynchronous continuations. I'll use it as causality chain storage for Windows Store apps scenarios.

Tracing Causality in Windows Store Apps

The described approach would still hold up in a Windows Store scenario if the System.Diagnostics.StackTrace API were available. For better or for worse, it isn't, which means you can't get any information about call stack frames above the current one from within your code. Thus, even while TPL events are still supported, a call to TaskWaitStart or TaskWaitEnd is buried deep in the framework method calls, so you have no information about your code that caused these events to fire.

Luckily, .NET for Windows Store apps (as well as the .NET Framework 4.5) provides CallerMemberNameAttribute (bit.ly/PsDH0p) and its peers CallerFilePathAttribute and CallerLineNumberAttribute. When optional method arguments are decorated with these, the compiler will initialize the arguments with corresponding values *at compile time*. For example, the following code will output "Main() in c:\Full\Path\To\Program.cs at line 14":

```
static void Main(string[] args)
{
  LogCurrentFrame();
}

static void LogCurrentFrame([CallerMemberName] string name = null,
  [CallerFilePath] string path = null, [CallerLineNumber] int line = 0)
{
  Console.WriteLine("{0}() in {1} at line {2}", name, path, line);
}
```

This only allows the logging method to get information about the calling frame, which means you have to ensure it gets called from all the methods you want captured in the causality chain. One convenient location for this would be decorating each await expression with a call to an extension method, like this:

```
await WorkAsync().WithCausality();
```

Here, the WithCausality method captures the current frame, appends it to causality chain and returns a Task or awaitable (depending on what WorkAsync returns), which upon completion of the original one removes the frame from the causality chain.

As multiple different things can be awaited, there should be multiple overloads of WithCausality. This is straightforward for a Task<T> (and even easier for a Task):

```
public static Task<T> WithCausality<T>(this Task<T> task,
  [CallerMemberName] string member = null,
  [CallerFilePath] string file = null,
  [CallerLineNumber] int line = 0)
{
  var removeAction =
    AddFrameAndCreateRemoveAction(member, file, line);
  return task.ContinueWith(t => { removeAction(); return t.Result; });
}
```

However, it's trickier for custom awaitables. As you might know, the C# compiler allows you to await an instance of any type that follows a particular pattern (see bit.ly/AmAUIF), which makes writing overloads that would accommodate any custom awaitable impossible using static typing only. You may make a few shortcut overloads for awaitables predefined in the framework, such as YieldAwaitable or ConfiguredTaskAwaitable—or the ones defined in your solution—but in general you have to resort to the Dynamic Language Runtime (DLR). Handling all the cases requires a lot of boilerplate code, so feel free to look into the accompanying source code for details.

It's also worth noting that in case of nested awaits, WithCausality methods will be executed from inner to outer (as await expressions are evaluated), so care must be taken to assemble the stack in the correct order.

Viewing Causality Chains

Both described approaches keep causality information in memory as lists of call stack segments or frames. However, walking them and concatenating into a single causality chain for display is tedious to do by hand.

The easiest option to automate this is to leverage the debugger evaluator. In this case, you author a public static property (or method) on a public class, which, when called, walks the list of stored segments and returns a concatenated causality chain. Then you can evaluate this property during debugging and see the result in the text visualizer.

Unfortunately, this approach doesn't work in two situations. One occurs when the topmost stack frame is in native code, which is quite a common scenario for debugging application hangs, as kernel-based synchronization primitives do call into native code. The debugger evaluator would just display, "Cannot evaluate expression because the code of the current method is optimized" (Mike Stall describes these limitations in detail at bit.ly/SLINuT).

The other issue is with post-mortem debugging. You can actually open a mini-dump in Visual Studio and, surprisingly (given that there's no process to debug, only its memory dump), you're allowed to examine property values (run property getters) and even call some methods! This amazing piece of functionality is built into the Visual Studio debugger and works by interpreting a watch expression and all methods that it calls into (in contrast to live debugging, where compiled code gets executed).

Obviously, there are limitations. For example, while doing dump debugging, you can't in any way call into native methods (meaning that you can't even execute a delegate, because its Invoke method is generated in native code) or access some restricted APIs (such as System.Reflection). Interpreter-based evaluation is also expectedly slow—and, sadly, due to a bug, the evaluation timeout for dump debugging is limited to 1 second in Visual Studio 2012, regardless of configuration. This, given the number of method calls required to traverse the list of stack trace segments and iterate over all frames, prohibits the use of the evaluator for this purpose.

Luckily, the debugger always allows access to field values (even in dump debugging or when the top stack frame is in native code), which makes it possible to crawl through the objects constituting a stored causality chain and reconstruct it. This is obviously tedious, so I wrote a Visual Studio extension that does this for you (see accompanying sample code). **Figure 2** shows what the final experience looks like. Note that the graph on the right is also generated by this extension and represents the async equivalent of Parallel Stacks.

Comparison and Caveats

Both causality-tracking approaches are not free. The second one (caller-info-based) is more lightweight, as it doesn't involve the expensive StackTrace API, relying instead on the compiler to provide caller frame information during compile time, which means "free" in a running program. However, it still uses eventing infrastructure with its cost to support AsyncLocal<T>. On the other hand, the first approach provides more data, not skipping frames without awaits. It also automatically tracks several other situations where Task-based asynchrony arises without await, such as the Task.Run method; on the other hand, it does not work with custom awaitables.

WHAT IF DEVELOPING FOR HUNDREDS OF SERVERS WAS AS EASY AS ONE?

IT IS...
MEET FATCLOUD

The next generation cloud enabled application platform for .NET is here. For a free three-node developer download visit **FATCLOUD.COM**.

An additional benefit of the TPL events-based tracker is that existing asynchronous code doesn't have to be modified, while for the caller info attributes-based approach, you have to alter every await statement in your program. But only the latter supports Windows Store apps.

The TPL events tracker also suffers from a lot of boilerplate framework code in stack trace segments, though it can be easily filtered out by frame namespace or class name. See the sample code for a list of common filters.

Another caveat concerns loops in asynchronous code. Consider the following snippet:

```
async static Task Loop()
{
  for (int i = 0; i < 10; i++)
  {
    await FirstAsync();
    await SecondAsync();
    await ThirdAsync();
  }
}
```

By the end of the method, its causality chain would grow to more than 30 segments, repeatedly alternating between FirstAsync, SecondAsync and ThirdAsync frames. For a finite loop, this may be tolerable, though it's still a waste of memory to store duplicate frames 10 times. However, in some cases, a program might introduce a valid infinite loop, for example, in the case of a message loop. Moreover, infinite repetition might be introduced without loop or await constructs—a timer rescheduling itself on every tick is a perfect example. Tracking an infinite causality chain is a sure way to run out of memory, so the amount of data stored has to be reduced to a finite amount somehow.

This issue doesn't affect the caller-info-based tracker, as it removes a frame from the list immediately upon the start of a continuation. There are two (combinable) approaches to fix this for the TPL events scenario. One is to cut older data based on the rolling maximum storage amount. The other is to represent loops efficiently and avoid duplication. For both approaches, you might also detect common infinite loop patterns and cut the causality chain explicitly at these points.

Feel free to refer to the accompanying sample project to see how loop folding might be implemented.

As stated, the TPL events API only lets you capture a causality chain, not a graph. This is because the Task.WaitAll and Task.WhenAll methods are implemented as countdowns, where continuation is scheduled only when the last task comes in completed and the counter reaches zero. Thus, only the last completed task forms a causality chain.

Wrapping Up

In this article, you've learned the difference between a call stack, a return stack and a causality chain. You should now be aware of extension points that the .NET Framework provides to track scheduling and execution of asynchronous operations and be able to leverage these to capture and preserve causality chains. The approaches described cover tracking causality in classic and Windows Store apps, both in live and post-mortem debugging scenarios. You also learned about the concept of async-local storage and its possible implementation for Windows Store apps.

Now you can go ahead and incorporate causality tracking into your asynchronous codebase or use async-local storage in parallel calculations; explore the event sources that the .NET Framework 4.5 and .NET for Windows Store apps offer to build something new, such as a tracker for unfinished tasks in your program; or use this extension point to fire your own events to fine-tune the performance of your application. ■

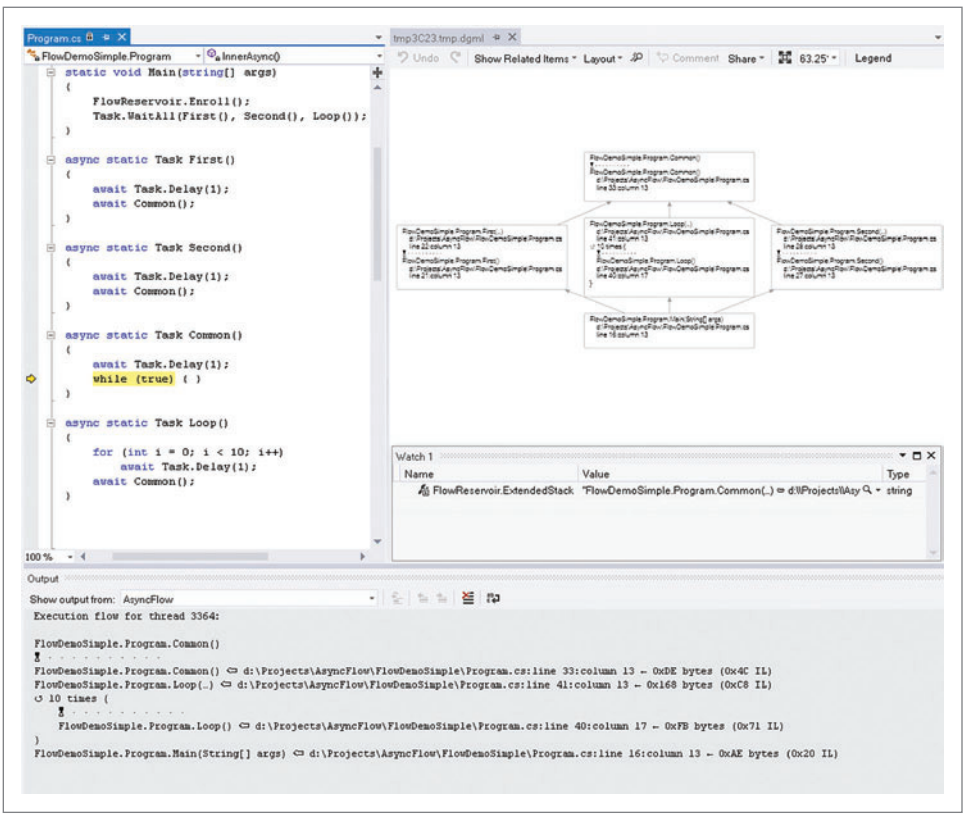


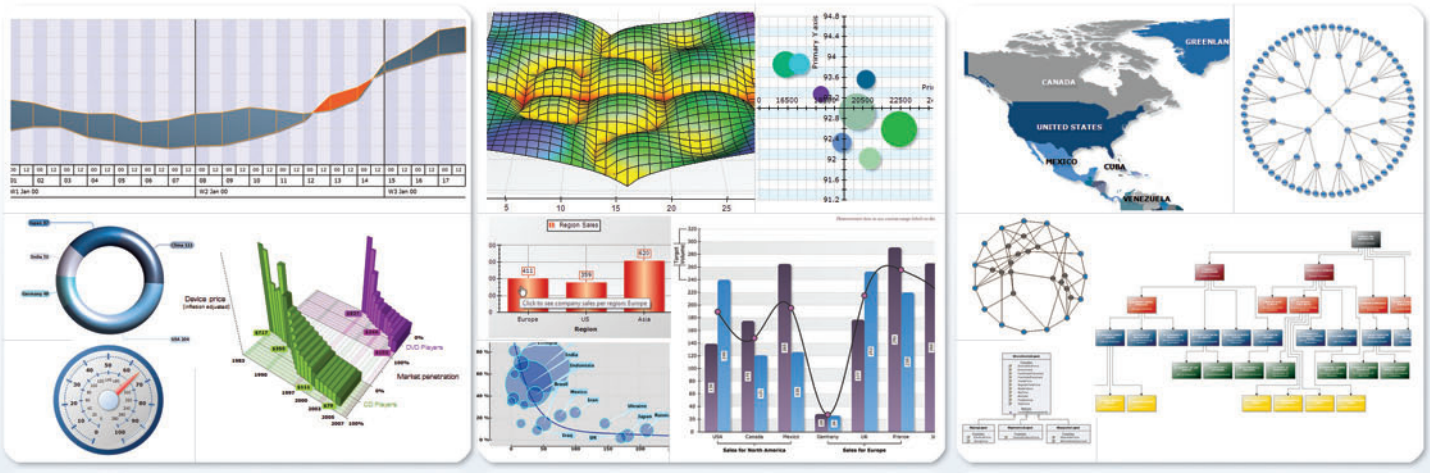
Figure 2 **Causality Chain for an Asynchronous Method and "Parallel" Causality for All Threads**

Andriy (Andrew) Stasyuk *is a software development engineer in test II on the Managed Languages team at Microsoft. He has seven years of experience as a participant, task author, jury member, and coach at various national and international programming contests. He worked in financial software development at Paladyne/Broadridge Financial Solutions Inc. and Deutsche Bank AG before moving to Microsoft. His main interests in programming are algorithms, parallelism and brainteasers.*

Thanks to the following technical experts for reviewing this article: *Vance Morrison and Lucian Wischik*

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 and 2010. 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 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 2005/2008/2012 reports and SharePoint 2007/2010 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.

Building a Simple Comet Application in the Microsoft .NET Framework

Derrick Lau

Comet is a technique for pushing content from a Web server to a browser without an explicit request, using long-lived AJAX connections. It allows for a more interactive UX and uses less bandwidth than the typical server round-trip triggered by a page postback to retrieve more data. Although there are plenty of Comet implementations available, most are Java-based. In this article I'll focus on building a C# service based on the cometbox code sample available at code.google.com/p/cometbox.

There are newer methods for implementing the same behavior using HTML5 features such as WebSockets and server-side events, but these are available only in the latest browser versions. If you must support older browsers, Comet is the most-compatible solution. However, the browser must support AJAX by implementing the xmlHttpRequest object; otherwise it won't be able to support Comet-style communication.

This article discusses:

- Combining .NET and Windows services technologies with AJAX to enable Comet-style communication
- Pitfalls in implementing Comet-style architecture
- Testing the application

Technologies discussed:

ASP.NET 4, Windows Services, C#, JavaScript, JSON, Microsoft .NET Framework

Code download available at:

archive.msdn.microsoft.com/mag201302Comet

The High-Level Architecture

Figure 1 shows basic Comet-style communication, while **Figure 2** depicts the architecture of my example. Comet uses the browser's xmlHttpRequest object, which is essential for AJAX communication, to establish a long-lived HTTP connection to a server. The server holds the connection open, and pushes content to the browser when available.

Between the browser and the server is a proxy page, which resides in the same Web application path as the Web page containing the client code and does nothing except forward the messages from browser to server and from server to browser. Why do you need a proxy page? I'll explain in a bit.

The first step is to select a format for the messages exchanged between the browser and server—JSON, XML or a custom format. For simplicity's sake, I picked JSON because it's naturally supported in JavaScript, jQuery and the Microsoft .NET Framework, and can transmit the same amount of data as XML using fewer bytes and, therefore, less bandwidth.

To set up Comet-style communication, you open an AJAX connection to the server. The easiest way to do this is to use jQuery because it supports multiple browsers and provides some nice wrapper functions such as $.ajax. This function is essentially a

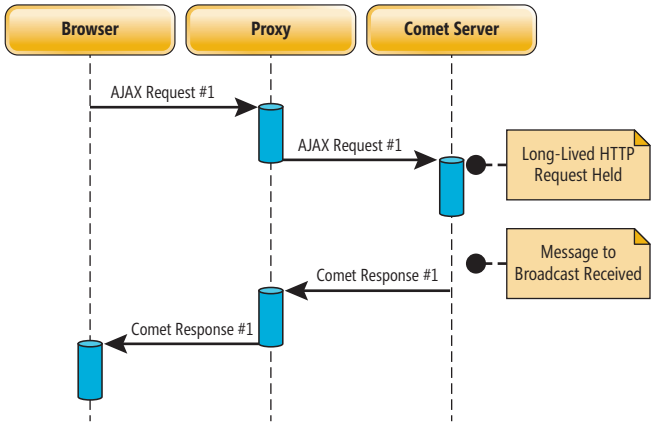


Figure 1 **Comet-Style Communication**

wrapper for each browser's xmlHttpRequest object, and neatly provides event handlers that can be implemented to process incoming messages from the server.

Before starting the connection, you instantiate the message to send. To do this, declare a variable and use JSON.stringify to format the data as a JSON message, as shown in **Figure 3**.

Next, initialize the function with the URL to connect to, the HTTP method of communication to use, the communication style and the connection timeout parameter. JQuery supplies this functionality in a library call named ajaxSetup. I set the timeout in this example to 10 minutes because I'm only building a proof of concept solution here; you can change the timeout setting to whatever you want.

Now open a connection to the server using the jQuery $.ajax method, with the definition of the success event handler as the only parameter:

```
$.ajax({
  success: function (msg) {
    // Alert("ajax.success().");
    if (msg == null || msg.Message == null) {
      getResponse();
      return;
    }
```

The handler tests the message object returned to ensure it contains valid information before parsing; this is necessary because if an error code is returned, jQuery will fail and display an undefined message to the user. Upon a null message, the handler should recursively call the AJAX function again and return; I've found that adding the return stops the code from continuing. If the message is OK, you simply read the message and write the contents to the page:

```
$("#_receivedMsgLabel").append(msg.Message + "<br/>");
getResponse();
return;
    }

  });
```

This creates a simple client that illustrates how Comet-style communication works, as well as providing a means for running performance and scalability tests. For my example, I put the getResponse JavaScript code in a Web user control and registered it in the codebehind so the AJAX connection opens immediately when the control is loaded onto the ASP.NET page:

```
public partial class JqueryJsonCometClientControl :
  System.Web.UI.UserControl
{
  protected void Page_Load(object sender, EventArgs e)
  {
    string getResponseScript =
      @"<script type=text/javascript>getResponse();</script>";
    Page.ClientScript.RegisterStartupScript(GetType(),
      "GetResponseKey", getResponseScript);
  }
}
```

The Server

Now that I have a client that can send and receive messages, I'll build a service that can receive and respond to them.

I tried implementing several different techniques for Comet-style communication, including the use of ASP.NET pages and HTTP handlers, none of which were successful. What I couldn't seem to do was get a single message to broadcast to multiple clients. Luckily, after a lot of research I stumbled across the cometbox project and found it to be the easiest approach. I did some tinkering to make it run as a Windows service so it would be easier to use, then gave it the ability to hold a long-lived connection and push content to the browser. (Unfortunately, in doing so, I wrecked some of the cross-platform compatibility.) Finally, I added support for JSON and my own HTTP content message types.

To get started, create a Windows service project in your Visual Studio solution and add a service installer component (you'll find the instructions at bit.ly/TrHQ80) so you can turn your service on and off in the Services applet of the Administrative Tools in Control Panel. Once this is done, you need to create two threads: one that will bind to the TCP port and receive as well as transmit messages; and one that will block on a message queue to ensure that content is transmitted only when a message is received.

First, you must create a class that listens on the TCP port for new messages and transmits the responses. Now, there are several styles of Comet communication that can be implemented, and in the implementation there's a Server class (see the code file Comet_Win_Service HTTP\Server.cs in the sample code) to abstract these. For simplicity's sake, however, I'll focus on what's required to do a very basic receive of a JSON message over HTTP, and to hold the connection until there's content to push back.

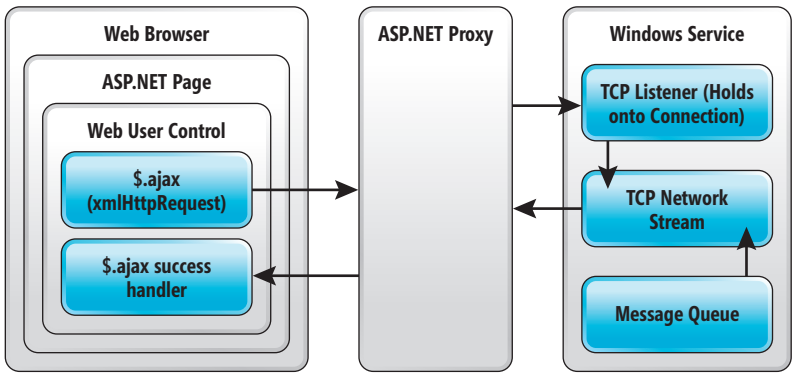


Figure 2 **Architecture of the Comet Application**

Figure 3 **Format the Data as a JSON Message**

```
function getResponse() {
  var currentDate = new Date();
  var sendMessage = JSON.stringify({
    SendTimestamp: currentDate,
    Message: "Message 1"
  });
  $.ajaxSetup({
    url: "CometProxy.aspx",
    type: "POST",
    async: true,
    global: true,
    timeout: 600000
  });
```

In the Server class, I'll create some protected members to hold objects I'll need to access from the Server object. These include the thread that will bind to and listen on the TCP port for HTTP connections, some semaphores and a list of client objects, each of which will represent a single connection to the server. Of importance is _isListenerShutDown, which will be exposed as a public property so it can be modified in the service Stop event.

Next, in the constructor, I'll instantiate the TCP Listener object against the port, set it for exclusive use of the port, and then start it. Then I'll start a thread to receive and handle clients that connect to the TCP listener.

Figure 4 **Listening for Client Connections**

```
private void Loop()
{
  try
  {
    while (true)
    {
      TcpClient client = null;
      bool isServerStopped = false;
      _listenerMutex.WaitOne();
      isServerStopped = _isListenerShutDown;
      _listenerMutex.ReleaseMutex();
      if (!isServerStopped)
      {
        client = listener.AcceptTcpClient();
      }
    else
    {
      continue;
    }

    Trace.WriteLineIf(_traceSwitch.TraceInfo, "TCP client accepted.",
      "COMET Server");

    bool addClientFlag = true;

    Client dc = new Client(client, this, authconfig, _currentClientId);
    _currentClientId++;
    foreach (Client currentClient in clients)
    {
      if (dc.TCPClient == currentClient.TCPClient)
      {
        lock (_lockObj)
        {
          addClientFlag = false;
        }
      }

    }

    if (addClientFlag)
    {
      lock (_lockObj)
      {
        clients.Add(dc);
      }
    }
```

The thread that listens for client connections contains a while loop that continually resets a flag indicating whether the service Stop event was raised (see **Figure 4**). I set the first part of this loop to a mutex to block on all listening threads to check whether the service Stop event was raised. If so, the _isListenerShutDown property will be true. When the check completes, the mutex is released and if the service is still running, I call the TcpListener.AcceptTcpClient, which will return a TcpClient object. Optionally, I check existing TcpClients to ensure I don't add an existing client. However, depending on the number of clients you expect, you might want to replace this with a system where the service generates a unique ID and sends it to the browser client, which remembers and resends the ID each time it communicates with the server to ensure it holds only a single connection. This can become problematic, though, if the service fails; it resets the ID counter and could give new clients already-used IDs.

Finally, the thread goes through the list of clients and removes any that are no longer alive. For simplicity, I put this code in the method that's called when the TCP listener accepts a client connection, but this can affect performance when the number of clients gets into the hundreds of thousands. If you intend on using this in public-facing Web applications, I suggest adding a timer that fires every so often and doing the cleanup in that.

When a TcpClient object is returned in the Server class Loop method, it's used to create a client object that represents the browser client. Because each client object is created in a unique thread, as with the server constructor, the client class constructor must wait on a mutex to ensure the client hasn't been closed before continuing. Afterward, I check the TCP stream and begin reading it, and initiate a callback handler to be executed once the read has been completed. In the callback handler, I simply read the bytes and parse them using the ParseInput method, which you can see in the sample code provided with this article.

Figure 5 **The Default XML Message Handler**

```
if (request.Headers["Content-Type"].Contains("xml"))
{
  Trace.WriteLineIf(_traceSwitch.TraceVerbose, "Received XML content from client.");
  _messageFormat = MessageFormat.xml;

  #region Process HTTP message as XML

  try
  {
    // Picks up message from HTTP
    XmlSerializer s = new XmlSerializer(typeof(Derrick.Web.SIServer.SIRequest));

    // Loads message into object for processing
    Derrick.Web.SIServer.SIRequest data =
      (Derrick.Web.SIServer.SIRequest)s.Deserialize(mem);
  }
  catch (Exception ex)
  {
    Trace.WriteLineIf(_traceSwitch.TraceVerbose,
      "During parse of client XML request got this exception: " + ex.ToString());
  }

  #endregion Process HTTP message as XML

}
```

Does your Team do more than just track bugs?

Free Trial and Single User FreePack™ available at www.alexcorp.com

Alexsys Team® does! Alexsys Team 2 is a multi-user Team management system that provides a powerful yet easy way to manage all the members of your team and their tasks - including defect tracking. Use Team right out of the box or tailor it to your needs.

Track all your project tasks in one database so you can work together to get projects done.

- Quality Control / Compliance Tracking
- Project Management
- End User Accessible Service Desk Portal
- Bugs and Features
- Action Items
- Sales and Marketing
- Help Desk

Native Smart Card Login Support including Government and DOD

New in Team 2.11

- Full Windows 7 Support
- Windows Single Sign-on
- System Audit Log
- Trend Analysis
- Alternate Display Fields for Data Normalization
- Lookup Table Filters
- XML Export
- Network Optimized for Enterprise Deployment

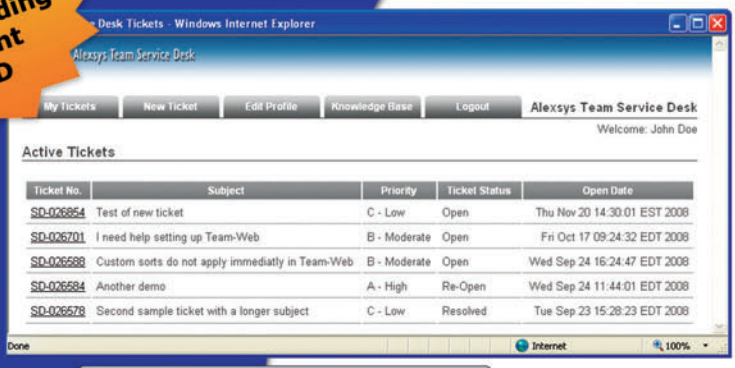
Service Desk Features

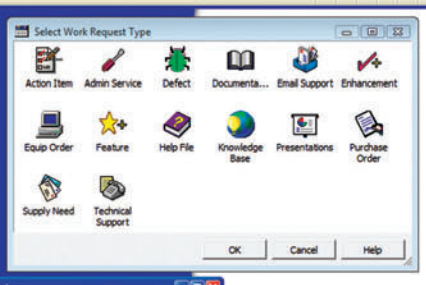
- Fully Secure
- Unlimited Users Self Registered or Active Directory
- Integrated into Your Web Site
- Fast/AJAX Dynamic Content
- Unlimited Service Desks
- Visual Service Desk Builder

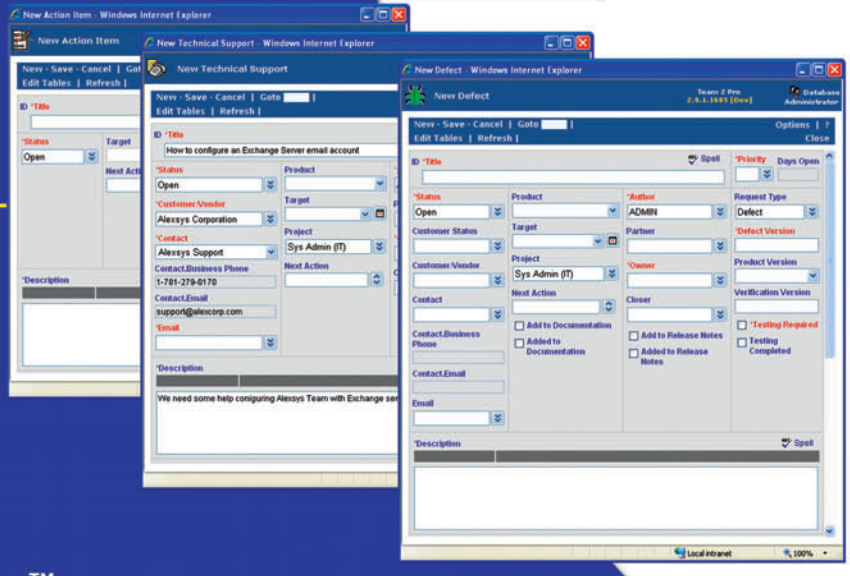
Team 2 Features

- Windows and Web Clients
- Multiple Work Request Forms
- Customizable Database
- Point and Click Workflows
- Role Based Security
- Clear Text Database
- Project Trees
- Time Recording
- Notifications and Escalations
- Outlook Integration

ZDNet Editors' Pick







Free Trial and Single User FreePack™ available at www.alexcorp.com.
FreePack™ includes a free single user Team Pro and Team-Web license.
Need more help? Give us a call at 1-888-880-ALEX (2539).

Team 2 works with its own standard database, while Team Pro works with Microsoft SQL, MySQL, and Oracle Servers.
Team 2 works with Windows 7/2008/2003/Vista/XP.
Team-Web works with Internet Explorer, Firefox, Netscape, Safari, and Chrome.

Figure 6 **Deserializing a JSON Message**

```
else if (request.Headers["Content-Type"].Contains("json"))
{
  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Received json content from client.");
  _messageFormat = MessageFormat.json;

  #region Process HTTP message as JSON

  try
  {
    JavaScriptSerializer jsonSerializer = new JavaScriptSerializer();
    ClientMessage3 clientMessage =
      jsonSerializer.Deserialize<ClientMessage3>(request.Body);
    _lastUpdate = clientMessage.SendTimestamp;
    Trace.WriteLineIf(_traceSwitch.TraceVerbose,
      "Received the following message: ");
    Trace.WriteLineIf(_traceSwitch.TraceVerbose, "SendTimestamp: " +
      clientMessage.SendTimestamp.ToString());
    Trace.WriteLineIf(_traceSwitch.TraceVerbose, "Browser: " +
      clientMessage.Browser);
    Trace.WriteLineIf(_traceSwitch.TraceVerbose, "Message: " +
      clientMessage.Message);


  }
  catch (Exception ex)
  {
    Trace.WriteLineIf(_traceSwitch.TraceVerbose,
      "Error deserializing JSON message: " + ex.ToString());
  }


  #endregion Process HTTP message as JSON

}
```

In the ParseInput method of the Client class, I build a Request object with members that correspond to the different parts of the typical HTTP message and populate those members appropriately. First, I parse the header information by searching for the token characters, such as "\r\n," determining the pieces of header information from the format of the HTTP header. Then I call the ParseRequestContent method to get the body of the HTTP message. The first step of ParseInput is to determine the method of HTTP communication used and the URL the request was sent to. Next, the HTTP message headers are extracted and stored in the Request object Headers property, which is a Dictionary of header types and values. Once again, take a look at the downloadable sample code to see how this is done. Finally, I load the contents of the request into the Request object's Body property, which is just a String variable containing all the bytes of the content. The content has yet to be parsed at this point. At the end, if there are any problems with the HTTP request received from the client, I send out an appropriate error response message.

Figure 7 **Content-Type "Ping"**

```
else if (request.Headers["Content-Type"].Contains("ping"))
{
  string msg = request.Body;
  Trace.WriteLineIf(_traceSwitch.TraceVerbose, "Ping received.");
  if (msg.Equals("PING"))
  {
    SendMessageEventArgs args = new SendMessageEventArgs();
    args.Client = this;
    args.Message = "PING";
    args.Request = request;
    args.Timestamp = DateTime.Now;
    SendResponse(args);
  }

}
```

I separated the method for parsing the HTTP request's content so I could add in support for different message types, such as plain text, XML, JSON and so forth:

```
public void ParseRequestContent()
{

  if (String.IsNullOrEmpty(request.Body))
  {
    Trace.WriteLineIf(_traceSwitch.TraceVerbose,
      "No content in the body of the request!");
    return;
  }

  try
  {
```

First the contents are written to a MemoryStream so, if necessary, they can be deserialized into object types depending on the request's Content-Type, as certain deserializers only work with streams:

```
MemoryStream mem = new MemoryStream();
mem.Write(System.Text.Encoding.ASCII.GetBytes(request.Body), 0,
  request.Body.Length);
mem.Seek(0, 0);

if (!request.Headers.ContainsKey("Content-Type"))
{
  _lastUpdate = DateTime.Now;
  _messageFormat = MessageFormat.json;
}
else
{
```

As shown in **Figure 5**, I kept the default action of handling XML-formatted messages because XML is still a popular format.

For Web applications, however, I highly recommend formatting the messages in JSON as, unlike XML, it doesn't have the overhead of beginning and cancel tags and it's natively supported in JavaScript. I just use the Content-Type header of the HTTP request to indicate whether the message was sent in JSON, and deserialize the contents using the System.Web.Script.Serialization namespace JavaScriptSerializer class. This class makes it very easy to deserialize a JSON message into a C# object, as shown in **Figure 6**.

Finally, for testing purposes I added a ping Content-Type that simply responds with a text HTTP response containing only the word PING. This way I can easily test to see if my Comet server is running by sending it a JSON message with Content-Type "ping," as shown in **Figure 7**.

Ultimately, ParseRequestContent is just a string parsing method—nothing more, nothing less. As you can see, parsing XML data is a little more involved because the content has to be written to a MemoryStream first and then deserialized, using the XmlSerializer class, into a class created to represent the message from the client.

To better organize the source code, I create a Request class, shown in **Figure 8**, that simply contains members to hold the headers and other information sent in the HTTP request in a manner easily accessible within the service. If you wish, you can add helper methods to determine if the request has any content or not, and authentication checks, too. However, I didn't do this here to keep this service simple and easy to implement.

The Response class, like the Request class, contains methods to store the HTTP response information in a manner easily accessible by a C# Windows service. In the SendResponse method, I added logic to attach custom HTTP headers as required for cross-origin resource sharing (CORS), and had those headers loaded from a

componentart.com/windows8

Figure 8 **The Request Class**

```
public class Request
{
  public string Method;
  public string Url;
  public string Version;
  public string Body;
  public int ContentLength;
  public Dictionary<string, string> Headers = new Dictionary<string, string>();

  public bool HasContent()
  {
    if (Headers.ContainsKey("Content-Length"))
    {
      ContentLength = int.Parse(Headers["Content-Length"]);
      return true;
    }
    return false;
  }
```

configuration file so they can be easily modified. The Response class also contains methods to output messages for some common HTTP statuses, such as 200, 401, 404, 405 and 500.

The SendResponse member of the Response class simply writes the message to the HTTP response stream that should still be alive, as the timeout set by the client is quite long (10 minutes):

```
public void SendResponse(NetworkStream stream, Client client)
{
```

As shown in **Figure 9**, the appropriate headers are added to the HTTP response to fit with the W3C specification for CORS. For simplicity, the headers are read from the configuration file so the header contents can be easily modified.

Now I add the regular HTTP response headers and content, as shown in **Figure 10**.

Here the entire HTTP response message, which was built as a String, is now written to the HTTP response stream, which was passed in as a parameter to the SendResponse method:

```
byte[] htext = Encoding.ASCII.GetBytes(r.ToString());
stream.Write(htext, 0, htext.Length);
```

Transmitting Messages

The thread to transmit messages is essentially nothing more than a While loop that blocks on a Microsoft message queue. It has a SendMessage event that's raised when the thread picks up a message from the queue. The event is handled by a method in the server object that basically calls the SendResponse method of each client, thus broadcasting the message to every browser connected to it.

The thread waits on the appropriate message queue until there's a message placed on it, indicating the server has some content it wishes to broadcast to the clients:

```
Message msg = _intranetBannerQueue.Receive();
// Holds thread until message received
Trace.WriteLineIf(_traceSwitch.TraceInfo,
  "Message retrieved from the message queue.");

SendMessageEventArgs args = new SendMessageEventArgs();
args.Timestamp = DateTime.Now.ToUniversalTime();
```

When the message is received, it's converted into the expected object type:

```
msg.Formatter = new XmlMessageFormatter(new Type[] { typeof(string) });
string cometMsg = msg.Body.ToString();
args.Message = cometMsg;
```

After determining what will be sent to the clients, I raise a Windows event on the server indicating there's a message to be broadcast:

```
if (SendMessageEvent != null)
{
  SendMessageEvent(this, args);
  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Message loop raised SendMessage event.");
}
```

Next, I need a method that will build the actual HTTP response body—the contents of the message the server will broadcast to all the clients. The preceding message takes the message contents dumped onto the Microsoft message queue and formats it as a JSON object for transmission to the clients via an HTTP response message, as shown in **Figure 11**.

Next, I need to instantiate an instance of the JavaScriptSerializer object to put the message contents into JSON format. I add the following try/catch error handling because sometimes there are difficulties instantiating an instance of a JavaScriptSerializer object:

```
try
{
  jsonSerializer = new JavaScriptSerializer();
}
catch (Exception ex)
{
  errorInSendResponse = true;
  Trace.WriteLine("Cannot instantiate JSON serializer: " + ex.ToString());
}
```

Figure 9 **Adding the CORS Headers**

```
if (client.Request.Headers.ContainsKey("Origin"))
{
  AddHeader("Access-Control-Allow-Origin", client.Request.Headers["Origin"]);
  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Access-Control-Allow-Origin from client: " +
    client.Request.Headers["Origin"]);
}
else
{
  AddHeader("Access-Control-Allow-Origin",
    ConfigurationManager.AppSettings["RequestOriginUrl"]);
  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Access-Control-Allow-Origin from config: " +
    ConfigurationManager.AppSettings["RequestOriginUrl"]);
}

AddHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS");
AddHeader("Access-Control-Max-Age", "1000");
// AddHeader("Access-Control-Allow-Headers", "Content-Type");
string allowHeaders = ConfigurationManager.AppSettings["AllowHeaders"];
// AddHeader("Access-Control-Allow-Headers", "Content-Type, x-requested-with");
AddHeader("Access-Control-Allow-Headers", allowHeaders);

StringBuilder r = new StringBuilder();
```

Figure 10 **Adding the Regular HTTP Response Headers**

```
r.Append("HTTP/1.1 " + GetStatusString(Status) + "\r\n");
r.Append("Server: Derrick Comet\r\n");
r.Append("Date: " + DateTime.Now.ToUniversalTime().ToString(
  "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'") + "\r\n");
r.Append("Accept-Ranges: none\r\n");

foreach (KeyValuePair<string, string> header in Headers)
{
  r.Append(header.Key + ": " + header.Value + "\r\n");
}

if (File != null)
{
  r.Append("Content-Type: " + Mime + "\r\n");
  r.Append("Content-Length: " + File.Length + "\r\n");
}
else if (Body.Length > 0)
{
  r.Append("Content-Type: " + Mime + "\r\n");
  r.Append("Content-Length: " + Body.Length + "\r\n");
}
r.Append("\r\n");
```

Figure 11 **Building the HTTP Response Body**

```
public void SendResponse(SendMessageEventArgs args)
{

  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Client.SendResponse(args) called...");

  if (args == null || args.Timestamp == null)
  {
    return;
  }
  if (_lastUpdate > args.Timestamp)
  {
    return;
  }

  bool errorInSendResponse = false;

  JavaScriptSerializer jsonSerializer = null;
```

Then I create a string variable to hold the JSON-formatted message and an instance of the Response class to send the JSON message.

I immediately do some basic error checking to make sure I'm working with a valid HTTP request. Because this Comet service spawns a thread for each TCP client, as well as for the server objects, I felt it safest to include these safety checks every so often, to make debugging easier.

Once I verify that it's a valid request, I put together a JSON message to send to the HTTP response stream. Note that I just create the JSON message, serialize it and use it to create an HTML response message:

```
if (request.HasContent())
{

  if (_messageFormat == MessageFormat.json)
  {
    ClientMessage3 jsonObjectToSend = new ClientMessage3();
    jsonObjectToSend.SendTimestamp = args.Timestamp;
    jsonObjectToSend.Message = args.Message;
    jsonMessageToSend = jsonSerializer.Serialize(jsonObjectToSend);
    response = Response.GetHtmlResponse(jsonMessageToSend,
      args.Timestamp, _messageFormat);
    response.SendResponse(stream, this);
  }
```

To hook it all together, I first create instances of the message loop object and the server loop object during the service Start event. Note that these objects should be protected members of the service class so that methods on them can be called during other service events. Now the message loop send message event should be handled by the server object BroadcastMessage method:

```
public override void BroadcastMessage(Object sender, SendMessageEventArgs args)
{
  // Throw new NotImplementedException();
  Trace.WriteLineIf(_traceSwitch.TraceVerbose,
    "Broadcasting message [" + args.Message + "] to all clients.");

  int numOfClients = clients.Count;
  for (int i = 0; i < numOfClients; i++)
  {
    clients[i].SendResponse(args);
  }
}
```

The BroadcastMessage just sends the same message to all clients. If you wish, you can modify it to send the message only to the clients you want; in this way you can use this service to handle, for instance, multiple online chat rooms.

The OnStop method is called when the service is stopped. It subsequently calls the Shutdown method of the server object, which goes through the list of client objects that are still valid and shuts them down.

At this point, I have a reasonably decent working Comet service, which I can install into the services applet from the command prompt using the installutil command (for more information, see bit.ly/OtQCB7). You could also create your own Windows installer to deploy it, as you've already added the service installer components to the service project.

Why Doesn't It Work? The Problem with CORS

Now, try setting the URL in the $.ajax call of the browser client to point to the Comet service URL. Start the Comet service and open the browser client in Firefox. Make sure you have the Firebug extension installed in the Firefox browser. Start Firebug and refresh the page; you'll notice you get an error in the console output area stating "Access denied." This is due to CORS, where for security reasons, JavaScript can't access resources outside the same Web application and virtual directory its housing page resides in. For example, if your browser client page is in http://www.somedomain.com/somedir1/somedir2/client.aspx, then any AJAX call made on that page can go only to resources in the same virtual directory or a subdirectory. This is great if you're calling another page or HTTP handler within the Web application, but you don't want pages and handlers to block on a message queue when transmitting the same message to all clients, so you need to use the Windows Comet service and you need a way of getting around the CORS restriction.

Figure 12 **Writing to the HttpWebRequest Stream**

```
Stream stream = null;

if (cometRequest.ContentLength > 0 && !cometRequest.Method.Equals("OPTIONS"))
{
  stream = cometRequest.GetRequestStream();
  stream.Write(bytes, 0, bytes.Length);
}

if (stream != null)
{
  stream.Close();
}

// Console.WriteLine(System.Text.Encoding.ASCII.GetString(bytes));
System.Diagnostics.Trace.WriteLineIf(_proxySwitch.TraceVerbose,
"Forwarding message: " + System.Text.Encoding.ASCII.GetString(bytes));
```

Figure 13 **Writing the Server Message to the HTTP Response Stream**

```
string msgSizeStr = ConfigurationManager.AppSettings["MessageSize"];
int messageSize = Convert.ToInt32(msgSizeStr);

byte[] read = new byte[messageSize];
// Reads 256 characters at a time
int count = s.Read(read, 0, messageSize);
while (count > 0)
{
  // Dumps the 256 characters on a string and displays the string to the console
  byte[] actualBytes = new byte[count];
  Array.Copy(read, actualBytes, count);
  string cometResponseStream = Encoding.ASCII.GetString(actualBytes);
  Response.Write(cometResponseStream);
  count = s.Read(read, 0, messageSize);
}

Response.End();
System.Diagnostics.Trace.WriteLineIf(_proxySwitch.TraceVerbose, "Sent Message.");

s.Close();
}
```

Figure 14 **Varying the Number of Users**

Users	Repetitions	Message Size (in Bytes)	Response Time (in Milliseconds)
1,000	10	512	2.56
5,000	10	512	4.404
10,000	10	512	18.406
15,000	10	512	26.368
20,000	10	512	36.612
25,000	10	512	48.674
30,000	10	512	64.016
35,000	10	512	79.972
40,000	10	512	99.49
45,000	10	512	122.777
50,000	10	512	137.434

To do this, I recommend building a proxy page in the same virtual directory, whose only function is to intercept the HTTP message from the browser client, extract all the relevant headers and content, and build another HTTP request object that connects to the Comet service. Because this connection is done on the server, it isn't impacted by CORS. Thus, through a proxy, you can keep a long-lived connection between your browser client and the Comet service. Moreover, you can now transmit a single message when it arrives on a message queue to all connected browser clients simultaneously.

First, I take the HTTP request and stream it into an array of bytes so I can pass it to a new HTTP request object that I'll instantiate shortly:

```
byte[] bytes;
using (Stream reader = Request.GetBufferlessInputStream())
{
  bytes = new byte[reader.Length];
  reader.Read(bytes, 0, (int)reader.Length);
}
```

Next, I create a new HttpWebRequest object and point it to the Comet server, whose URL I put in the web.config file so it can be easily modified later:

```
string newUrl = ConfigurationManager.AppSettings["CometServer"];
HttpWebRequest cometRequest = (HttpWebRequest)HttpWebRequest.Create(newUrl);
```

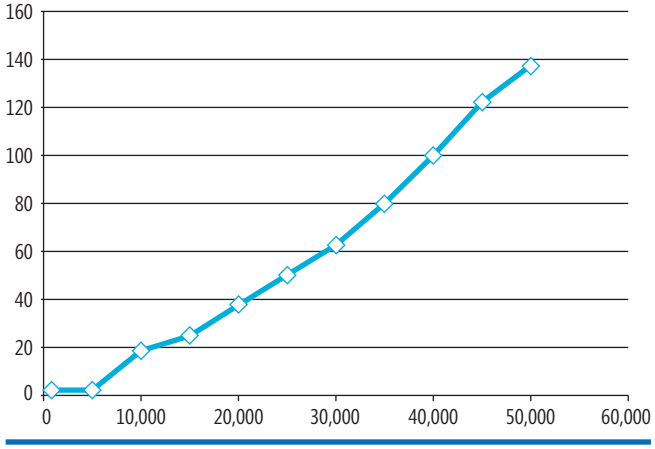


Figure 15 **Response Times for Varying Numbers of Users for a 512-Byte Message**

This creates a connection to the Comet server for each user, but since the same message is being broadcast to each user, you can just encapsulate the cometRequest object in a double locking singleton to reduce the connection load on the Comet server, and let IIS do the connection load balancing for you.

Then I populate the HttpWebRequest headers with the same values I received from the jQuery client, especially setting the KeepAlive property to true so I maintain a long-lived HTTP connection, which is the fundamental technique behind Comet-style communication.

Here I check for an Origin header, which is required by the W3C specification when dealing with CORS-related issues:

```
for (int i = 0; i < Request.Headers.Count; i++)
{
  if (Request.Headers.GetKey(i).Equals("Origin"))
  {
    containsOriginHeader = true;
    break;
  }
}
```

I then pass the Origin header on to the HttpWebRequest so the Comet server will receive it:

```
if (containsOriginHeader)
{
  // cometRequest.Headers["Origin"] = Request.Headers["Origin"];
  cometRequest.Headers.Set("Origin", Request.Headers["Origin"]);
}
else
{
  cometRequest.Headers.Add("Origin", Request.Url.AbsoluteUri);
}

System.Diagnostics.Trace.WriteLineIf(_proxySwitch.TraceVerbose,
  "Adding Origin header.");
```

Next, I take the bytes from the content of the HTTP request from the jQuery client and write them to the request stream of the HttpWebRequest, which will be sent to the Comet server, as shown in **Figure 12**.

After forwarding the message to the Comet server, I call the GetResponse method of the HttpWebRequest object, which provides an HttpWebResponse object that allows me to process the server's response. I also add the required HTTP headers that I'll send with the message back to the client:

```
try
{
  Response.ClearHeaders();
  HttpWebResponse res = (HttpWebResponse)cometRequest.GetResponse();
  for (int i = 0; i < res.Headers.Count; i++)
  {
    string headerName = res.Headers.GetKey(i);
    // Response.Headers.Set(headerName, res.Headers[headerName]);
    Response.AddHeader(headerName, res.Headers[headerName]);
  }

  System.Diagnostics.Trace.WriteLineIf(_proxySwitch.TraceVerbose,
    "Added headers.");
```

I then wait for the server's response:

```
Stream s = res.GetResponseStream();
```

When I receive the Comet server's message, I write it to the original HTTP request's response stream so the client can receive it, as shown in **Figure 13**.

Test the Application

To test your application, create a Web site to hold the sample application pages. Make sure the URL to your Windows service is correct and the message queue is properly configured and useable. Start the service and open the Comet client page in one browser

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 +1 877-462-4772
EU +49 421-4270671-0

WWW.TEXTCONTROL.COM

Figure 16 **Testing with a Message Size of 1,024 Bytes**

Users	Repetitions	Response Time (in Milliseconds)
1,000	10	144.227
5,000	10	169.648
10,000	10	233.031
15,000	10	272.919
20,000	10	279.701
25,000	10	220.209
30,000	10	271.799
35,000	10	230.114
40,000	10	381.29
45,000	10	344.129
50,000	10	342.452

and the page to send messages in another. Type in a message and press send; after roughly 10 ms you should see the message appear in the other browser window. Try this with various browsers—especially some of the older ones. As long as they support the xmlHttpRequest object, it should work. This provides almost real-time Web behavior (en.wikipedia.org/wiki/Real-time_web), where content is pushed to the browser almost instantaneously without requiring action from the user.

Before any new application is deployed, you have to do performance and load testing. To do this, you should first identify the metrics you want to gather. I suggest measuring usage load against both response times and data-transfer size. Additionally, you should test usage scenarios that are relevant to Comet, in particular broadcasting a single message to multiple clients without postback.

To do the testing, I constructed a utility that opens multiple threads, each with a connection to the Comet server, and waits until the server fires a response. This test utility allows me to set a few parameters, such as the total number of users that will connect to my Comet server and the number of times they reopen the connection (currently the connection is closed after the server's response is sent).

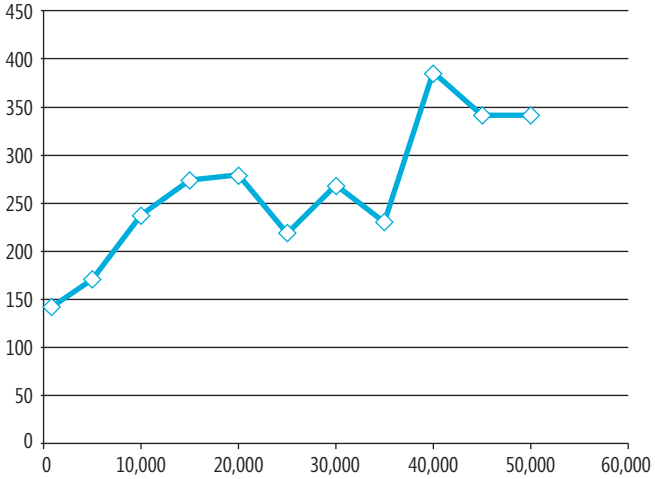


Figure 17 **User Load vs Response Time for a 1KB Message**

I then created a utility that dumps a message of x number of bytes to the message queue, with the number of bytes set by a text field on the main screen, and a text field to set the number of milliseconds to wait between messages sent from the server. I'll use this to send the test message back to the client. I then started the test client, specified the number of users plus the number of times the client will reopen the Comet connection, and the threads opened the connections against my server. I waited a few seconds for all the connections to be opened, then went to the message-sending utility and submitted a certain number of bytes. I repeated this for various combinations of total users, total repetitions and message sizes.

The first data sampling I took was for a single user with increasing repetitions but with the response message a consistent (small) size throughout the testing. As you can see in **Figure 14**, the number of repetitions doesn't seem to have an impact on system performance or reliability.

The times are gradually increasing in a linear/constant manner, which means the code on the Comet server is generally robust. **Figure 15** graphs the number of users against the response time for a 512-byte message. **Figure 16** shows some statistics for a message size of 1,024 bytes. Finally, **Figure 17** shows the chart from **Figure 16** in graphical format.All of these tests were done on a single laptop with 8GB of RAM and a 2.4 GHz Intel Core i3 CPU.

The numbers don't show any particular trend, except that response times are reasonable, remaining at below one second for message sizes up to 1KB. I didn't bother tracking bandwidth use because that's affected by the message format. Also, because all testing was done on a single computer, network latency was eliminated as a factor. I could've tried it against my home network, but I didn't think it would be worthwhile because the public Internet is far more complex than my wireless router and cable modem setup. However, because the key point of Comet communication techniques is to reduce server round-trips by pushing content from the server as updated, theoretically half the network bandwidth usage should be reduced through Comet techniques.

Wrapping Up

I hope you can now successfully implement your own Comet-style applications and use them effectively to reduce network bandwidth and increase Web site application performance. Of course, you'll want to check out the new technologies included with HTML5, which can replace Comet, such as WebSockets (bit.ly/UVMcBg) and Server-Sent Events (SSE) (bit.ly/UVMhoD). These technologies hold the promise of providing a simpler way of pushing content to the browser, but they do require the user to have a browser that supports HTML5. If you still have to support users on older browsers, Comet-style communication remains the best choice. ■

Derrick Lau *is an experienced software development team leader with approximately 15 years of relevant experience. He has worked in the IT shops of financial firms and the government, as well as in the software development sections of technology-focused companies. He won the grand prize in an EMC development contest in 2010 and came in as a finalist in 2011. He is also certified as an MCSD and as an EMC content management developer.*

Thanks to the following technical expert for reviewing this article: *Francis Cheung*

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.

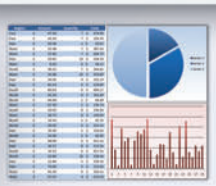
Excel Reporting for ASP.NET, WinForms, WPF and Silverlight

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

Detecting Abnormal Data Using k-Means Clustering

James McCaffrey

Consider the problem of identifying abnormal data items in a very large data set, for example, identifying potentially fraudulent credit-card transactions, risky loan applications and so on. One approach to detecting abnormal data is to group the data items into similar clusters and then seek data items within each cluster that are different in some sense from other data items within the cluster.

There are many different clustering algorithms. One of the oldest and most widely used is the k-means algorithm. In this article I'll explain how the k-means algorithm works and present a complete C# demo program. There are many existing standalone data-clustering tools, so why would you want to create k-means clustering code from scratch? Existing clustering tools can be difficult or impossible to integrate into a software system, they might not be customizable to deal with unusual scenarios, and the tools might have copyright or other intellectual property issues.

This article discusses:

- The k-means clustering algorithm
- Computing cluster centroids
- Euclidian distance
- Looking for abnormal data

Technologies discussed:

C#, Visual Studio 2010

Code download available at:

archive.msdn.microsoft.com/mag201302kmeans

After reading this article you'll be able to experiment with k-means clustering and have the base knowledge to add clustering functionality to a .NET application.

The best way to get a feel for what k-means clustering is and to see where I'm headed in this article is to take a look at **Figure 1**. The demo program begins by creating a dummy set of 20 data items. In clustering terminology, data items are sometimes called tuples. Each tuple here represents a person and has two numeric attribute values, a height in inches and a weight in pounds. One of the limitations of the k-means algorithm is that it applies only in cases where the data tuples are completely numeric.

The dummy data is loaded into an array in memory. Next, the number of clusters is set to three. Although there are advanced clustering techniques that can suggest the optimal number of clusters to use, in general data clustering is an exploratory process and the best number of clusters to use is typically found through trial and error. As you'll see shortly, k-means clustering is an iterative process. The demo program has a variable maxCount, which is used to limit the number of times the main clustering loop will execute. Here that value is arbitrarily set to 30.

Next, behind the scenes, the demo program uses the k-means algorithm to place each data tuple into one of three clusters. There are many ways to encode a clustering. In this case, a clustering is defined by an array of int where the array index represents a tuple, and the associated array value represents the 0-based cluster ID. So, in **Figure 1**, tuple 0 (65.0, 220.0) is assigned to cluster 0, tuple 1 (73.0, 160.0) is assigned to cluster 1, tuple 2 (59.0, 110.0) is assigned to cluster 2, tuple 3 (61.0, 120.0) is assigned to cluster 2 and so on.

Notice there are eight tuples assigned to cluster 0, five tuples assigned to cluster 1, and seven tuples assigned to cluster 2.

Next, the demo program displays the data, grouped by cluster. If you examine the clustered data you'll see that cluster 0 might be called the heavy people cluster, cluster 1 might be called the tall people cluster, and cluster 2 might be called the short people cluster. The demo program concludes by analyzing the tuples assigned to cluster 0 and determines that by some criterion, tuple 5 (67.0, 240.0) is the most abnormal tuple.

In the sections that follow, I'll walk you through the code that produced the screenshot in **Figure 1** so that you'll be able to modify this code to meet your own needs. This article assumes you have at least intermediate-level programming skill with a C-family language, but does not assume you know anything about data clustering. I coded the demo program using C#, but I used a non-OOP style so you shouldn't have too much difficulty refactoring the demo to another language if you wish. I present all the source code for the demo program in this article. The source code is also available at archive.msdn.microsoft.com/mag201302kmeans.

The k-Means Algorithm

In principle, at least, the k-means algorithm is quite simple. But as you'll see, some of the implementation details are a bit tricky. The central concept in the k-means algorithm is the centroid. In data clustering, the centroid of a set of data tuples is the one tuple that's most representative of the group. The idea is best explained by example. Suppose you have three height-weight tuples similar to those shown in **Figure 1**:

```
[a] (61.0, 100.0)
[b] (64.0, 150.0)
[c] (70.0, 140.0)
```

Which tuple is most representative? One approach is to compute a mathematical average (mean) tuple, and then select as the centroid the tuple that is closest to that average tuple. So, in this case, the average tuple is:

```
[m] = ((61.0 + 64.0 + 70.0) / 3, (100.0 + 150.0 + 140.0) / 3)
    = (195.0 / 3, 390.0 / 3)
    = (65.0, 130.0)
```

And now, which of the three tuples is closest to (65.0, 130.0)? There are several ways to define closest. The most common approach, and the one used in the demo program, is to use the Euclidean distance. In words, the Euclidean distance between two tuples is the square root of the sum of the squared differences between each component of the tuples. Again, an example is the best way to explain. The Euclidean distance between tuple (61.0, 100.0) and the average tuple (65.0, 130.0) is:

```
dist(m,a) = sqrt((65.0 - 61.0)^2 + (130.0 - 100.0)^2)
          = sqrt(4.0^2 + 30.0^2)
          = sqrt(16.0 + 900.0)
          = sqrt(916.0)
          = 30.27
```

Similarly:

```
dist(m,b) = sqrt((65.0 - 64.0)^2 + (130.0 - 150.0)^2)
          = 20.02

dist(m,c) = sqrt((65.0 - 70.0)^2 + (130.0 - 140.0)^2)
          = 11.18
```

Because the smallest of the three distances is the distance between the math average and tuple [c], the centroid of the three tuples is tuple [c]. You might wish to experiment with the demo program by using different definitions of the distance between two tuples to see how those affect the final clustering produced.

With the notion of a cluster centroid established, the k-means algorithm is relatively simple. In pseudo-code:

```
assign each tuple to a randomly selected cluster
compute the centroid for each cluster
loop until no improvement or until maxCount
  assign each tuple to best cluster
   (the cluster with closest centroid to tuple)
  update each cluster centroid
   (based on new cluster assignments)
end loop
return clustering
```

If you search the Web, you can find several good online animations of the k-means algorithm in action. The image in **Figure 2**

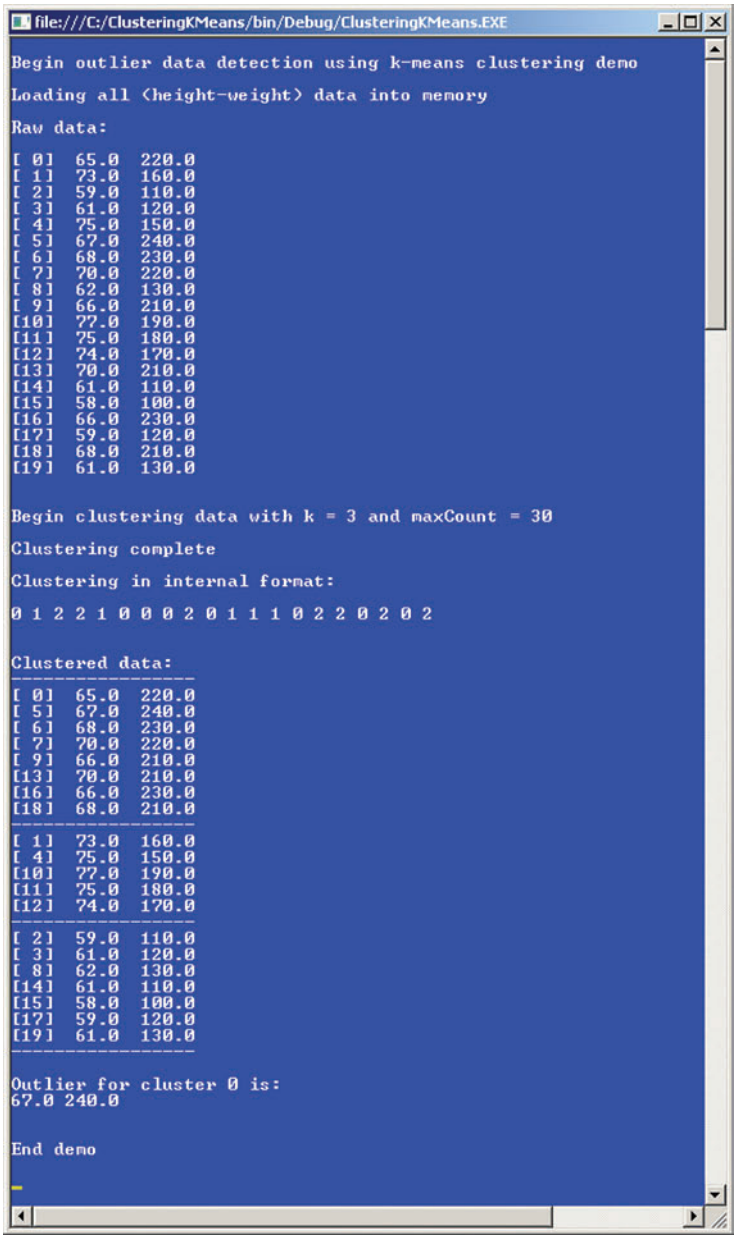


Figure 1 **Clustering Using k-Means**

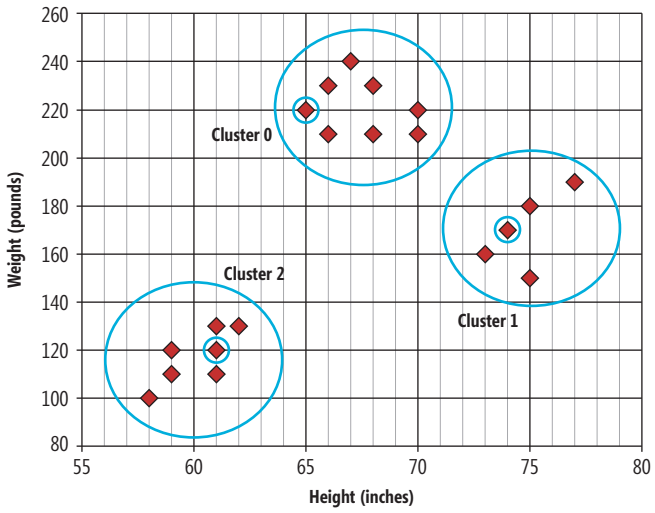


Figure 2 **Clustered Data and Centroids**

shows the clustering produced by the demo program. The circled data item in each cluster is the cluster centroid.

Overall Program Structure

The overall program structure for the demo shown in **Figure 1**, with a few minor edits, is listed in **Figure 3**. I used Visual Studio 2010 to create a new C# console application named ClusteringKMeans; any recent version of Visual Studio should work, too. In the Solution Explorer window I renamed file Program.cs to ClusteringKMeansProgram.cs, which automatically renamed the template-generated class. I removed unneeded using statements at the top of the file.

For simplicity I used a static method approach and removed all error-checking. The first part of the demo code sets up the height and weight data to be clustered. Because there are only 20 tuples, I hardcoded the data and stored the data in memory in an array named rawData. Typically, your data will be stored in a text file or SQL table. In those cases you'll have to write a helper function to load the data into memory. If your data source is too large to fit into machine memory, you'll have to modify the demo code to iterate through an external data source rather than a data array.

> In principle, at least, the k-means algorithm is quite simple. But as you'll see, some of the implementation details are a bit tricky.

After setting up the raw data, the demo program calls helper function ShowMatrix to display the data. Next, variables numAttributes, numClusters, and maxCount are assigned values of 2 (height and weight), 3 and 30, respectively. Recall maxCount limits the number of iterations in the main algorithm processing loop. The k-means algorithm tends to converge quickly, but you might have to experiment a bit with the value of maxCount.

All the clustering work is performed by method Cluster. The method returns an int array that defines how each tuple is

Figure 3 **Overall Program Structure**

```
using System;
namespace ClusteringKMeans
{
  class ClusteringKMeansProgram
  {
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin outlier data detection demo\n");

        Console.WriteLine("Loading all (height-weight) data into memory");
        string[] attributes = new string[] { "Height", "Weight" };
        double[][] rawData = new double[20][];

        rawData[0] = new double[] { 65.0, 220.0 };
        rawData[1] = new double[] { 73.0, 160.0 };
        rawData[2] = new double[] { 59.0, 110.0 };
        rawData[3] = new double[] { 61.0, 120.0 };
        rawData[4] = new double[] { 75.0, 150.0 };
        rawData[5] = new double[] { 67.0, 240.0 };
        rawData[6] = new double[] { 68.0, 230.0 };
        rawData[7] = new double[] { 70.0, 220.0 };
        rawData[8] = new double[] { 62.0, 130.0 };
        rawData[9] = new double[] { 66.0, 210.0 };
        rawData[10] = new double[] { 77.0, 190.0 };
        rawData[11] = new double[] { 75.0, 180.0 };
        rawData[12] = new double[] { 74.0, 170.0 };
        rawData[13] = new double[] { 70.0, 210.0 };
        rawData[14] = new double[] { 61.0, 110.0 };
        rawData[15] = new double[] { 58.0, 100.0 };
        rawData[16] = new double[] { 66.0, 230.0 };
        rawData[17] = new double[] { 59.0, 120.0 };
        rawData[18] = new double[] { 68.0, 210.0 };
        rawData[19] = new double[] { 61.0, 130.0 };

        Console.WriteLine("\nRaw data:\n");
        ShowMatrix(rawData, rawData.Length, true);

        int numAttributes = attributes.Length;
        int numClusters = 3;
        int maxCount = 30;

        Console.WriteLine("\nk = " + numClusters + " and maxCount = " + maxCount);
        int[] clustering = Cluster(rawData, numClusters, numAttributes, maxCount);
        Console.WriteLine("\nClustering complete");

        Console.WriteLine("\nClustering in internal format: \n");
        ShowVector(clustering, true);

        Console.WriteLine("\nClustered data:");
        ShowClustering(rawData, numClusters, clustering, true);

        double[] outlier = Outlier(rawData, clustering, numClusters, 0);
        Console.WriteLine("Outlier for cluster 0 is:");
        ShowVector(outlier, true);

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

    // 14 short static method definitions here
  }
}
```

Figure 4 **Method UpdateMeans**

```
static void UpdateMeans(double[][] rawData, int[] clustering,
  double[][] means)
{
  int numClusters = means.Length;
  for (int k = 0; k < means.Length; ++k)
    for (int j = 0; j < means[k].Length; ++j)
      means[k][j] = 0.0;

  int[] clusterCounts = new int[numClusters];
  for (int i = 0; i < rawData.Length; ++i)
  {
    int cluster = clustering[i];
    ++clusterCounts[cluster];

    for (int j = 0; j < rawData[i].Length; ++j)
      means[cluster][j] += rawData[i][j];
  }

  for (int k = 0; k < means.Length; ++k)
    for (int j = 0; j < means[k].Length; ++j)
      means[k][j] /= clusterCounts[k]; // danger

  return;
}
```

assigned to one cluster. After finishing, the demo program displays the encoded clustering and also displays the raw data, grouped according to cluster.

The demo program concludes by analyzing the clustered data for outlier, possibly abnormal, tuples using method Outliers. That method accepts a cluster ID and returns the values of the data tuple that's the farthest (as measured by Euclidean distance) from the cluster centroid (most representative tuple). In this case, for cluster 0, the heavy person cluster, the outlier tuple is (67.0, 240.0), the heaviest person.

Computing Cluster Centroids

Recall that a cluster centroid is a tuple that is most representative of the tuples assigned to a cluster, and that one way to determine a cluster centroid is to compute a math average tuple and then find the one tuple that's closest to the average tuple. Helper method UpdateMeans computes the math average tuple for each cluster and is listed in **Figure 4**.

Method UpdateMeans assumes that an array of arrays named means already exists, as opposed to creating the array and then returning it. Because array means is assumed to exist, you might want to make it a ref parameter. Array means is created using helper method Allocate:

```
static double[][] Allocate(int numClusters, int numAttributes)
{
  double[][] result = new double[numClusters][];
  for (int k = 0; k < numClusters; ++k)
    result[k] = new double[numAttributes];
  return result;
}
```

The first index in the means array represents a cluster ID and the second index indicates the attribute. For example, if means[0][1] = 150.33 then the average of the weight (1) values of the tuples in cluster 0 is 150.33.

Method UpdateMeans first zeros out the existing values in array means, then iterates through each data tuple and tallies the count of tuples in each cluster and accumulates the sums for each attribute, and then divides each accumulated sum by the appropriate cluster count. Notice that the method will throw an exception if any cluster count is 0, so you might want to add an error-check.

Method ComputeCentroid (listed in **Figure 5**) determines the centroid values—the values of the one tuple that's closest to the average tuple values for a given cluster.

Method ComputeCentroid iterates through each tuple in the data set, skipping tuples that aren't in the specified cluster. For each tuple in the specified cluster, the Euclidean distance between the tuple and the cluster mean is calculated using helper method Distance. The tuple values that are closest (having the smallest distance) to the mean values are stored and returned.

Method UpdateCentroids calls ComputeCentroid for each cluster to give the centroids for all clusters:

```
static void UpdateCentroids(double[][] rawData, int[] clustering,
  double[][] means, double[][] centroids)
{
  for (int k = 0; k < centroids.Length; ++k)
  {
    double[] centroid = ComputeCentroid(rawData, clustering, k, means);
    centroids[k] = centroid;
  }
}
```

Method UpdateCentroids assumes that an array of arrays named centroids exists. Array centroids is very similar to array means: The first index represents a cluster ID and the second index indicates the data attribute.

Figure 5 **Method ComputeCentroid**

```
static double[] ComputeCentroid(double[][] rawData, int[] clustering,
  int cluster, double[][] means)
{
  int numAttributes = means[0].Length;
  double[] centroid = new double[numAttributes];
  double minDist = double.MaxValue;
  for (int i = 0; i < rawData.Length; ++i) // walk thru each data tuple
  {
    int c = clustering[i];
    if (c != cluster) continue;

    double currDist = Distance(rawData[i], means[cluster]);
    if (currDist < minDist)
    {
      minDist = currDist;
      for (int j = 0; j < centroid.Length; ++j)
        centroid[j] = rawData[i][j];
    }
  }
  return centroid;
}
```

Figure 6 **Method Assign**

```
static bool Assign(double[][] rawData, int[] clustering, double[][] centroids)
{
  int numClusters = centroids.Length;
  bool changed = false;

  double[] distances = new double[numClusters];
  for (int i = 0; i < rawData.Length; ++i)
  {
    for (int k = 0; k < numClusters; ++k)
      distances[k] = Distance(rawData[i], centroids[k]);

    int newCluster = MinIndex(distances);
    if (newCluster != clustering[i])
    {
      changed = true;
      clustering[i] = newCluster;
    }
  }
  return changed;
}
```

All these data sources **at your fingertips** – and that is just a start.

RSSBus Data Providers [ADO.NET]

Build cutting-edge .NET applications that connect to any data source with ease.

- *Easily "databind" to applications, databases, and services using standard Visual Studio wizards.*
- *Comprehensive support for CRUD (Create, Read, Update, and Delete operations).*
- *Industry standard ADO.NET Data Provider, fully integrated with Visual Studio.*

Databind to the Web...

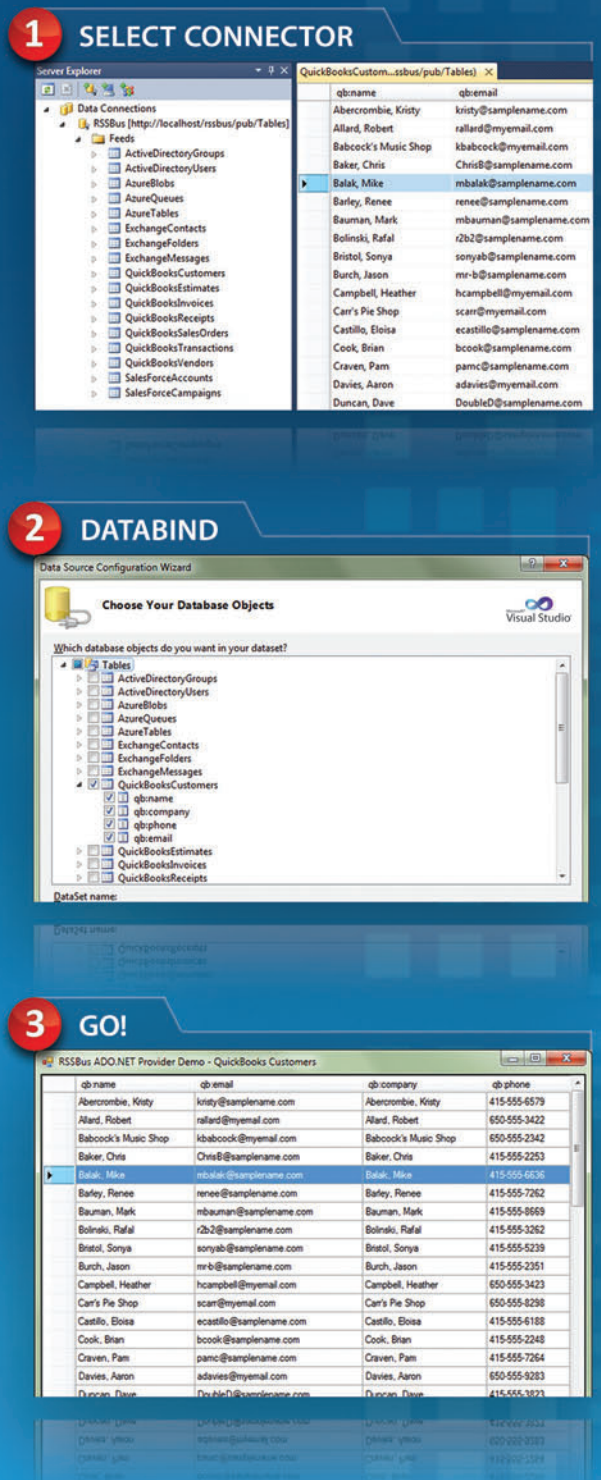
The RSSBus Data Providers give your .NET applications the power to databind (just like SQL) to Amazon, PayPal, eBay, QuickBooks, FedEx, Salesforce, MS-CRM, Twitter, SharePoint, Windows Azure, and much more! Leverage your existing knowledge to deliver cutting-edge WinForms, ASP.NET, and Windows Mobile solutions with full readwrite functionality quickly and easily.

Databind to Local Apps...

The RSSBus Data Providers make everything look like a SQL table, even local application data. Using the RSSBus Data Providers your .NET applications interact with local applications, databases, and services in the same way you work with SQL Tables and Stored Procedures. No code required. It simply doesn't get any easier!

"Databind to anything...
...just like you do with SQL"

Also available for:
JDBC | ODBC | SQL SSIS | Excel | OData | SharePoint ...



rssbus

Download a free 30-day trial online at: **www.rssbus.com**

Figure 7 **The Cluster Method**

```
static int[] Cluster(double[][] rawData, int numClusters,
  int numAttributes,  int maxCount)
{
  bool changed = true;
  int ct = 0;

  int numTuples = rawData.Length;
  int[] clustering = InitClustering(numTuples, numClusters, 0);
  double[][] means = Allocate(numClusters, numAttributes);
  double[][] centroids = Allocate(numClusters, numAttributes);
  UpdateMeans(rawData, clustering, means);
  UpdateCentroids(rawData, clustering, means, centroids);

  while (changed == true && ct < maxCount)
  {
    ++ct;
    changed = Assign(rawData, clustering, centroids);
    UpdateMeans(rawData, clustering, means);
    UpdateCentroids(rawData, clustering, means, centroids);
  }
  return clustering;
}
```

To summarize, each cluster has a centroid, which is the most representative tuple in the cluster. Centroid values are computed by finding the one tuple in each cluster that's closest to the average tuple (the mean) in each cluster. Each data tuple is assigned to the cluster whose cluster centroid is closest to the tuple.

The Distance Function and Data Normalization

Method ComputeCentroid calls a Distance method to determine which data tuple is closest to a cluster mean. As described earlier, the most common way to measure distance from tuples to means is to use Euclidean distance:

```
static double Distance(double[] tuple, double[] vector)
{
  double sumSquaredDiffs = 0.0;
  for (int j = 0; j < tuple.Length; ++j)
    sumSquaredDiffs += Math.Pow((tuple[j] - vector[j]), 2);
  return Math.Sqrt(sumSquaredDiffs);
}
```

You might want to consider alternative ways to define distance. A very common option is to use the sum of the absolute values of the differences between each component. Because Euclidean distance squares differences, larger differences are weighted much more heavily than smaller differences.

Figure 8 **The Outlier Method**

```
static double[] Outlier(double[][] rawData, int[] clustering,
  int numClusters, int cluster)
{
  int numAttributes = rawData[0].Length;

  double[] outlier = new double[numAttributes];
  double maxDist = 0.0;

  double[][] means = Allocate(numClusters, numAttributes);
  double[][] centroids = Allocate(numClusters, numAttributes);
  UpdateMeans(rawData, clustering, means);
  UpdateCentroids(rawData, clustering, means, centroids);

  for (int i = 0; i < rawData.Length; ++i)
  {
    int c = clustering[i];
    if (c != cluster) continue;
    double dist = Distance(rawData[i], centroids[cluster]);
    if (dist > maxDist)
    {
      maxDist = dist;
      Array.Copy(rawData[i], outlier, rawData[i].Length);
    }
  }
  return outlier;
}
```

Another important factor related to the choice of distance function in the k-means clustering algorithm is data normalization. The demo program uses raw, un-normalized data. Because tuple weights are typically values such as 160.0 and tuple heights are typically values like 67.0, differences in weights have much more influence than differences in heights. In many situations, in addition to exploring clustering on raw data, it's useful to normalize the raw data before clustering. There are many ways to normalize data. A common technique is to compute the mean (m) and standard deviation (sd) for each attribute, then for each attribute value (v) compute a normalized value nv = (v-m)/sd.

Assigning Each Tuple to a Cluster

With a method to compute the centroid of each cluster in hand, it's possible to write a method to assign each tuple to a cluster. Method Assign is listed in **Figure 6**.

Method Assign accepts an array of centroid values and iterates through each data tuple. For each data tuple, the distance to each of the cluster centroids is computed and stored in a local array named distances, where the index of the array represents a cluster ID. Then helper method MinIndex determines the index in array distances that has the smallest distance value, which is the cluster ID of the cluster that has centroid closest to the tuple.

Here's helper method MinIndex:

```
static int MinIndex(double[] distances)
{
  int indexOfMin = 0;
  double smallDist = distances[0];
  for (int k = 0; k < distances.Length; ++k)
  {
    if (distances[k] < smallDist)
    {
      smallDist = distances[k]; indexOfMin = k;
    }
  }
  return indexOfMin;
}
```

In Assign, if the computed cluster ID is different from the existing cluster ID stored in array clustering, array clustering is updated and a Boolean flag to indicate that there has been at least one change in the clustering is toggled. This flag will be used to determine when to stop the main algorithm loop—when the maximum number of iterations is exceeded or when there's no change in the clustering.

This implementation of the k-means algorithm assumes that there's always at least one data tuple assigned to each cluster. As given in **Figure 6**, method Assign does not prevent a situation where a cluster has no tuples assigned. In practice, this usually isn't a problem. Preventing the error condition is a bit tricky. The approach I generally use is to create an array named centroidIndexes that works in conjunction with array centroids. Recall that array centroids holds centroid values, for example (61.0, 120.0) is the centroid for cluster 2 in **Figure 2**. Array centroidIndexes holds the associated tuple index, for example [3]. Then in the Assign method, the first step is to assign to each cluster the data tuple that holds the centroid values, and only then does the method iterate through each remaining tuple and assign each to a cluster. This approach guarantees that every cluster has at least one tuple.

The Cluster Method

Method Cluster, listed in **Figure 7**, is the high-level routine that calls all the helper and sub-helper methods to actually perform the data clustering.

The main while loop repeatedly assigns each data tuple to a cluster, computes the new tuple means for each cluster, then uses the new means to compute the new centroid values for each cluster. The loop exits when there's no change in cluster assignment or some maximum count is reached. Because the means array is used only to compute centroids, you might want to refactor Cluster by placing the call to UpdateMeans inside method UpdateCentroids.

Before kicking the processing loop off, the clustering array is initialized by method InitClustering:

```
static int[] InitClustering(int numTuples, int numClusters, int
randomSeed)
{
  Random random = new Random(randomSeed);
  int[] clustering = new int[numTuples];
  for (int i = 0; i < numClusters; ++i)
    clustering[i] = i;
  for (int i = numClusters; i < clustering.Length; ++i)
    clustering[i] = random.Next(0, numClusters);
  return clustering;
}
```

The InitClustering method first assigns tuples 0 through numClusters-1 to clusters 0 through numClusters-1, respectively, so that every cluster will start with at least one tuple assigned. The remaining tuples are assigned to a randomly selected cluster.

A somewhat surprising amount of research has been done on k-means clustering initialization and you may want to experiment with alternatives to the approach given here. In many cases, the final clustering produced by the k-means algorithm depends on how the clustering is initialized.

Looking for Abnormal Data

One way to use data clustering is to simply explore different clusterings and look for unexpected or surprising results. Another possibility is to look for unusual data tuples within a cluster. The demo program checks cluster 0 to find the tuple in that cluster that's farthest from the cluster centroid using a method named Outlier, which is listed in **Figure 8**.

After initializing means and centroids arrays, method Outlier iterates through each tuple in the specified cluster and computes the Euclidean distance from the tuple to the cluster centroid, then returns the values of the tuple that has the greatest distance to the centroid values. A minor alternative for you to consider is to return the index of the farthest data tuple.

There are many other ways you can examine clustered data for abnormalities. For example, you might want to determine the average distance between each tuple and its assigned cluster centroid, or you might want to examine the distances of the cluster centroids from each other.

Display Routines

For the sake of completeness, here are some simplified display routines. The code download has slightly fancier versions. If you use these simplified routines, you'll have to modify their calls in the Main method. To display raw data, means and centroids you can use:

```
static void ShowMatrix(double[][] matrix)
{
  for (int i = 0; i < numRows; ++i)
  {
    Console.Write("[" + i.ToString().PadLeft(2) + "]  ");
    for (int j = 0; j < matrix[i].Length; ++j)
      Console.Write(matrix[i][j].ToString("F1") + "  ");
    Console.WriteLine("");
  }
}
```

To display the clustering array you can use:

```
static void ShowVector(int[] vector)
{
  for (int i = 0; i < vector.Length; ++i)
    Console.Write(vector[i] + " ");
  Console.WriteLine("");
}
```

To display an outlier's values you can use:

```
static void ShowVector(double[] vector)
{
  for (int i = 0; i < vector.Length; ++i)
    Console.Write(vector[i].ToString("F1") + " ");
  Console.WriteLine("");
}
```

And to display raw data grouped by cluster you can use:

```
static void ShowClustering(double[][] rawData, int numClusters, int[] clustering)
{
  for (int k = 0; k < numClusters; ++k) // Each cluster
  {
    for (int i = 0; i < rawData.Length; ++i) // Each tuple
      if (clustering[i] == k)
      {
        for (int j = 0; j < rawData[i].Length; ++j)
          Console.Write(rawData[i][j].ToString("F1") + " ");
        Console.WriteLine("");
      }
    Console.WriteLine("");
  }
}
```

Wrapping Up

Data clustering is closely related to and sometimes confused with data classification. Clustering is an unsupervised technique that groups data items together without any foreknowledge of what those groups might be. Clustering is typically an exploratory process. Classification, in contrast, is a supervised technique that requires the specification of known groups in training data, after which each data tuple is placed into one of these groups. Classification is typically used for prediction purposes.

The code and explanation presented in this article should give you enough information to experiment with k-means data clustering, or to create a fully customizable standalone clustering tool, or to add clustering features to a .NET application without relying on any external dependencies. There are many other clustering algorithms in addition to k-means and I'll present some of these in future *MSDN Magazine* articles, including data entropy minimization, category utility and Naive Bayes inference. ■

Dr. James McCaffrey *works for Volt Information Sciences Inc., where he manages technical training for software engineers working at the Microsoft 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: *Darren Gehring*

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

Sweet 127.0.0.1 Chicago!

Visual Studio Live! is thrilled to be back in Chicago! Register for your backstage pass to the Microsoft Platform and join your fellow developers, software architects, designers and more for 4 days of unbiased training at Visual Studio Live! Chicago.

Topics will include:

- ASP.NET
- Azure / Cloud Computing
- Cross-Platform Mobile
- Data Management
- HTML5 / JavaScript
- Windows 8 / WinRT
- WPF / Silverlight
- Visual Studio 2012 / .NET 4.5

REGISTER TODAY AND SAVE $300

USE PROMO CODE CHFEB1

VISUAL STUDIO LIVE! CHICAGO

HILTON CHICAGO | **MAY 13-16, 2013**

vslive.com/chicago

SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA

Taming the Event Stream: Fast Approximate Counting

Michael Meijer

So you have a voluminous and potentially infinite stream of events such as a clickstream, sensor data, credit-card transaction data or Internet traffic. It's infeasible to store all events or analyze them in multiple passes. Why not resort to a window of recent events to simplify analysis?

Suppose you want to count the number of interesting events in a large window covering the latest N events of the stream. A naïve approach to counting requires all N events to be in memory and a full iteration over them. As the window slides upon the arrival of a new event, its oldest event expires and the new event is inserted. Counting over the new window from scratch wastes the processing time spent on N-2 events shared. Yuck! This article explains a data structure to reduce memory space usage and processing time to a small fraction of what would be required with that method, while supporting an event rate exceeding many thousands of events per second on commodity hardware. This article also shows how to embed the data structure in a user-defined stream operator in C# for the Microsoft streaming data processor, StreamInsight 2.1. Intermediate programming skills are required to follow along, and some experience with StreamInsight can come in handy.

This article discusses:

- Approximate event counting basics
- Using buckets for counting
- Using StreamInsight for counting

Technologies discussed:

StreamInsight 2.1

Code download available at:

archive.msdn.microsoft.com/mag201302StreamInsight

A Tale of Counting

Before diving into StreamInsight, I'll investigate the seemingly trivial problem of counting. For simplicity, assume the stream has events with payloads of 0 or 1—uninteresting and interesting events, respectively (regardless of what constitutes "interesting" in your specific scenario). The number of 1s is counted over a (fixed-size) count-based window containing the most recent N events. Naïve counting takes O(N) time and space.

As an astute reader, you probably came up with the idea of maintaining the count between consecutive windows and incrementing it for new 1s and decrementing it for expired 1s, sharing the N-2 events already processed. Good thinking! Maintaining the count now takes O(1) time. However, should you decrement for an expired event or not? Unless you know the actual event, the count can't be maintained. Unfortunately, to know the events until they have expired requires the entire window in memory—that is, it takes O(N) space. Another strategy might be to filter out the uninteresting events and count only the remaining interesting events. But that doesn't reduce computational complexity and leaves you with a variable-size window.

Figure 1 **The Exponential Histogram Class Outline**

```
[DataContract]
public class ExponentialHistogram
{
  [DataMember]
  private long n;

  [DataMember]
  private double epsilon;

  [DataMember]
  private long total;

  [DataMember]
  private LinkedList<Bucket> buckets;

  public ExponentialHistogram(long n, double epsilon)
  {
    this.n = n;
    this.epsilon = epsilon;
    this.buckets = new LinkedList<Bucket>();
  }

  public void Update(long timestamp, bool e) { ... }

  protected void ExpireBuckets(long timestamp) { ... }

  protected void PrependNewBucket(long timestamp) { ... }

  protected void MergeBuckets() { ... }

  public long Query() { ... }
}
```

Can the memory beast be tamed? Yes, it can! However, it requires a compromise between processing time and memory space at the expense of accuracy. The seminal paper by Mayur Datar, Aristides Gionis, Piotr Indyk and Rajeev Motwani titled "Maintaining Stream Statistics over Sliding Windows" (stanford.io/SRjWT0) describes a data structure called the exponential histogram. It maintains an approximate count over the last N events with a bounded relative error ε. This means that at all times:

$$\frac{|\text{exact count} - \text{approximate count}|}{\text{exact count}} \leq \varepsilon, \textit{ where } 0 < \varepsilon < 1$$

Conceptually, the histogram stores events in buckets. Every bucket initially covers one event, so it has a count of 1 and a timestamp of the event it covers. When an event arrives, expired buckets (covering expired events) are removed. A bucket is created only for an interesting event. As buckets are created over time, they're merged to save memory. Buckets are merged so they have exponentially growing counts from the most recent to the last bucket, that is, 1, 1, ..., 2, 2, ..., 4, 4, ..., 8, 8 and so on. This way, the number of buckets is logarithmic in the window size N. More precisely, it requires $O(\frac{1}{\varepsilon} \log N)$ time and space for maintenance. All but the last bucket cover only non-expired events. The last bucket covers at least one non-expired event. Its count must be estimated, which causes the error in approximating the overall count. Hence, the last bucket must be kept small enough to respect the relative error upper bound.

In the next section, the implementation of the exponential histogram in C# is discussed with a bare minimum of math. Read the aforementioned paper for the intricate details. I'll explain the code and follow up with a pen-and-paper example. The histogram is a building block for the StreamInsight user-defined stream operator developed later in this article.

To Bucket or Not to Bucket

Here's the bucket class:

```
[DataContract]
public class Bucket
{
  [DataMember]
  private long timestamp;

  [DataMember]
  private long count;

  public long Timestamp {
    get { return timestamp; }
    set { timestamp = value; } }

  public long Count { get { return count; } set { count = value; } }
}
```

It has a count of the (interesting) events it covers and a timestamp of the most recent event it covers. Only the last bucket can cover expired events, as mentioned, but it must cover at least one non-expired event. Hence, all but the last bucket counts are exact. The last bucket count must be estimated by the histogram. Buckets containing only expired events are themselves expired and can be removed from the histogram.

Using just two operations, the exponential histogram ensures a relative error upper bound ε on the count of interesting events over the N most recent events. One operation is for updating the histogram with new and expired events, maintaining the buckets. The other is for querying the approximate count from the buckets. The histogram class outline is shown in **Figure 1**. Next to the linked list of buckets, its key variables are the window size (n), the relative error upper bound (epsilon) and the cached sum of all bucket counts (total). In the constructor, the given window size, the given relative error upper bound and an initial empty list of buckets are set.

> A naïve approach to counting requires all N events to be in memory and a full iteration over them.

The maintenance of the histogram is performed by this update method:

```
public void Update(long timestamp, bool eventPayload)
{
  RemoveExpiredBuckets(timestamp);

  // No new bucket required; done processing
  if (!eventPayload)
    return;

  PrependNewBucket(timestamp);

  MergeBuckets();
}
```

It accepts a discrete timestamp, as opposed to wall-clock time, to determine what the latest N events are. This is used to find and remove expired buckets. If the new event has a payload of 0 (false), processing stops. When the new event has a payload of 1 (true), a new bucket is created and prepended to the list of buckets. The real fireworks are in merging the buckets. The methods called by the update method are discussed in sequence.

Here's the code for the removal of buckets:

```
protected void RemoveExpiredBuckets(long timestamp)
{
  LinkedListNode<Bucket> node = buckets.Last;

  // A bucket expires if its timestamp
  // is before or at the current timestamp - n
  while (node != null && node.Value.Timestamp <= timestamp - n)
  {
    total -= node.Value.Count;

    buckets.RemoveLast();
    node = buckets.Last;
  }
}
```

The traversal starts from the oldest (last) bucket and ends at the first non-expired bucket. Each bucket whose most recent event's timestamp is expired—that is, whose timestamp is no greater than the current timestamp minus the window size—is removed from the list. This is where the discrete timestamp comes in. The sum of all bucket counts (total) is decremented by the count of each expired bucket.

After expired events and buckets are accounted for, the new event is processed:

```
protected void PrependNewBucket(long timestamp)
{
  Bucket newBucket = new Bucket()
  {
    Timestamp = timestamp,
    Count = 1
  };

  buckets.AddFirst(newBucket);

  total++;
}
```

Figure 2 **Merging Buckets in the Histogram**

```
protected void MergeBuckets()
{
  LinkedListNode<Bucket> current = buckets.First;
  LinkedListNode<Bucket> previous = null;

  int k = (int)Math.Ceiling(1 / epsilon);
  int kDiv2Add2 = (int)(Math.Ceiling(0.5 * k) + 2);
  int numberOfSameCount = 0;

  // Traverse buckets from first to last, hence in order of
  // descending timestamp and ascending count
  while (current != null)
  {
    if (previous != null && previous.Value.Count == current.Value.Count)
      numberOfSameCount++;
    else
      numberOfSameCount = 1;

    // Found k/2+2 buckets of the same count?
    if (numberOfSameCount == kDiv2Add2)
    {
      // Merge oldest (current and previous) into current
      current.Value.Timestamp = previous.Value.Timestamp;
      current.Value.Count = previous.Value.Count + current.Value.Count;

      buckets.Remove(previous);

      // A merged bucket can cause a cascade of merges due to
      // its new count, continue iteration from merged bucket
      // otherwise the cascade might go unnoticed
      previous = current.Previous;
    }
    else
    {
      // No merge, continue iteration with next bucket
      previous = current;
      current = current.Next;
    }
  }
}
```

A new bucket for the event with a payload of 1 (true) is created with a count of 1 and a timestamp equal to the current timestamp. The new bucket is prepended to the list of buckets and the sum of all bucket counts (total) is incremented.

The memory space-saving and error-bounding magic is in the merging of buckets. The code is listed in **Figure 2**. Buckets are merged so that consecutive buckets have exponentially growing counts, that is, 1, 1, ..., 2, 2, ..., 4, 4, ..., 8, 8 and so on. The number of buckets with the same count is determined by the choice of the relative error upper bound ε. The total number of buckets grows logarithmically with the size of the window n, which explains the memory space savings. As many buckets as possible are merged, but the last bucket's count is kept small enough (compared to the sum of the other bucket counts) to ensure the relative error is bounded.

More formally, buckets have non-decreasing counts from the first (most recent) to the last (oldest) bucket in the list. The bucket counts are constrained to powers of two. Let $k = \frac{1}{\varepsilon}$ and $\frac{k}{2}$ be an integer, or else replace the latter by $\frac{k}{2}$. Except for the last bucket's count, let there be at least $\frac{k}{2}$ and at most $\frac{k}{2} + 1$ buckets of the same count. Whenever there are $\frac{k}{2} + 2$ buckets of the same count, the oldest two are merged into one bucket with twice the count of the oldest bucket and the most recent of their timestamps. Whenever two buckets are merged, traversal continues from the merged bucket. The merge can cause a cascade of merges. Otherwise traversal continues from the next bucket.

To get a feeling for the count approximation, look at the histogram's query method:

```
public long Query()
{
  long last = buckets.Last != null ? buckets.Last.Value.Count : 0;
  return (long)Math.Ceiling(total - last / 2.0);
}
```

The sum of the bucket counts up to the last bucket is exact. The last bucket must cover at least one non-expired event, otherwise the bucket is expired and removed. Its count must be estimated because it can cover expired events. By estimating the actual count of the last bucket as half the last bucket's count, the absolute error of that estimate is no larger than half that bucket's count. The overall count is estimated by the sum of all bucket counts (total) minus half the last bucket's count. To ensure the absolute error is within bounds of the relative error, the last bucket's influence must be small enough compared to the sum of the other bucket counts. Thankfully, this is ensured by the merge procedure.

Do the code listings and explanations up to this point leave you puzzled about the workings of the histogram? Read through the following example.

Suppose you have a newly initialized histogram with window size $n = 7$ and relative error upper bound $\varepsilon = 0.5$, so $k = 2$. The histogram develops as shown in **Figure 3**, and a schematic overview of this histogram is depicted in **Figure 4**. In **Figure 3**, merges are at timestamps 5, 7 and 9. A cascaded merge is at timestamp 9. An expired bucket is at timestamp 13. I'll go into more detail about this.

The first event has no effect. At the fifth event, a merge of the oldest buckets occurs because there are $\frac{k}{2} + 2$ buckets with the same count of 1. Again, a merge happens at the seventh event. At the ninth event, a merge cascades into another merge. Note that after the seventh event, the first event expires. No bucket carries

Figure 3 **Example of the Exponential Histogram**

Timestamp	Event	Buckets (Timestamp, Count) Newest → ... → Oldest	Total	Query	Exact	Relative Error
1	0		0	0	0	0
2	1	(2,1)	1	1	1	0
3	1	(3,1) → (2,1)	2	2	2	0
4	0	(3,1) → (2,1)	2	2	2	0
5 (merge)	1	(5,1) → (3,1) → (2,1) (5,1) → (3,2)	3	2	3	0.333...
6	1	(6,1) → (5,1) → (3,2)	4	3	4	0.25
7 (merge)	1	(7,1) → (6,1) → (5,1) → (3,2) (7,1) → (6,2) → (3,2)	5	4	5	0.2
8	1	(8,1) → (7,1) → (6,2) → (3,2)	6	5	6	0.166...
9 (merge) (cascaded merge)	1	(9,1) → (8,1) → (7,1) → (6,2) → (3,2) (9,1) → (8,2) → (6,2) → (3,2) (9,1) → (8,2) → (6,4)	7	5	6	0.166...
10	0	(9,1) → (8,2) → (6,4)	7	5	5	0
11	0	(9,1) → (8,2) → (6,4)	7	5	5	0
12	0	(9,1) → (8,2) → (6,4)	7	5	4	0.25
13	0	(9,1) → (8,2)	3	2	3	0.333...

an expired timestamp until the 13th event. At the 13th event, the bucket with timestamp 6 no longer covers at least one non-expired event and thus expires. Note that the observed relative error is clearly less than the relative error upper bound.

> The memory savings of the histogram are huge compared to windows.

In **Figure 4**, a dotted box is the window size at that point; it contains the buckets and implies the span of events covered. A solid box is a bucket with timestamp on top and count on bottom. Situation A shows the histogram at timestamp 7 with arrows to the counted events. Situation B shows the histogram at timestamp 9. The last bucket covers expired events. Situation C shows the histogram at timestamp 13. The bucket with timestamp 6 expired.

After putting it all together, I wrote a small demonstration program for the exponential histogram (check out the source code download for this article). The results are shown in **Figure 5**. It simulates a stream of 100 million events with a count-based window size N of 1 million events. Each event has a payload of 0 or 1 with 50 percent chance. It estimates the approximate count of 1s with an arbitrarily chosen relative error upper bound ε of 1 percent, or 99 percent accuracy. The memory savings of the histogram are huge compared to windows; the number of buckets is far less than the number of events in the window. An event rate of a few hundred thousand events per second is achieved on a machine with an Intel 2.4 GHz dual-core processor and 3GB of RAM running Windows 7.

A Beauty Called StreamInsight

Perhaps you're wondering what Microsoft StreamInsight is and where it fits in. This section provides some basics. StreamInsight is a robust, high-performance, low-overhead, near-zero-latency and extremely flexible engine for processing on streams. It's currently at version 2.1. The full version requires a SQL Server license, though a trial version is available. It's run either as a stand-alone service or embedded in-process.

At the heart of streaming data processing is a model with temporal streams of events. Conceptually, it's a potentially infinite and voluminous collection of data arriving over time. Think of stock exchange prices, weather telemetry, power monitoring, Web clicks, Internet traffic, toll booths and so on. Each event in the stream has a header with metadata and a payload of data. In the header of the event, a timestamp is kept, at a minimum. Events can arrive steadily, intermittently or perhaps in bursts of up to many thousands per second. Events come in three flavors: An event can be confined to a point in time; be valid for a certain interval; or be valid for an open-ended interval (edge). Besides events from the stream, a special punctuation event is issued by the engine called the Common Time Increment (CTI). Events can't be inserted into the stream with a timestamp less than the CTI's timestamp. Effectively, CTI events determine the extent to which events can arrive out of order. Thankfully, StreamInsight takes care of this.

Heterogeneous sources of input and sinks of output streams must somehow be adapted to fit into this model. The events in the (queryable) temporal streams are captured in an IQStreamable<TPayload>.

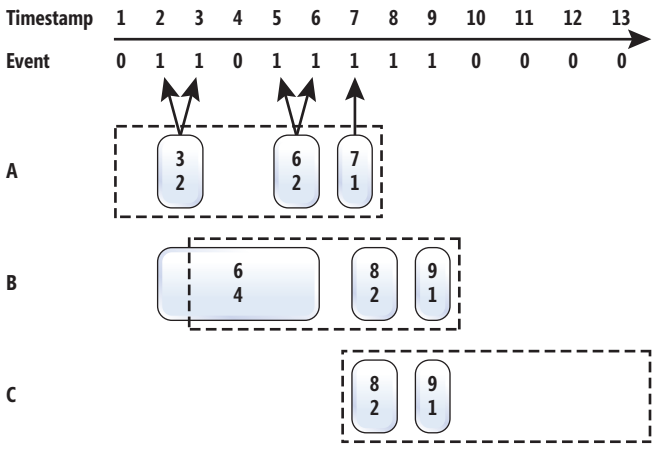


Figure 4 **A Schematic Overview of the Histogram Depicted in Figure 3**

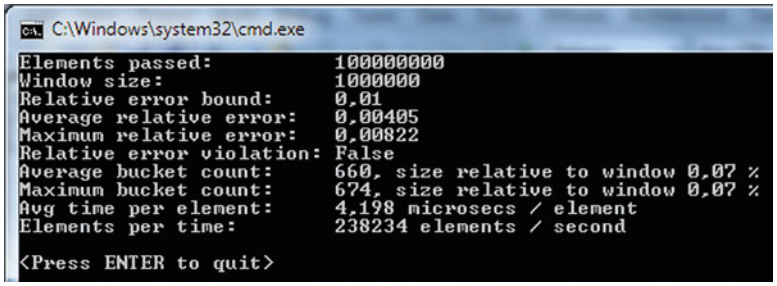


Figure 5 **Empirical Results for the Exponential Histogram**

Event payloads are conceptually pulled by enumeration or pushed by observation into the stream. Hence, underlying data can be exposed through an IEnumerable<T>/IQueryable<T> (Reactive Extension) or IObservable<T>/IQbservable<T> (Reactive Extension), respectively, parameterized with the type of data exposed. They leave the maintenance of temporal aspects to the processing engine. Conversion from and to the various interfaces is possible.

The sources and sinks just discussed live on the boundaries, whereas the actual processing happens within queries. A query is a basic unit of composition written in LINQ. It continuously processes events from one or more streams and outputs another stream. Queries are used to project, filter, group-apply, multicast, operate/aggregate, join, union and window events. Operators can be user-defined. They work on events (incremental) or on windows (non-incremental) as they arrive.

A note on windowing is in order. Windowing partitions a stream into finite subsets of events that might overlap between consecutive windows. Windowing effectively produces a stream of windows, reflected by an IQWindowedStreamable<TPayload> in StreamInsight. Currently, three different kinds of windowing constructs are supported: count-based, time-based and snapshot windows.

Figure 6 **User-Defined Stream Operator Implementation**

```
[DataContract]
public class ApproximateCountUDSO : CepPointStreamOperator<bool, long>
{
  [DataMember]
  private ExponentialHistogram histogram;

  [DataMember]
  private long currentTimestamp;  // Current (discrete) timestamp

  public ApproximateCountUDSO(long n, double epsilon)
  {
    histogram = new ExponentialHistogram(n, epsilon);
  }

  public override IEnumerable<long> ProcessEvent(
    PointEvent<bool> inputEvent)
  {
    currentTimestamp++;

    histogram.Update(currentTimestamp, inputEvent.Payload);


    yield return histogram.Query();
  }

  public override bool IsEmpty
  {
    get { return false; }
  }
}
```

Count-based windows cover the most recent N events and slide upon the arrival of a new event, expiring the oldest and inserting the newest. Time-based windows cover the most recent events in the most recent interval of time and slide by some interval (also called hopping or tumbling). Snapshot windows are driven by event start and end times; that is, for every pair of closest event start and end times, a window is created. In contrast to time-based windows driven by intervals along the timeline, regardless of events, snapshot windows aren't fixed along the timeline.

That just scratches the surface. More information is available from several sources, including the online Developer's Guide (bit.ly/T7Trrx), "A Hitchhiker's Guide to StreamInsight 2.1 Queries" (bit.ly/NbybvY), CodePlex examples, the StreamInsight team blog (blogs.msdn.com/b/streaminsight) and others.

Windowing partitions a stream into finite subsets of events that might overlap between consecutive windows.

Putting It All Together

The foundations are laid. At this point, you're probably wondering how approximate counting is brought to life in StreamInsight. In short, some (temporal) source stream of point-in-time events, carrying a payload of 0 or 1, is required. It's fed into a query that computes the approximate count of 1s over the N most recent events using the exponential histogram. The query produces some (temporal) stream of point-in-time events—carrying the approximate count—that's fed into a sink.

Let's start with a user-defined operator for approximate counting. You might be tempted to capture the N most recent events using the count-based windowing construct. Think again! That would defy the memory-saving benefits of the exponential histogram. Why? The construct forces entire windows of events to be kept in memory. It's not required by the exponential histogram, because it has an equivalent implicit notion of windowing through the maintenance of buckets. Moreover, having an operator over windows is non-incremental, that is, with no processing of events as they arrive, but only when a (next) window is available. The solution is a user-defined stream operator without explicit windowing constructs on the query. The code is listed in **Figure 6**.

The operator derives from the abstract CepPointStreamOperator<TInputPayload, TOutputPayload>. It has an exponential histogram instance variable. Note the decoration with DataContract and DataMember attributes. This informs StreamInsight how to serialize the operator—for example, for resiliency purposes. The operator overrides the IsEmpty operator to indicate it's non-empty, that is, the operator is stateful. This prevents StreamInsight from messing with the operator when minimizing memory utilization.

The ProcessEvent method is the operator's core. It increments the current (discrete) timestamp and passes it along with the event payload to the histogram's update method. The histogram handles the bucketing and is queried for the approximate count. Make sure to use the yield-return syntax, which makes the operator enumerable. Operators are generally wrapped in some extension method hidden in a utility class. This code shows how it's done:

```
public static partial class Utility
{
  public static IQStreamable<long> ApproximateCount(
    this IQStreamable<bool> source, long n, double epsilon)
  {
    return source.Scan(() => new ApproximateCountUDSO(n, epsilon));
  }
}
```

That's it! Plug the operator into a query via the extension method. A bit of extra code is required to actually demonstrate its use. Here's a trivial source stream:

```
public static partial class Utility
{
  private static Random random = new Random((int)DateTime.Now.Ticks);

  public static IEnumerable<bool> EnumeratePayloads()
  {
    while (true)  // ad infinitum
    {
      bool payload = random.NextDouble() >= 0.5;

      yield return payload;
    }
  }
}
```

This generates random payloads of 0s and 1s. The yield-return syntax turns it into an enumerable source. Put it in a utility class, if you will.

Figure 7 **Embedding and Executing in StreamInsight**

```
class Program
{
  public const long N = 10000;
  public const double Epsilon = 0.05;

  static void Main(string[] args)
  {
    using (Server server = Server.Create("StreamInsight21"))
    {
      var app = server.CreateApplication("ApproximateCountDemo");

      // Define an enumerable source
      var source = app.DefineEnumerable(() =>
        Utility.EnumeratePayloads());

      // Wrap the source in a (temporal) point-in-time event stream
      // The time settings determine when CTI events
      // are generated by StreamInsight
      var sourceStream = source.ToPointStreamable(e =>
        PointEvent.CreateInsert(DateTime.Now, e),
        AdvanceTimeSettings.IncreasingStartTime);

      // Produces a stream of approximate counts
      // over the latest N events with relative error bound Epsilon
      var query =
        from e in sourceStream.ApproximateCount(N, Epsilon) select e;

      // Unwrap the query's (temporal) point-in-time
      // stream to an enumerable sink
      var sink = query.ToEnumerable<long>();

      foreach (long estimatedCount in sink)
      {
        Console.WriteLine(string.Format(
          "Enumerated Approximate count: {0}", estimatedCount));
      }
    }
  }
}
```

The infamous Program class is shown in **Figure 7**. It creates the in-process embedded StreamInsight server instance. A so-called application instance named ApproximateCountDemo is created as a streaming processing (metadata) container, for example, for named streams, queries and so on. An enumerable source of point-in-time events is defined, using the payload-generating utility method described earlier. It's transformed into a temporal stream of point-in-time events. The query is defined with LINQ and selects the operator approximate counts computed over the source stream. This is where the extension method for the user-defined operator comes in handy. It's bootstrapped with a window size and relative error upper bound. Next, the query output is transformed into an enumerable sink, stripping the temporal properties. Finally, the sink is iterated over, thereby actively pulling the events through the pipeline. Execute the program and enjoy its number-crunching output on the screen.

> The histogram and operator can be extended to support variable-size windows, such as time-based windows.

To briefly recap, this article explains how to approximate the count over a window of events in logarithmic time and space with upper-bounded error using an exponential histogram data structure. The histogram is embedded in a StreamInsight user-defined operator.

The histogram and operator can be extended to support variable-size windows, such as time-based windows. This requires the histogram to know the window interval/timespan rather than the window size. Buckets are expired when their timestamp is before the new event's timestamp minus the window timespan. An extension to compute variance is proposed in the paper, "Maintaining Variance and k–Medians over Data Stream Windows," by Brian Babcock, Mayur Datar, Rajeev Motwani and Liadan O'Callaghan (stanford.io/UEUG0i). Apart from histograms, other so-called synopsis structures are described in literature. You can think of random samples, heavy hitters, quantiles and so on.

The source code accompanying this article is written in C# 4.0 with Visual Studio 2010 and requires StreamInsight 2.1. The code is free for use under the Microsoft Public License (Ms-PL). Note that it was developed for educational purposes and was neither optimized nor tested for production environments. ■

Michael Meijer *is as a software engineer at CIMSOLUTIONS BV, where he provides IT consulting services and software development solutions to companies throughout the Netherlands. His interests in StreamInsight and streaming data processing started during his research at the University of Twente, Enschede, Netherlands, where he received a Master of Science degree in Computer Science–Information Systems Engineering.*

Thanks to the following technical experts for reviewing this article: *Erik Hegeman, Roman Schindlauer and Bas Stemerdink*

JAMES MCCAFFREY

Naive Bayes Classification with C#

Naive Bayes classification is a machine-learning technique that can be used to predict to which category a particular data case belongs. In this article I explain how Naive Bayes classification works and present an example coded with the C# language.

There are plenty of standalone tools available that can perform Naive Bayes classification. However, these tools can be difficult or impossible to integrate directly into your application, and difficult to customize to meet specific needs. And they might have hidden copyright issues. This article will give you a solid foundation for adding Naive Bayes classification features to a .NET application, without relying on any external dependencies.

The best way to understand what Naive Bayes classification is and to see where I'm headed in the article is to examine the screenshot of a demo program in **Figure 1**. The demo program begins by generating 40 lines of data that will be used to train the classifier. In most cases you'd be using an existing data source, but I generated dummy data to keep the demo simple. The first line of data is "administrative,right,72.0,female." The first field is an occupation, the second is hand dominance, the third is height in inches and the fourth is sex. The goal of the classifier is to predict sex from a given set of values for occupation, dominance and height. Because the dependent variable sex has two possible values, this is an example of binary classification.

After generating raw data, the demo program converts each numeric height field to a category—short, medium or tall—by binning height. As I'll explain, binning numeric data into categorical data is an approach that has pros and cons. After the training data has been binned, the demo program scans the 40 lines of categorical data and computes joint counts. For example, the number of data cases where the person's occupation is administrative and the person's sex is male is 2. Additionally, the total numbers of each dependent value (the attribute to be predicted, male or female in this example) are computed. You can see that there are 24 males and 16 females in the training data.

The demo program then has all the information needed to classify the sex of a new data case where the occupation is education,

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

Begin Naive Bayes Classification demo

Demo will classify sex based on occupation, dominance, height

Generating 40 lines of occupation, dominance, height, sex data

First 4 lines of training data are:

administrative,right,72.0,female
construction,right,63.0,male
technology,right,65.9,female
administrative,right,69.0,female


Converting numeric height data to categorical data on 64.0 71.0

First 4 lines of binned training data are:

administrative,right,tall,female
construction,right,short,male
technology,right,medium,female
administrative,right,medium,female


Scanning binned data to compute joint and dependent counts

Total male = 24
Total female = 16

administrative & male   = 2
construction   & male   = 5
education      & male   = 2
technology     & male   = 15
left           & male   = 7
right          & male   = 17
short          & male   = 1
medium         & male   = 19
tall           & male   = 4

administrative & female = 7
construction   & female = 0
education      & female = 4
technology     & female = 5
left           & female = 2
right          & female = 14
short          & female = 6
medium         & female = 8
tall           & female = 2

Using Naive Bayes with Laplacian smoothing to classify when:
 occupation = education
 dominance = right
 height = tall

Probability of male   = 0.3855
Probability of female = 0.6145

Data case is most likely female

End demo
```

Figure 1 **Naive Bayes Classification Demo**

Code download available at archive.msdn.microsoft.com/mag201302TestRun.

Figure 2 **Naive Bayes Program Structure**

```
using System;
namespace NaiveBayes
{
  class Program
  {
    static Random ran = new Random(25); // Arbitrary

    static void Main(string[] args)
    {
      try
      {
        string[] attributes = new string[] { "occupation", "dominance",
          "height", "sex"};

        string[][] attributeValues = new string[attributes.Length][];
        attributeValues[0] = new string[] { "administrative",
          "construction", "education", "technology" };
        attributeValues[1] = new string[] { "left", "right" };
        attributeValues[2] = new string[] { "short", "medium", "tall" };
        attributeValues[3] = new string[] { "male", "female" };

        double[][] numericAttributeBorders = new double[1][];
        numericAttributeBorders[0] = new double[] { 64.0, 71.0 };

        string[] data = MakeData(40);
        for (int i = 0; i < 4; ++i)
          Console.WriteLine(data[i]);

        string[] binnedData = BinData(data, attributeValues,
          numericAttributeBorders);
        for (int i = 0; i < 4; ++i)
          Console.WriteLine(binnedData[i]);

        int[][][] jointCounts = MakeJointCounts(binnedData, attributes,
          attributeValues);
        int[] dependentCounts = MakeDependentCounts(jointCounts, 2);
        Console.WriteLine("Total male = " + dependentCounts[0]);
        Console.WriteLine("Total female = " + dependentCounts[1]);

        ShowJointCounts(jointCounts, attributeValues);

        string occupation = "education";
        string dominance = "right";
        string height = "tall";

        bool withLaplacian = true;

        Console.WriteLine(" occupation = " + occupation);
        Console.WriteLine(" dominance = " + dominance);
        Console.WriteLine(" height = " + height);

        int c = Classify(occupation, dominance, height, jointCounts,
          dependentCounts, withLaplacian, 3);
        if (c == 0)
          Console.WriteLine("\nData case is most likely male");
        else if (c == 1)
          Console.WriteLine("\nData case is most likely female");

        Console.WriteLine("\nEnd demo\n");
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    } // End Main

    // Methods to create data
    // Method to bin data
    // Method to compute joint counts
    // Helper method to compute partial probabilities
    // Method to classify a data case

  } // End class Program
}
```

the dominance is right and the height is tall. In this example, it turns out the demo determined the probability that the data case is a male is 0.3855 and the probability that the case is female is 0.6145, and so the system concludes the data case is most likely a female.

In the sections that follow I'll first explain exactly how Naive Bayes classification works, walk you through the code in the demo program, and describe how to modify the demo to meet your own needs. This article assumes you have at least beginning programming skills with a C-family language, but doesn't assume you know anything about Naive Bayes classification. The code for the demo program is a bit too long to present in its entirety here, but the complete source is available from the MSDN download site at archive.msdn.microsoft.com/mag201302TestRun.

How Naive Bayes Classification Works

Using the example shown in **Figure 1**, the goal is to predict the sex (male or female) of a person whose occupation is education, who is right-handed and whose height is tall (greater than or equal to 71.0 inches). To do this, we can compute the probability that the person is male given that information, and the probability the person is female given the information, and then predict the sex with the larger probability. Expressed symbolically, we want to know P(male | X), usually read as, "the probability of male given independent variable values X") and P(female | X), where X is (education, right, tall). The term "naive" in Naive Bayes means that all X attributes are assumed to be mathematically independent, which greatly simplifies classification. You can find many online references that explain the rather interesting mathematics behind Naive Bayes classification, but the result is relatively simple. Symbolically:

```
P(male | X) =
  [ P(education | male) * P(right | male) * P(tall | male) * P(male) ] /
    [ PP(male | X) + PP(female | X) ]
```

The term "naive" in Naive Bayes means that all X attributes are assumed to be mathematically independent, which greatly simplifies classification.

Notice the equation is a fraction. The numerator, sometimes loosely called a partial probability, consists of four terms multiplied together. In this article I use the nonstandard notation of PP for a partial probability term. The denominator is the sum of two terms, one of which is the numerator. The first piece to compute is P(education | male), or the probability that a person's occupation is education, given that he is male. This, as it turns out, can be estimated by the count of training cases where occupation is education and sex is male, divided by the number of cases that are male (with any occupation), so:

```
P(education | male ) = count(education & male) / count(male) = 2/24 = 0.0833
```

Figure 3 **Method BinData for Categorizing Height**

```
static string[] BinData(string[] data, string[][] attributeValues,
  double[][] numericAttributeBorders)
{
  string[] result = new string[data.Length];
  string[] tokens;
  double heightAsDouble;
  string heightAsBinnedString;

  for (int i = 0; i < data.Length; ++i)
  {
    tokens = data[i].Split(',');
    heightAsDouble = double.Parse(tokens[2]);
    if (heightAsDouble <= numericAttributeBorders[0][0]) // Short
      heightAsBinnedString = attributeValues[2][0];
    else if (heightAsDouble >= numericAttributeBorders[0][1]) // Tall
      heightAsBinnedString = attributeValues[2][2];
    else
      heightAsBinnedString = attributeValues[2][1]; // Medium

    string s = tokens[0] + "," + tokens[1] + "," + heightAsBinnedString +
      "," + tokens[3];
    result[i] = s;
  }
  return result;
}
```

Using the same logic:

```
P(right | male) = count(right & male) / count(male) = 17/24 = 0.7083
P(tall | male) = count(tall & male) / count(male) = 4/24 = 0.1667
```

The next piece of the puzzle is P(male). In Naive Bayes terminology, this is called a prior. There's some debate about how best to compute priors. On the one hand, we can hypothesize that there's no reason to believe that the presence of males is more or less likely than the presence of females and so assign 0.5 to P(male). On the other hand, we can use the fact that the training data has 24 males and 16 females and estimate a probability of 24/40 = 0.6000 for P(male). I prefer this approach, where priors are estimated using training data.

Figure 4 **Method MakeJointCounts**

```
static int[][][] MakeJointCounts(string[] binnedData, string[] attributes,
  string[][] attributeValues)
{
  int[][][] jointCounts = new int[attributes.Length - 1][][]; // -1 (no sex)

  jointCounts[0] = new int[4][]; // 4 occupations
  jointCounts[1] = new int[2][]; // 2 dominances
  jointCounts[2] = new int[3][]; // 3 heights

  jointCounts[0][0] = new int[2]; // 2 sexes for administrative
  jointCounts[0][1] = new int[2]; // construction
  jointCounts[0][2] = new int[2]; // education
  jointCounts[0][3] = new int[2]; // technology

  jointCounts[1][0] = new int[2]; // left
  jointCounts[1][1] = new int[2]; // right

  jointCounts[2][0] = new int[2]; // short
  jointCounts[2][1] = new int[2]; // medium
  jointCounts[2][2] = new int[2]; // tall

  for (int i = 0; i < binnedData.Length; ++i)
  {
    string[] tokens = binnedData[i].Split(',');

    int occupationIndex = AttributeValueToIndex(0, tokens[0]);
    int dominanceIndex = AttributeValueToIndex(1, tokens[1]);
    int heightIndex = AttributeValueToIndex(2, tokens[2]);
    int sexIndex = AttributeValueToIndex(3, tokens[3]);

    ++jointCounts[0][occupationIndex][sexIndex];
    ++jointCounts[1][dominanceIndex][sexIndex];
    ++jointCounts[2][heightIndex][sexIndex];
  }

  return jointCounts;
}
```

Now, if you refer to the earlier equation for P(male | X), you'll note that it contains the PP(female | X). The bottom sum, PP(male | X) + PP(female | X), is sometimes called the evidence. The pieces for PP(female | X) are computed like so:

```
P(education | female) = count(education & female) / count(female) = 4/16 = 0.2500
P(right | female) = count(right & female) / count(female) = 14/16 = 0.8750
P(tall | female) = count(tall & female) / count(female) = 2/16 = 0.1250
P(female) = 16/40 = 0.4000
```

So the partial probability numerator for P(male | X) is:

```
PP(male | X) = 0.0833 * 0.7083 * 0.1667 * 0.6000 = 0.005903
```

Using the same logic, the partial probability for female given X = (education, right, tall) is:

```
PP(female | X) = 0.2500 * 0.8750 * 0.1250 * 0.4000 = 0.010938
```

And, finally, the overall probabilities of male and female are:

```
P(male | X) = 0.005903 / (0.005903 + 0.010938) = 0.3505
P(female | X) = 0.010938 / (0.005903 + 0.010938) = 0.6495
```

These overall probabilities are sometimes called the posteriors. Because P(female | X) is greater than P(male | X), the system concludes the sex of the unknown person is female. But wait. These two probabilities, 0.3505 and 0.6495, are close to but definitely not the same as the two probabilities, 0.3855 and 0.6145, shown in **Figure 1**. The reason for this discrepancy is that the demo program uses an important optional modification of basic Naive Bayes called Laplacian smoothing.

Laplacian Smoothing

If you refer to **Figure 1**, you'll see that the count of training cases in which the person has occupation = construction and sex = female is 0. In the demo, the X values are (education, right, tall), which doesn't include construction. But suppose X had been (construction, right, tall). In the computation of PP(female | X) it would be necessary to compute P(construction | female) = count(construction & female) / count(female), which would be 0, and which in turn would zero-out the entire partial probability. In short, it's bad when a joint count is 0. The most common technique to avoid this situation is to simply add 1 to all joint counts. This has the feel of a hack but, in fact, has a solid mathematical basis. The technique is called add-one smoothing, which is a specific kind of Laplacian smoothing.

With Laplacian smoothing, if X = (education, right, tall) as in the previous section, P(male | X) and P(female | X) are calculated as follows:

```
P(education | male ) =
count(education & male) + 1 / count(male) + 3 = 3/27 = 0.1111
P(right | male) =
count(right & male) + 1 / count(male) + 3 = 18/27 = 0.6667
P(tall | male) =
count(tall & male) + 1 / count(male) + 3 = 5/27 = 0.1852
P(male) = 24/40 = 0.6000

P(education | female) =
count(education & female) + 1 / count(female) + 3 = 5/19 = 0.2632
P(right | female) =
count(right & female) + 1 / count(female) + 3 = 15/19 = 0.7895
P(tall | female) =
count(tall & female) + 1 / count(female) + 3 = 3/19 = 0.1579
P(female) = 16/40 = 0.4000
```

The partial probabilities are:

```
PP(male | X) = 0.1111 * 0.6667 * 0.1852 * 0.6000 = 0.008230
PP(female | X) = 0.2632 * 0.7895 * 0.1579 * 0.4000 = 0.013121
```

Figure 5 **Method Classify**

```
static int Classify(string occupation, string dominance, string height,
  int[][][] jointCounts, int[] dependentCounts, bool withSmoothing,
  int xClasses)
{
  double partProbMale = PartialProbability("male", occupation, dominance,
    height, jointCounts, dependentCounts, withSmoothing, xClasses);
  double partProbFemale = PartialProbability("female", occupation, dominance,
    height, jointCounts, dependentCounts, withSmoothing, xClasses);
  double evidence = partProbMale + partProbFemale;
  double probMale = partProbMale / evidence;
  double probFemale = partProbFemale / evidence;

  if (probMale > probFemale) return 0;
  else return 1;
}
```

And so the two final probabilities are:

```
P(male | X) = 0.008230 / (0.008230 + 0.013121) = 0.3855
P(female | X) = 0.013121 / (0.008230 + 0.013121) = 0.6145
```

These are the values shown in the screenshot in **Figure 1**. Notice that 1 is added to each joint count but that 3 is added to denominators count(male) and count(female). The 3 is to some extent arbitrary in the sense that Laplacian smoothing doesn't specify any particular value to be used. In this case, it's the number of X attributes (occupation, dominance, height). This is the most common value to add to denominators of partial probabilities in Laplacian smoothing, but you may wish to experiment with other values. The value to add to the denominator is often given the symbol k in math literature on Naive Bayes. Also, notice that the priors, P(male) and P(female), are typically not modified in Naive Bayes Laplacian smoothing.

Overall Program Structure

The demo program shown running in **Figure 1** is a single C# console application. The Main method, with some WriteLine statements removed, is listed in **Figure 2**.

The program begins by setting up the hardcoded X attributes occupation, dominance, and height, and the dependent attribute sex. In some situations you may prefer to scan your existing data source to determine the attributes, especially when the source is a data file with headers or a SQL table with column names. The demo program also specifies the nine categorical X attribute values: (administrative, construction, education, technology) for occupation; (left, right) for dominance; and (short, medium, tall) for height. In this example there are two dependent variable attribute values: (male, female) for sex. Again, you may want to programmatically determine attribute values by scanning your data.

The demo sets up hardcoded boundary values of 64.0 and 71.0 to bin the numeric height values so that values less than or equal to 64.0 are categorized as short; heights between 64.0 and 71.0 are medium; and heights greater than or equal to 71.0 are tall. When binning numeric data for Naive Bayes, the number of boundary values will be one less than the number of categories. In this example, the 64.0 and 71.0 were determined by scanning the training data for minimum and maximum height values (57.0 and 78.0), computing the difference, 21.0, and then computing interval size by dividing by number of height categories, 3, giving 7.0. In most situations, you'll want to the determine boundary values for numeric X attributes programmatically rather than manually.

The demo program calls a helper method MakeData to generate somewhat random training data. MakeData calls helpers MakeSex, MakeOccupation, MakeDominance and MakeHeight. For example, these helpers generate data so that male occupations are more likely to be construction and technology, male dominance is more likely to be right, and male height is most likely to be between 66.0 and 72.0 inches.

The key methods called in Main are BinData to categorize height data; MakeJointCounts to scan binned data and compute the joint counts; MakeDependentCounts to compute total number of males and females; and Classify, which uses joint counts and dependent counts to perform a Naive Bayes classification.

Binning Data

Method BinData is listed in **Figure 3**. The method accepts an array of comma-delimited strings where each string looks like "education,left,67.5,male." In many situations, you'll be reading training data from a text file where each line is a string. The method uses String.Split to parse each string into tokens. Token[2] is the height. It's converted from a string into type double using the double.Parse method. The numeric height is compared against the boundary values until the height's interval is found, and then the corresponding height category as a string is determined. A result string is stitched together using the old tokens, comma delimiters and the new computed-height category string.

> It's not a requirement to bin numeric data when performing Naive Bayes classification.

It's not a requirement to bin numeric data when performing Naive Bayes classification. Naive Bayes can deal with numeric data directly, but those techniques are outside the scope of this article. Binning data has the advantages of simplicity and avoiding the need to make any particular explicit assumptions about the mathematical distribution (such as Gaussian or Poisson) of the data. However, binning essentially loses information and does require you to determine and specify into how many categories to divide the data.

Determining Joint Counts

The key to Naive Bayes classification is computing joint counts. In the demo example, there are nine total independent X attribute values (administrative, construction, … tall) and two dependent attribute values (male, female), so a total of 9 * 2 = 18 joint counts must be computed and stored. My preferred approach is to store joint counts in a three-dimensional array int[][][] jointCounts. The first index indicates the independent X attribute; the second index indicates the independent X attribute value; and the third index indicates the dependent attribute value. For example, jointCounts[0][3][1] means attribute 0 (occupation), attribute value 3 (technology) and sex 1 (female), or in other words the value at jointCounts[0][3]

[1] is the count of training cases where occupation is technology and sex is female. Method MakeJointCounts is listed in **Figure 4**.

The implementation has many hardcoded values to make it easier to understand. For example, these three statements could be replaced by a single for loop that allocates space using Length properties in array attributeValues:

```
jointCounts[0] = new int[4][]; // 4 occupations
jointCounts[1] = new int[2][]; // 2 dominances
jointCounts[2] = new int[3][]; // 3 heights
```

Helper function AttributeValueToIndex accepts an attribute index and an attribute value string and returns the appropriate index. For example, AttributeValueToIndex(2, "medium") returns the index of "medium" in the height attribute, which is 1.

The demo program uses a method MakeDependentCounts to determine the number of male and number of female data cases. There are several ways to do this. If you refer to **Figure 1**, you'll observe that one approach is to add the number of joint counts of any of the three attributes. For example, the number of males is the sum of count(administrative & male), count(construction & male), count(education & male) and count(technology & male):

```
static int[] MakeDependentCounts(int[][][] jointCounts,
  int numDependents)
{
  int[] result = new int[numDependents];
  for (int k = 0; k < numDependents; ++k)
  // Male then female
    for (int j = 0; j < jointCounts[0].Length; ++j)
    // Scanning attribute 0
      result[k] += jointCounts[0][j][k];

  return result;
}
```

Classifying a Data Case

Method Classify, shown in **Figure 5**, is short because it relies on helper methods.

Method Classify accepts the jointCounts and dependentCounts arrays; a Boolean field to indicate whether or not to use Laplacian smoothing; and parameter xClasses, which in this example will be 3 because there are three independent variables (occupation, dominance, height). This parameter could also be inferred from the jointCounts parameter.

Method Classify returns an int that represents the index of the predicted dependent variable. Instead, you might want to return an array of probabilities for each dependent variable. Notice that the classification is based on probMale and probFemale, both of which are computed by dividing partial probabilities by the evidence value. You might want to simply omit the evidence term and just compare the values of the partial probabilities by themselves.

Method Classify returns the index of the dependent variable that has the largest probability. An alternative is to supply a threshold value. For example, suppose probMale is 0.5001 and probFemale is 0.4999. You may wish to consider these values too close to call and return a classification value representing "undetermined."

Method PartialProbability does most of the work for Classify and is listed in **Figure 6**.

Method PartialProbability is mostly hardcoded for clarity. For example, there are four probability pieces, p0, p1, p2 and p3. You can make PartialProbability more general by using an array of probabilities where the size of the array is determined from the jointCounts array.

Figure 6 **Method PartialProbability**

```
static double PartialProbability(string sex, string occupation, string dominance,
  string height, int[][][] jointCounts, int[] dependentCounts,
  bool withSmoothing, int xClasses)
{
  int sexIndex = AttributeValueToIndex(3, sex);
  int occupationIndex = AttributeValueToIndex(0, occupation);
  int dominanceIndex = AttributeValueToIndex(1, dominance);
  int heightIndex = AttributeValueToIndex(2, height);

  int totalMale = dependentCounts[0];
  int totalFemale = dependentCounts[1];
  int totalCases = totalMale + totalFemale;

  int totalToUse = 0;
  if (sex == "male") totalToUse = totalMale;
  else if (sex == "female") totalToUse = totalFemale;

  double p0 = (totalToUse * 1.0) / (totalCases); // Prob male or female
  double p1 = 0.0;
  double p2 = 0.0;
  double p3 = 0.0;

  if (withSmoothing == false)
  {
    p1 = (jointCounts[0][occupationIndex][sexIndex] * 1.0) / totalToUse
    p2 = (jointCounts[1][dominanceIndex][sexIndex] * 1.0) / totalToUse;
    p3 = (jointCounts[2][heightIndex][sexIndex] * 1.0) / totalToUse;
  }
  else if (withSmoothing == true)
  {
    p1 = (jointCounts[0][occupationIndex][sexIndex] + 1) /
     ((totalToUse + xClasses) * 1.0);
    p2 = (jointCounts[1][dominanceIndex][sexIndex] + 1) /
     ((totalToUse + xClasses) * 1.0 ;
    p3 = (jointCounts[2][heightIndex][sexIndex] + 1) /
     ((totalToUse + xClasses) * 1.0);
  }

  //return p0 * p1 * p2 * p3; // Risky if any very small values
  return Math.Exp(Math.Log(p0) + Math.Log(p1) + Math.Log(p2) + Math.Log(p3));
}
```

Notice that instead of returning the product of the four probability pieces, the method returns the equivalent Exp of the sum of the Log of each piece. Using log probabilities is a standard technique in machine-learning algorithms that's used to prevent numeric errors that can occur with very small real numeric values.

Wrapping Up

The example presented here should give you a good foundation for adding Naive Bayes classification features to your .NET applications. Naive Bayes classification is a relatively crude technique, but it does have several advantages over more-sophisticated alternatives such as neural network classification, logistic regression classification and support vector machine classification. Naive Bayes is simple, relatively easy to implement and scales well to very large data sets. And Naive Bayes easily extends to multinomial classification problems—those with three or more dependent variables. ■

Dr. James McCaffrey *works for Volt Information Sciences Inc., where he manages technical training for software engineers working at the Microsoft 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 Microsoft technical expert for reviewing this article: *Rich Caruana*

TECHMENTOR CONFERENCES

15 YEARS OF IT EDUCATION
15TH ANNIVERSARY

Celebrating 15 years of IT education, TechMentor returns in 2013 with immediately usable training that will keep you relevant in the workforce. Get inside the IT classroom and learn how you can build a more productive IT environment.

Choose Your Campus

TECHMENTOR CONFERENCES 2013

REGISTRATION OPEN

THE IT CLASSROOM
REAL-WORLD EDUCATION

ORLANDO
March 4 - 8 | Buena Vista Palace

TECHMENTOR CONFERENCES 2013

THE IT CLASSROOM

LAS VEGAS
Sept 30-Oct 4 | The Tropicana

TECHMENTOREVENTS.COM

SUPPORTED BY
Redmond MAGAZINE
Redmond Channel Partner
VIRTUALIZATION REVIEW

PRODUCED BY
1105 MEDIA

TED NEWARD

.NET Collections, Part 2: Working with C5

Welcome back.

In the first part of this series, I looked briefly at the Copenhagen Comprehensive Collection Classes for C# (C5) library, a set of classes designed to supplement (if not replace) the System.Collections classes that ship with the Microsoft .NET Framework runtime library. There's a fair amount of overlap between the two libraries, partly because C5 wants to follow many of the same idioms that the .NET Framework Class Library (FCL) uses, and partly because there are only so many ways one can reasonably represent a particular collection type. (It's hard to imagine an indexed collection—such as a Dictionary or a List—not supporting the language syntax for indexed properties: the "[]" operator in C# and the "()" operator in Visual Basic.) Where the FCL collections are utilitarian, however, the C5 collections go a step or two beyond that, and that's where we want to spend our time.

> Where the FCL collections are utilitarian, however, the C5 collections go a step or two beyond that.

(Note that there's also very likely some performance differences between the two libraries that proponents or critics of each will be quick to point out—the C5 collection manual discusses some of the performance implications, for example. That said, I eschew most performance benchmarks on the grounds that, generally, all a benchmark proves is that for one particular case or set of cases, somebody got one of the two to run faster than the other, which doesn't really say whether that will hold true for all cases between the two. This doesn't mean all benchmarks are useless, just that the context matters to the benchmark. Readers are strongly encouraged to take their own particular scenarios, turn them into a benchmark and have a shootout between the two, just to see if there's a marked difference in those particular cases.)

Implementations

First of all, let's take a quick look at the different collection implementations that C5 provides. Again, as we discussed last time, developers using C5 shouldn't generally worry about the implementation in use except when deciding which implementation to create—the rest of the time, the collection should be referenced by interface type. (For a description of the interface types, see the previous column in the series at msdn.microsoft.com/magazine/jj883961, or the C5 documentation at bit.ly/UcOcZH.) Here are the implementations:

- CircularQueue<T> implements both IQueue<T> and IStack<T> to provide either the first-in-first-out semantics of IQueue<T> (via Enqueue and Dequeue) or the last-in-first-out semantics of IStack<T> (via Push and Pop), backed by a linked list. It grows in capacity as needed.
- ArrayList<T> implements IList<T>, IStack<T> and IQueue<T>, backed by an array.
- LinkedList<T> implements IList<T>, IStack<T> and IQueue<T>, using a doubly linked list of nodes.
- HashedArrayList<T> implements IList<T>, backed by an array, but also maintains a hash table internally to efficiently find the position of an item within the list. Also, it doesn't allow duplicates in the list (because duplicates would screw up the hash table lookup).
- HashedLinkedList<T> implements IList<T>, backed by a linked list, and like its array-backed cousin, it uses an internal hash table to optimize lookups.
- WrappedArray<T> implements IList<T>, wrapping around a single-dimensional array. The advantage of this class is that it simply "decorates" the array, making it far faster to obtain C5 functionality, as opposed to copying the elements out of the array and into an ArrayList<T>.
- SortedArray<T> implements IIndexedSorted<T>, which means the collection can be indexed as well as sorted—we'll get to this in a second. It keeps its items sorted and doesn't allow duplicates.

Figure 1 **Creating Views on a Collection**

```
[TestMethod]
public void GettingStarted()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy" });

  // Print item 1 ("Roosevelt") in the list
  Assert.AreEqual("Roosevelt", names[1]);
  Console.WriteLine(names[1]);

  // Create a list view comprising post-WW2 presidents
  IList<String> postWWII = names.View(2, 3);

  // Print item 2 ("Kennedy") in the view
  Assert.AreEqual("Kennedy", postWWII[2]);
}
```

Figure 2 **Views Are Potentially Mutable**

```
[TestMethod]
public void ViewExploration()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Washington", "Adams", "Jefferson",
      "Hoover", "Roosevelt", "Truman",
      "Eisenhower", "Kennedy" });

  IList<String> postWWII = names.View(4, names.Count - 4);
  Assert.AreEqual(postWWII.Count, 4);
  IList<String> preWWII = names.View(0, 5);
  Assert.AreEqual(preWWII.Count, 5);
  Assert.AreEqual("Washington", preWWII[0]);

  names.Insert(3, "Jackson");
  Assert.AreEqual("Jackson", names[3]);
  Assert.AreEqual("Jackson", preWWII[3]);
}
```

- TreeSet<T> implements IIndexedSorted<T> and IPersistedSorted<T> and is backed by a balanced red-black tree, which is great for insertion, removal and sorting. Like all sets, it doesn't allow duplicates.
- TreeBag<T> implements IIndexedSorted<T> and IPersistedSorted<T>, is backed by a balanced red-black tree, but is essentially a "bag" (sometimes called a "multiset"), meaning it allows duplicates.
- HashSet<T> implements IExtensible<T>, and backs the set (meaning no duplicates) by a hash table with linear chaining. This means lookups will be fast, modifications less so.
- HashBag<T> implements IExtensible<T>, and backs the bag (meaning duplicates are allowed) by a hash table with linear chaining, again making lookups fast.
- IntervalHeap<T> implements IPriorityQueue<T>, using an interval heap stored as an array of pairs, making it efficient to pull from either the "max" or "min" end of the priority queue.

There are a few more implementations, and the C5 manual and docs have more details if you're interested. However, aside from the performance implications, the critical thing to know is which implementations implement which interfaces, so that you can have a good idea of each when it's time to choose one to instantiate. (You can always switch it around to a different implementation later, assuming you follow the C5 design guideline of always referencing the collections by the interfaces rather than their implementation types.)

Figure 3 **Guarded (Immutable) Collections**

```
public void IWannaBePresidentToo(IList<String> presidents)
{
  presidents.Add("Neward");
}
[TestMethod]
public void NeverModifiedCollection()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Hoover", "Roosevelt", "Truman","Eisenhower", "Kennedy" });
  try
  {
    IWannaBePresidentToo(new GuardedList<String>(names));
  }
  catch (Exception x)
  {
    // This is expected! Should be a ReadOnlyException
  }

  Assert.IsFalse(names.Contains("Neward"));
 }
```

Functionality

If C5 were just a larger collection of collection implementations, it would be interesting, but probably not enough to warrant significant interest or discussion. Fortunately, it offers a few new features to developers that deserve discussion.

Views One of the interesting little tidbits of the C5 library is the notion of "views": subcollections of elements from the source collection that are, in fact, not copies but backed by the original collection. This was actually what the code from the previous column did, in the exploration test. See **Figure 1** for how to create views on a collection.

The view is backed by the original list, which means that if the original list changes for whatever reason, the view on it is also affected. See **Figure 2** to see how views are potentially mutable.

As this test shows, changing the underlying list ("names") means that the views defined on it (in this case, the "preWWII" view) also find their contents changing, so that now the first element in the view is "Washington," instead of "Hoover."

However, when possible, C5 will preserve the sanctity of the view; so, for example, if the insertion occurs at the front of the collection (where C5 can insert it without changing the contents of the "preWWII" view), then the view's contents remain unchanged:

```
[TestMethod]
public void ViewUnchangingExploration()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy" });

  IList<String> preWWII = names.View(0, 2);
  Assert.AreEqual(preWWII.Count, 2);

  names.InsertFirst("Jackson");
  Assert.AreEqual("Jackson", names[0]);
  Assert.AreEqual("Hoover", preWWII[0]);
}
```

Immutable (Guarded) Collections With the rise of functional concepts and programming styles, a lot of emphasis has swung to immutable data and immutable objects, largely because immutable objects offer a lot of benefits vis-à-vis concurrency and parallel

Figure 4 **"When I Become President ..."**

```
[TestMethod]
public void InaugurationDay()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy" });

  names.ItemsAdded +=
    delegate (Object c, ItemCountEventArgs<string> args)
  {
    testContextInstance.WriteLine(
      "Happy Inauguration Day, {0}!", args.Item);
  };
  names.Add("Neward");

  Assert.IsTrue(names.Contains("Neward"));
}
```

programming, but also because many developers find immutable objects easier to understand and reason about. Corollary to that concept, then, follows the concept of immutable collections—the idea that regardless of whether the objects inside the collection are immutable, the collection itself is fixed and unable to change (add or remove) the elements in the collection. (Note: You can see a preview of immutable collections released on NuGet in the MSDN Base Class Library (BCL) blog at bit.ly/12AXD78.)

> All collections in C5 offer the ability to hang delegates off the collection.

Within C5, immutable collections are handled by instantiating "wrapper" collections around the collection containing the data of interest; these collections are "Guarded" collections and are used in classic Decorator pattern style:

```
public void ViewImmutableExploration()
{
  IList<String> names = new ArrayList<String>();
  names.AddAll(new String[]
    { "Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy" });
  names = new GuardedList<String>(names);

  IList<String> preWWII = names.View(0, 2);
  Assert.AreEqual("Hoover", preWWII[0]);

  names.InsertFirst("Washington");
  Assert.AreEqual("Washington", names[0]);
}
```

If anyone tries to write code that adds or removes elements from the list, C5 quickly disabuses said developer of the idea: An exception is thrown as soon as any of the "modifying" methods (Add, Insert, InsertFirst and so on) are called.

This offers a pretty powerful opportunity, by the way. In the previous column, I mentioned that one of the key design points that went into C5 is the idea that collections should only be used through interfaces. Assuming developers using C5 carry that design idea forward, it now becomes really simple to ensure that a collection is never modified by a method to which it is passed (see **Figure 3**).

Again, when the IWannaBePresidentToo method tries to modify the collection passed in (which, arguably, is bad design on the part of the programmer who wrote it, but unfortunately there's a lot of that kind of code out there), an exception is thrown.

By the way, should you prefer that the collection not throw an exception and instead silently fail the modification (which I think is too subtle, but some developers may need that functionality), it's relatively easy to put together your own version of GuardedArray<T> that doesn't throw.

Events Sometimes, modifications to collections are, in fact, what you want to allow—only you want to know when a collection is modified. Granted, you could spin up a thread and have it spin indefinitely over the collection, comparing the contents to the previous iteration's contents, but not only is this a horrible waste of CPU resources, but it's a pain to write and maintain, making it probably the worst possible design solution—and certainly a sight poorer than simply using a collection that supports events natively. All collections in C5 offer the ability to hang delegates off the collection, to be invoked when certain operations take place against the collection (see **Figure 4**).

Of course, the event handler can be written as a lambda; it's just a little more descriptive to show you the actual argument types. The first argument is—as is canon—the collection itself.

Just a NuGet Away

No part of C5 couldn't be built around the .NET FCL (aside from the emphasis on interfaces, which the FCL supports, but doesn't really endorse that strongly, it seems), but the nice thing about C5 is that it's done, it's tested and it's just a NuGet "Install-Package" away.

Happy coding! ■

Ted Neward *is a principal with Neward & Associates LLC. He has written more than 100 articles and authored and co-authored 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: *Immo Landwerth*

RACHEL APPEL

Create Windows Store Apps with HTML5 and JavaScript

Not only has Bill Gates' dream of a computer on every desk and in every home come to fruition, but the advent of devices such as the Surface tablet has taken his dream even further. In addition to the Surface, there has been an explosion of new consumer-oriented devices in every form factor possible. In other words, computers are everywhere.

Consider that on those computers are more than 1 billion existing Windows installations worldwide, with 300 million Windows 7 licenses sold each year in the previous two years. Combine the current, upgradable Windows installation base with the rapidly growing market of Windows 8 devices such as the Surface and you have the formula for monetization success. This is Windows reimagined—the unparalleled opportunity for you to make money by publishing apps in the Windows Store.

The Platform, Language and Toolset for Creating Windows Store Apps

In order to create a Windows Store app, you need Windows 8, Visual Studio 2012 and any SDKs specific to the requirements of your app, such as the Windows Live SDK or Bing Maps SDK. This minimal system setup and configuration makes app development on Windows 8 easy, from installation to deployment.

Once you've installed the requisite software, it's time to move onto choosing a language. If your development background lies in the Microsoft stack as a Microsoft .NET Framework developer writing Windows Forms, Windows Presentation Foundation (WPF) or Silverlight apps with C# or Visual Basic, then creating Windows Store apps with XAML and C# or Visual Basic is the path of least resistance. C++ developers can also use C++ as the compiled language with XAML as its GUI companion.

If you're a Web developer—including ASP.NET—you can apply your existing knowledge of open standard HTML5, JavaScript and CSS3 directly to Windows Store app development. Web developers may continue to use many popular third-party JavaScript libraries such as jQuery or Knockout. For this article, I'll use JavaScript as the language of choice.

No matter where your development background lies, the barrier to entry is low when developing native Windows 8 apps. This is because the Windows Runtime (WinRT) is a platform that contains APIs that sit on top of the Windows core services, as illustrated in **Figure 1**.

The WinRT APIs give you access to everything Windows 8 has to offer, including APIs for hardware such as built-in webcams, geolocation, light sensors and accelerometers. Of course, platform fundamentals such as memory management, authentication, globalization and asynchronous processing—as well as Windows Store app features such as search, share and communications—are also readily available. There are even APIs for manipulating audio and video; however, if you're writing JavaScript apps, HTML5 <audio> and <video> elements work great. You can browse the complete API on the "API reference for Windows Store apps" page at bit.ly/ZCwcJE.

Tenets of a Windows Store App

Windows Store apps run as fully immersive, full-screen experiences that deliver streamlined content to the user, without the app or its commands getting in the way of the user. Windows Store apps offer a clean, straightforward visualization of data that draws the user's attention to the content.

Windows Store apps do things traditional Windows or Web apps couldn't do before, such as sharing, searching and communicating with each other in an easy and unified way, using elements of the Windows Runtime called contracts as liaisons between apps.

Great UX is a key facet of Windows Store app development, from presentation and layout to navigation and app performance.

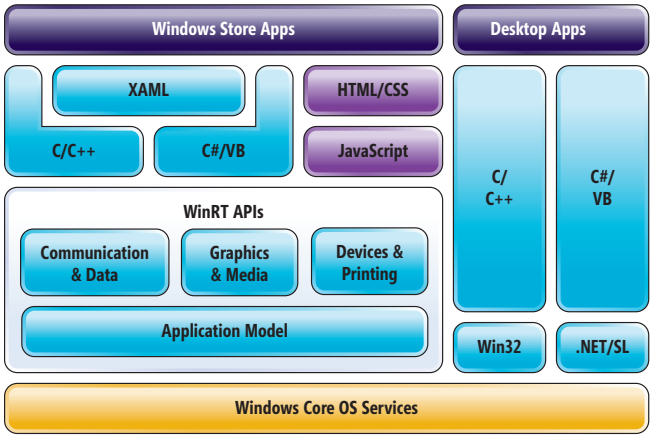


Figure 1 **Architecture of Windows Store Apps**

Code download available at archive.msdn.microsoft.com/mag201302ModernApps.

Users expect consistency between apps and between apps and the OS. Windows Store apps are all about UX, and employing design principles—such as using a consistent font, the Windows UI silhouette and a scalable grid system—enhance the user's comfort level when using your app. This consistency is carried throughout both apps and Windows 8 itself.

Ensuring both touch and mouse input work reliably and consistently is important because users now have more ways to interact with their computing devices in the form of mice, pens, touch, cameras and sensors.

> While many apps will work nicely as a Windows Store app, not every app is a good candidate.

Because battery-powered devices with processors such as ARM are becoming a major part of the computing landscape, Windows 8 must manage the overall and per-app memory in a strict fashion to enable a fast and fluid experience even when resources are low. Windows 8 apps enjoy a straightforward and simple process lifecycle to ensure the best experience.

While many apps will work nicely as a Windows Store app, not every app is a good candidate. For example, while Visual Studio itself embraces many modern UI design principles, its purpose is to allow the user to execute commands (in other words, programming). That's not a good fit for a Windows Store app.

Visual Studio 2012 Windows Store App Project Templates

Visual Studio 2012 introduced a set of new templates for Windows Store app development in C#, Visual Basic, C++ and JavaScript. Regardless of language, the following template styles are available:

- **Blank:** A bare-bones template with the minimum files required to build a Windows Store app.
- **Grid:** A template that displays a grid that uses the Windows 8 UI silhouette along with template code for several features, including navigation and snapped-view support (more on this later).
- **Split:** A template that displays a list of items and item details in a two-column view, making it easy for the user to switch quickly among the items.
- **Fixed:** A Blank template that uses a ViewBox object in the default.html page. A ViewBox is a WinRT object used in games.
- **Navigation:** A template with a Blank project structure plus navigation and a set of basic app assets (that is, home.html, home.js and home.css) under the /home directory.

Because the Grid template contains code that touts many great Windows 8 features such as support for snapped view, screen scaling and navigation, it's the perfect way to get started writing Windows Store apps.

After you create a new JavaScript Grid project template, examining its structure reveals a project full of standard Web file types—such as .html, .css and .js files—organized in folders under the project's root. You can then debug and run a Windows Store app by pressing F5 or selecting Start Debugging from the Debug menu.

In the Windows Store app templates, default.html is the starting page for a Windows Store app and has a companion script file, /js/default.js, which contains basic process lifecycle management code. As with any other HTML file, default.html has code you might expect, including script references and new HTML5 semantic markup that defines the page structure. The following code fragment lives inside the default.html <body> tag and uses WinJS controls for navigation and to load the groupedItems.html page:

```
<div id="contenthost"
  data-win-control="Application.PageControlNavigator"
  data-win-options=
  "{home: '/pages/groupedItems/groupedItems.html'}"></div>
```

The data-* attributes are the HTML5 way to apply custom code or behavior to an HTML element, and in Windows Store app development, data-win-* attributes usually refer to Windows JavaScript controls. Windows JavaScript controls are built-in WinRT components that you apply to HTML elements to enhance or modify their behavior or style. Data-win-* attributes are prevalent in Windows Store JavaScript apps, especially when data binding.

Figure 2 **Data Binding the List Object to HTML Elements with WinJS Controls**

```
<!-- These templates are used to display each
  item in the ListView declared below. -->
<div class="headertemplate" data-win-control="WinJS.Binding.Template">
  <button class="group-header win-type-x-large win-type-interactive"
    data-win-bind="groupKey: key"
    onclick="Application.navigator.pageControl.navigateToGroup(
      event.srcElement.groupKey)"
      role="link" tabindex="-1" type="button">
    <span class="group-title win-type-ellipsis"
      data-win-bind="textContent: title"></span>
    <span class="group-chevron"></span>
  </button>
</div>
<div class="itemtemplate" data-win-control="WinJS.Binding.Template" >
  <div id="myitem" class="item"
    data-win-bind="style.background: color">
    <img class="item-image" src="#"
      data-win-bind="src: image; alt: title" />
    <div class="item-overlay">
    <h2 class="item-title" data-win-bind="innerText: message"></h2>
      <h6 class="item-subtitle"
        data-win-bind="textContent: eventDate"></h6>
    </div>
  </div>
</div>
<!-- The content that will be loaded and displayed. -->
<div class="fragment groupeditemspage">
  <header aria-label="Header content" role="banner">
    <button class="win-backbutton" aria-label="Back"
       disabled type="button"></button>
    <h1 class="titlearea win-type-ellipsis">
      <span class="pagetitle">How long until...</span>
    </h1>
  </header>
    <section aria-label="Main content" role="main">
      <div id="listView" class="groupeditemslist"
        aria-label="List of groups"
        data-win-control="WinJS.UI.ListView"
        data-win-options="{  selectionMode: 'multi',
        tapBehavior:'toggleSelect' }"></div>
    </section>
</div>
```

Data Access in Windows Store Apps

As part of the Grid template, a file named data.js in the /js folder contains code that builds a data set of arrays as well as functions for grouping and manipulating the data. The data.js file also contains sample data that you should replace with your own. In this article, I'll use data for a countdown app that shows the number of days remaining until an event such as a holiday or vacation.

> Live tiles are called "live" for a reason, and that's because you can dynamically display information and images in them.

In the data.js file you can find the only // TODO comment near the beginning of the file. Replace the code under the comment with your own, so the code looks something like the following code snippet, which makes an XMLHttpRequest call to retrieve JSON data, then also creates the data set, including dynamic properties such as the daysToGo and message fields:

```
var list = new WinJS.Binding.List();
...
WinJS.xhr({ url: "data.json" }).then(function (xhr) {
  var items = JSON.parse(xhr.responseText);
  items.forEach(function (item) {
    item.daysToGo = Math.floor(
      (Date.parse(item.eventDate) - new Date()) / 86400000);
    item.message = item.daysToGo + " days until " + item.title;
    if (item.daysToGo >= 0) {
      list.push(item);
    };
  })
})
```

In the beginning of data.js is a line of code that instantiates a WinJS.Binding.List object aptly named list, and the preceding code pushes individual items onto this List.

The List object enables binding between JSON data or JavaScript arrays and HTML elements. Once the *list* variable is populated with data, use binding expressions in HTML markup to bind the List members to HTML elements.

When you read JSON data with a call to JSON.parse, the names in name/value pairs match properties of JavaScript objects at run time. The following JSON data shows how the JSON structure maps to members of the items variable in the preceding code snippet. The key, title, eventDate, image, color and group fields all map to the item object's properties:

```
[{"key":"1","group":{"key":"group1","title":"Important Dates"},"title":"Rachel's
Birthday","eventDate":"01/13/2013","image":"/images/birthday.png","color":"#6666FF"},

{"key":"2","group":{"key":"group1","title":"Important Dates"},"title":"Ada
Lovelace Day","eventDate":"10/16/2013","image":"/images/ada.jpg","color":"#fff"},

{"key":"3","group":{"key":"group2","title":"Holidays"},"title":"Christmas
","eventDate":"12/25/2013","image":"/images/tree.png","color":"#ef0d0d"},

{"key":"4","group":{"key":"group3","title":"School"},"title":"School
Ends","eventDate":"6/10/2013","image":"/images/schoolbus.png","color":"#fff"},

{"key":"5","group":{"key":"group2","title":"Holidays"},"title":"Thanksgiving",
"eventDate":"11/29/2012","image":"/images/thanksgiving.png","color":"#FFCC00"},

{"key":"6","group":{"key":"group2","title":"Holidays"},"title":"New Year's Day",
"eventDate":"1/1/2013","image":"/images/celebrate.png","color":"#f8baba"}]
```

Now that you've loaded the data, you need to ensure the List object is bound to the correct HTML elements. Modifications to the /pages/groupedItems/groupedItems.html page in **Figure 2** shows data binding in action.

Each HTML element in **Figure 2** that contains a data-win-bind attribute has a binding expression that matches a property name of the *item* variable from the preceding code snippet, so all you need to do is make sure that the binding expressions match the names of the fields. Don't forget to ensure that you also modify the binding expressions in the groupedDetail.html and itemDetails.html pages so correct data will show when a user navigates to them.

Running the project in the Windows Simulator yields results similar to that in **Figure 3**. (You can learn more about using the simulator at bit.ly/M1nWOY.)

As you can see, you can simply replace the code from the Visual Studio template for quick data access. However, projects are often quite large or complex, making maintenance difficult. If this is the case, then use the Model-View-ViewModel (MVVM) pattern to make maintenance easier. This pattern is extremely well-documented on the Web.

While your app now works, it's time take advantage of the many great Windows 8 features that can make your app stand out in the crowd.

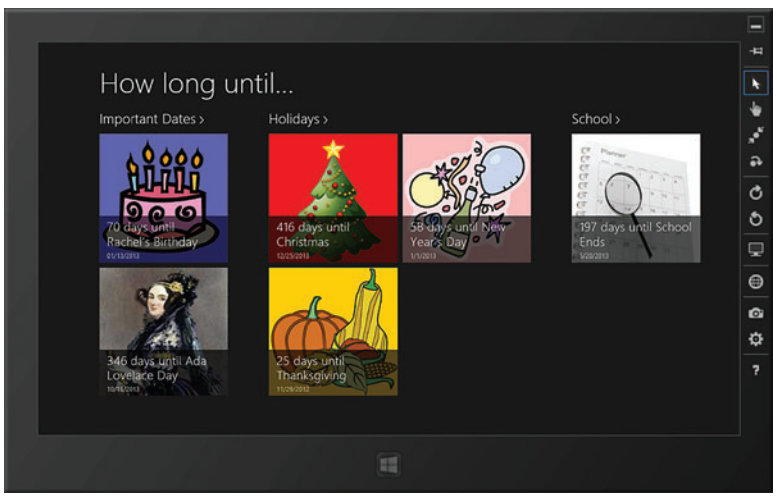


Figure 3 **Replace the Sample Data to Make a Basic App**

Branding Your Windows Store App

Considering that the focal point of Windows 8 is the Start page, it makes sense to start branding there. The Start page is filled with live tiles, and they aren't just a bunch of square icons, either. Instead, they're the best way to show off and attract users to your app. Live tiles are called "live" for a reason, and that's because you can dynamically display information and images in them, making your app even more attractive.

Windows Store apps require three separate tile images with the following pixel dimensions:

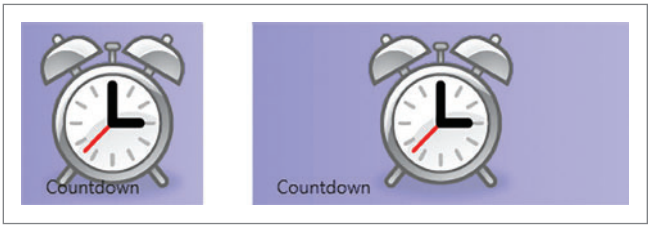


Figure 4 **Countdown App Standard and Wide Tiles**

- Logo: 150 x 150 (standard tile)
- Wide Logo: 150 x 310 (wide tile)
- Small Logo: 30 x 30 (this shows only in app lists in the store)

The images can be any popular image format, and those with transparent backgrounds work best. Opening the package.appxmanifest file from the project's root reveals the configuration palette, where you can select the tile images and set the background colors. **Figure 4** illustrates both a standard and wide tile.

When you're setting up the tiles is a good time to configure the splash screen by selecting just an image and background color—no code is used. Although tiles and splash screens are important factors in branding your app, you can do many more things to brand and polish your app, which you can read about at bit.ly/M4HYmL.

Windows 8 'Must-Have' Features for Your App

While your app might work at this point, there are many new features and APIs in the Windows 8 app ecosystem that you can tap into to really make your app stand out. I'll briefly discuss each.

AppBar An essential feature for every app is the AppBar, which is a WinJS control found in default.html. Normally, the AppBar stays out of sight, but when users right-click or swipe from the top or bottom of the screen, the AppBar displays as a bar across the bottom of the screen. **Figure 5** shows the markup for an AppBar containing three buttons as well as their corresponding event listeners.

Global AppBar commands should be located on the right side of the AppBar, while contextual commands should go on the left. Style the AppBar with CSS, as it's only a .

Snapped View Windows Store apps can run in full screen or a mode called snapped view that happens when the user "sticks" the app to the left or right side of the screen. Because the app now has less screen real estate, your app should display only necessary data.

Figure 5 **An AppBar with Buttons for Adding, Deleting and Exporting Data**

```
// AppBar markup in default.html
<div id="appbar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'addItem', label:'Add',
    icon:'add', section:'global'}" type="button"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'exportData', label:'Save',
    icon:'save', section:'global'}" type="button"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'deleteItem', label:'Delete',
    icon:'delete', section:'selection'}" type="button"></button>
</div>
// Script in groupedItems.js
document.getElementById("exportData").addEventListener("click", Data.exportData);
document.getElementById("addItem").addEventListener("click", this.addItem);
document.getElementById("deleteItem").addEventListener("click", this.deleteItem);
```

Because snapped-view support is built into the Grid template, you need to verify that the data displays nicely while snapped, showing pertinent and readable information. The AppBar also works while the app is snapped, so that also might need style adjustments.

Semantic Zoom This new touch-friendly feature of Windows 8 is a way to aggregate large amounts of data in a single, easy-to-digest view. Users invoke Semantic Zoom in the following ways:

- Touch: Pinch gesture
- Mouse: Ctrl+Scroll Wheel
- Keyboard: Ctrl - and Ctrl +

Semantic Zoom is more about visualizing data in a meaningful way that assists with navigation than simply exposing a zoomed view of it. If there's a lot of data, it's better for the user to have a bird's-eye view rather than having to scroll through an overload of information. Consider how to best present the data so it's the most meaningful.

Search and Share Searching and sharing data between apps are core aspects of modern apps. Users can now search across multiple apps at one time and then share the data they find. Or your app can register itself as a share target and accept data that users share from other Windows Store apps. Never before has app-to-app communication been so straightforward and consistent.

Picker Controls These are traditional Windows controls that have been updated for a modern UI—such as the File Open Picker or File Save Picker—or print settings dialogs that have been staples of Windows apps for many versions.

Media Because Windows Store apps built with JavaScript fully support HTML5, the <audio> and <video> elements work the same way as they do in ordinary Web pages.

Toast Notifications Toast notifications are a way to provide a momentary message to the user, regardless of whether the app is in use or not. The most popular forms of toast notifications are e-mail alert pop-ups and text messages on phones. Toast messages can contain text and images and can serve as another way to attract users to your app. You can post the same notifications to the Windows 8 lock screen for a quick glimpse of any waiting messages.

Generation App

To recap, Windows 8 is Windows reimagined, sporting some of the biggest changes in the OS since Windows 95, in an unprecedented market. The built-in Visual Studio project templates enable you to get started publishing moneymaking apps easier and faster than ever in the largest market for app creators.

There's not enough space here to discuss all of the awesome features you could and should use in your Windows Store app, so I highly recommend that you check out the Generation App program (bit.ly/W8GenAppDev). It guides you through the process of building a Windows Store (or Windows Phone) app in 30 days, offering free technical and design consultations and assistance along with exclusive tips and resources. ■

Rachel Appel *is a developer evangelist at Microsoft New York City. Reach her via her Web site at rachelappel.com or by e-mail at rachel.appel@microsoft.com. You can also follow her latest updates on Twitter at twitter.com/rachelappel.*

Thanks to the following technical expert for reviewing this article: *Ian LeGrow*

CHARLES PETZOLD

Constructing Audio Oscillators for Windows 8

I've been making electronic music instruments as a hobby for about 35 years now. I started in the late 1970s wiring up CMOS and TTL chips, and much later went the software route—first with the Multimedia Extensions to Windows in 1991 and more recently with the NAudio library for Windows Presentation Foundation (WPF), and the MediaStreamSource class in Silverlight and Windows Phone 7. Just last year, I devoted a couple installments of my Touch & Go column to applications for Windows Phone that play sound and music.

I should probably be jaded by this time, and perhaps reluctant to explore yet another sound-generation API. But I'm not, because I think Windows 8 is probably the best Windows platform yet for making musical instruments. Windows 8 combines a high-performance audio API—the XAudio2 component of DirectX—with touchscreens on handheld tablets. This combination offers much potential, and I'm particularly interested in exploring how touch can be exploited as a subtle and intimate interface to a musical instrument implemented entirely in software.

Oscillators, Samples and Frequency

At the heart of the sound-generation facility of any music synthesizer are multiple oscillators, so called because they generate a more or less periodic oscillating waveform at a particular frequency and volume. In generating sounds for music, oscillators that create unvarying periodic waveforms usually sound rather boring. More interesting oscillators incorporate vibrato, tremolo or changing timbres, and they're only roughly periodic.

A program that wishes to create oscillators using XAudio2 begins by calling the XAudio2Create function. This provides an object that implements the IXAudio2 interface. From that object you can call CreateMasteringVoice just once to obtain an instance of IXAudio2MasteringVoice, which functions as the main audio mixer. Only one IXAudio2MasteringVoice exists at any time. In contrast, you'll generally call CreateSourceVoice multiple times to create multiple instances of the IXAudio2SourceVoice interface. Each of these IXAudio2SourceVoice instances can function as an independent oscillator. Combine multiple oscillators for a multiphonic instrument, an ensemble or a full orchestra.

An IXAudio2SourceVoice object generates sound by creating and submitting buffers containing a sequence of numbers that describe a waveform. These numbers are often called samples. They're often 16 bits wide (the standard for CD audio), and they come at a constant rate—usually 44,100 Hz (also the standard for CD audio) or thereabouts. This technique has the fancy name Pulse Code Modulation, or PCM.

Although this sequence of samples can describe a very complex waveform, often a synthesizer generates a fairly simple stream of samples—most commonly a square wave, a triangle wave or a sawtooth—with a periodicity corresponding to the waveform's frequency (perceived as pitch) and an average amplitude that is perceived as volume.

Figure 1 **Much of the SawtoothOscillator1 Class**

```
SawtoothOscillator1::SawtoothOscillator1(IXAudio2* pXAudio2)
{
  // Create a source voice
  WAVEFORMATEX waveFormat;
  waveFormat.wFormatTag = WAVE_FORMAT_PCM;
  waveFormat.nChannels = 1;
  waveFormat.nSamplesPerSec = 44100;
  waveFormat.nAvgBytesPerSec = 44100 * 2;
  waveFormat.nBlockAlign = 2;
  waveFormat.wBitsPerSample = 16;
  waveFormat.cbSize = 0;

  HRESULT hr = pXAudio2->CreateSourceVoice(&pSourceVoice, &waveFormat,
                                           0, XAUDIO2_MAX_FREQ_RATIO);
  if (FAILED(hr))
    throw ref new COMException(hr, "CreateSourceVoice failure");

  // Initialize the waveform buffer
  for (int sample = 0; sample < BUFFER_LENGTH; sample++)
    waveformBuffer[sample] =
      (short)(65535 * sample / BUFFER_LENGTH - 32768);

  // Submit the waveform buffer
  XAUDIO2_BUFFER buffer = {0};
  buffer.AudioBytes = 2 * BUFFER_LENGTH;
  buffer.pAudioData = (byte *)waveformBuffer;
  buffer.Flags = XAUDIO2_END_OF_STREAM;
  buffer.PlayBegin = 0;
  buffer.PlayLength = BUFFER_LENGTH;
  buffer.LoopBegin = 0;
  buffer.LoopLength = BUFFER_LENGTH;
  buffer.LoopCount = XAUDIO2_LOOP_INFINITE;

  hr = pSourceVoice->SubmitSourceBuffer(&buffer);

  if (FAILED(hr))
    throw ref new COMException(hr, "SubmitSourceBuffer failure");

  // Start the voice playing
  pSourceVoice->Start();
}

void SawtoothOscillator1::SetFrequency(float freq)
{
  pSourceVoice->SetFrequencyRatio(freq / BASE_FREQ);
}

void SawtoothOscillator1::SetAmplitude(float amp)
{
  pSourceVoice->SetVolume(amp);
}
```

Code download available at archive.msdn.microsoft.com/mag201302DXF.

Figure 2 **OnVoiceProcessingPassStart in SawtoothOscillator2**

```
void _stdcall SawtoothOscillator2::OnVoiceProcessingPassStart(UINT32 bytesRequired)
{
  if (bytesRequired == 0)
      return;

  int startIndex = index;
  int endIndex = startIndex + bytesRequired / 2;

  if (endIndex <= BUFFER_LENGTH)
  {
    FillAndSubmit(startIndex, endIndex - startIndex);
  }
  else
  {
    FillAndSubmit(startIndex, BUFFER_LENGTH - startIndex);
    FillAndSubmit(0, endIndex % BUFFER_LENGTH);
  }
  index = (index + bytesRequired / 2) % BUFFER_LENGTH;
}

void SawtoothOscillator2::FillAndSubmit(int startIndex, int count)
{
  for (int i = startIndex; i < startIndex + count; i++)
  {
    pWaveformBuffer[i] = (short)(angle / WAVEFORM_LENGTH - 32768);
    angle = (angle + angleIncrement) % (WAVEFORM_LENGTH * 65536);
  }

  XAUDIO2_BUFFER buffer = {0};
  buffer.AudioBytes = 2 * BUFFER_LENGTH;
  buffer.pAudioData = (byte *)pWaveformBuffer;
  buffer.Flags = 0;
  buffer.PlayBegin = startIndex;
  buffer.PlayLength = count;

  HRESULT hr = pSourceVoice->SubmitSourceBuffer(&buffer);

  if (FAILED(hr))
    throw ref new COMException(hr, "SubmitSourceBuffer");
}
```

For example, if the sample rate is 44,100 Hz, and every cycle of 100 samples has values that get progressively larger, then smaller, then negative, and back to zero, the frequency of the resultant sound is 44,100 divided by 100, or 441 Hz—a frequency close to the perceptual center of the audible range for humans. (A frequency of 440 Hz is the A above middle C and is used as a tuning standard.)

The IXAudio2SourceVoice interface inherits a method named SetVolume from IXAudio2Voice and defines a method of its own named SetFrequencyRatio. I was particularly intrigued by this latter method, because it seemed to provide a way to create an oscillator that generates a particular periodic waveform at a variable frequency with a minimum of fuss.

Figure 1 shows the bulk of a class named SawtoothOscillator1 that implements this technique. Although I use familiar 16-bit integer samples for defining the waveform, internally XAudio2 uses 32-bit floating point samples. For performance-critical applications, you'll probably want to explore the performance differences between integer and floating-point.

In the header file, a base frequency is set that divides cleanly into the 44,100 sampling rate. From that, a buffer size can be calculated that is the length of a single cycle of a waveform of that frequency:

```
static const int BASE_FREQ = 441;
static const int BUFFER_LENGTH = (44100 / BASE_FREQ);
```

Also in the header file is the definition of that buffer as a field:

```
short waveformBuffer[BUFFER_LENGTH];
```

After creating the IXAudio2SourceVoice object, the SawtoothOscillator1 constructor fills up a buffer with one cycle of a sawtooth waveform—a simple waveform that goes from an amplitude of -32,768 to an amplitude of 32,767. This buffer is submitted to the IXAudio2SourceVoice with instructions that it should be repeated forever.

Without any further code, this is an oscillator that plays a 441 Hz sawtooth wave forever. That's great, but it's not very versatile. To give SawtoothOscillator1 a bit more versatility, I've also included a SetFrequency method. The argument to this is a frequency that the class uses to call SetFrequencyRatio. The value passed to SetFrequencyRatio can range from float values of XAUDIO2_MIN_FREQ_RATIO (or 1/1,024.0) up to a maximum value earlier specified as an argument to CreateSourceVoice. I used XAUDIO2_MAX_FREQ_RATIO (or 1,024.0) for that argument. The range of human hearing—about 20 Hz to 20,000 Hz—is well within the bounds defined by those two constants applied to the base frequency of 441.

Buffers and Callbacks

I must confess that I was initially somewhat skeptical of the SetFrequencyRatio method. Digitally increasing and decreasing the frequency of a waveform is not a trivial task. I felt obliged to compare the results with a waveform generated algorithmically. This is the impetus behind the OscillatorCompare project, which is among the downloadable code for this column.

The OscillatorCompare project includes the SawtoothOscillator1 class I've already described as well as a SawtoothOscillator2 class. This second class has a SetFrequency method that controls how the class dynamically generates the samples that define the waveform. This waveform is continuously constructed in a buffer and submitted in real time to the IXAudio2SourceVoice object in response to callbacks.

A class can receive callbacks from IXAudio2SourceVoice by implementing the IXAudio2VoiceCallback interface. An instance of the class that implements this interface is then passed as an argument to the CreateSourceVoice method. The SawtoothOscillator2 class implements this interface itself and it passes its own instance to CreateSourceVoice, also indicating that it won't be making use of SetFrequencyRatio:

```
pXAudio2->CreateSourceVoice(&pSourceVoice, &waveFormat,
        XAUDIO2_VOICE_NOPITCH, 1.0f,
        this);
```

A class that implements IXAudio2VoiceCallback can use the OnBufferStart method to be notified when it's time to submit a new buffer of waveform data. Generally when using OnBufferStart to keep waveform data up-to-date, you'll want to maintain a pair

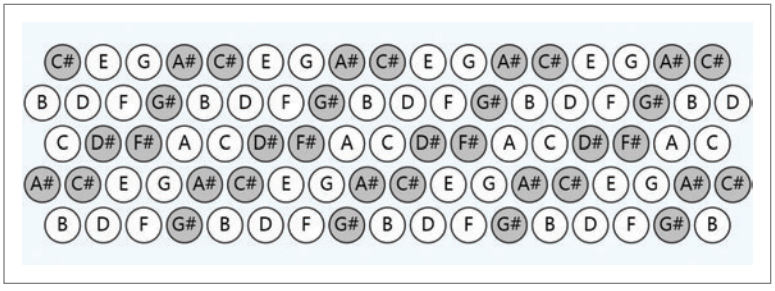


Figure 3 **The ChromaticButtonKeyboard Program**

of buffers and alternate them. This is probably the best solution if you're obtaining audio data from another source, such as an audio file. The goal is to not let the audio processor become "starved." Keeping a buffer ahead of the processing helps prevent starvation, but does not guarantee it.

But I gravitated toward another method defined by IXAudio-2VoiceCallback—OnVoiceProcessingPassStart. Unless you're working with very small buffers, generally OnVoiceProcessingPassStart is called more frequently than OnBufferStart and indicates when a chunk of audio data is about to be processed and how many bytes are needed. In the XAudio2 documentation, this callback method is promoted as the one with the lowest latency, which is often highly desirable for interactive electronic music instruments. You don't want a delay between pressing a key and hearing the note!

The SawtoothOscillator2 header file defines two constants:

```
static const int BUFFER_LENGTH = 1024;
static const int WAVEFORM_LENGTH = 8192;
```

The first constant is the length of the buffer used to submit waveform data. Here it functions as a circular buffer. Calls to the OnVoiceProcessingPassStart method request a particular number of bytes. The method responds by putting those bytes in the buffer (starting from where it left off the last time) and calling SubmitSourceBuffer just for that updated segment of the buffer. You want this buffer to be sufficiently large so your program code isn't overwriting the part of the buffer still being played in the background.

It turns out that for a voice with a sample rate of 44,100 Hz, calls to OnVoiceProcessingPassStart always request 882 bytes, or 441 16-bit samples. In other words, OnVoiceProcessingPassStart is called at the constant rate of 100 times per second, or every 10 ms. Although not documented, this 10 ms duration can be treated as an XAudio2 audio processing "quantum," and it's a good figure to keep in mind. Consequently, the code you write for this method can't dawdle. Avoid API calls and runtime library calls.

Figure 4 **The IsPressedChanged Handler for the Key Instances**

```
void MainPage::OnKeyIsPressedChanged(Object^ sender, bool isPressed)
{
  Key^ key = dynamic_cast<Key^>(sender);
  int keyNum = (int)key->Tag;

  if (isPressed)
  {
    if (availableOscillators.size() > 0)
    {
      SawtoothOscillator* pOscillator = availableOscillators.back();
      availableOscillators.pop_back();

      double freq = 440 * pow(2, (keyNum % 1000 - 69) / 12.0);
      pOscillator->SetFrequency((float)freq);
      pOscillator->SetAmplitude(1.0f / NUM_OSCILLATORS);
      playingOscillators[keyNum] = pOscillator;
    }
  }
  else
  {
    SawtoothOscillator * pOscillator = playingOscillators[keyNum];

    if (pOscillator != nullptr)
    {
      pOscillator->SetAmplitude(0);
      availableOscillators.push_back(pOscillator);
      playingOscillators.erase(keyNum);
    }
  }
}
```

The second constant is the length of a single cycle of the desired waveform. It could be the size of an array containing the samples of that waveform, but in SawtoothOscillator2 it's used only for calculations.

The SetFrequency method in SawtoothOscillator2 uses that constant to calculate an angle increment that's proportional to the desired frequency of the waveform:

```
angleIncrement = (int)(65536.0
                * WAVEFORM_LENGTH
                * freq / 44100.0);
```

Although angleIncrement is an integer, it's treated as though it comprises integral and fractional words. This is the value used to determine each successive sample of the waveform.

For example, suppose the argument to SetFrequency is 440 Hz. The angleIncrement is calculated as 5,356,535. In hexadecimal, this is 0x51BBF7, which is treated as an integer of 0x51 (or 81 decimal), with a fractional part of 0xBBF7, equivalent to 0.734. If the complete cycle of a waveform is 8,192 bytes and you use only the integer part and skip 81 bytes for each sample, the resultant frequency is about 436.05 Hz. (That's 44,100 times 81 divided by 8,192.) If you skip 82 bytes, the resultant frequency is 441.43 Hz. You want something between these two frequencies.

This is why a fractional part also needs to enter the calculation. The whole thing would probably be easier in floating point, and floating point might even be faster on some modern processors, but **Figure 2** shows a more "traditional" integer-only approach. Notice that only the updated section of the circular buffer is specified with each call to SubmitSourceBuffer.

SawtoothOscillator1 and SawtoothOscillator2 can be compared side-by-side in the OscillatorCompare program. MainPage has two pairs of Slider controls to change the frequency and volume of each oscillator. The Slider control for the frequency generates only integer values ranging from 24 to 132. I borrowed these values from the codes used in the Musical Instrument Digital Interface (MIDI) standard to represent pitches. The value of 24 corresponds to the C three octaves below middle-C, which is called C 1 (C in octave 1) in scientific pitch notation and has a frequency of about 32.7 Hz. The value of 132 corresponds to C 10, six octaves above middle-C, and a frequency of about 16,744 Hz. A tooltip converter on these sliders displays the current value in both scientific pitch notation and the frequency equivalent.

As I experimented with these two oscillators, I couldn't hear a difference. I also installed a software oscilloscope on another computer to visually examine the resultant waveforms, and I couldn't see any difference either. This indicates to me that the SetFrequency-Ratio method is implemented intelligently, which of course we should expect in a system as sophisticated as DirectX. I suspect that interpolations are being performed on resampled waveform data to shift the frequency. If you're nervous, you can set the BASE_FREQ very low—for example, to 20 Hz—and the class will generate a detailed waveform consisting of 2,205 samples. You can also experiment with a high value: For example, 8,820 Hz will cause a waveform of just five samples to be generated! To be sure, this has a somewhat different sound because the interpolated waveform lies somewhere between a sawtooth and a triangle wave, but the resultant waveform is still smooth without "jaggies."

This is not to imply that everything works hunky dory. With either sawtooth oscillator, the top couple octaves get rather chaotic. The sampling of the waveform tends to emit high and low frequency overtones of a sort I've heard before, and which I plan to investigate more fully in the future.

Keep the Volume Down!

The SetVolume method defined by IXAudio2Voice and inherited by IXAudio2SourceVoice is documented as a floating-point multiplier that can be set to values ranging from -2^{24} to 2^{24}, which equals 16,777,216.

In real life, however, you'll probably want to keep the volume on an IXAudio2SourceVoice object to a value between 0 and 1. The 0 value corresponds to silence and 1 corresponds to no gain or attenuation. Keep in mind that whatever the source of the waveform associated with an IXAudio2SourceVoice—whether it's being generated algorithmically or originates in an audio file—it probably has 16-bit samples that quite possibly come close to the minimum and maximum values of -32,768 and 32,767. If you try to amplify those waveforms with a volume level greater than 1, the samples will exceed the width of a 16-bit integer and will be clipped at the minimum and maximum values. Distortion and noise will result.

This becomes critical when you start combining multiple IXAudio2SourceVoice instances. The waveforms of these multiple instances are mixed by being added together. If you allow each of these instances to have a volume of 1, the sum of the voices could very well result in samples that exceed the size of the 16-bit integers. This might happen sporadically—resulting only in intermittent distortion—or chronically, resulting in a real mess.

When using multiple IXAudio2SourceVoice instances that generate full 16-bit-wide waveforms, one safety measure is setting the volume of each oscillator to 1 divided by the number of voices. That guarantees that the sum never exceeds a 16-bit value. An overall volume adjustment can also be made via the mastering voice. You might also want to look into the XAudio2CreateVolumeMeter function, which lets you create an audio processing object that can help monitor volume for debugging purposes.

Our First Musical Instrument

It's common for musical instruments on tablets to have a piano-style keyboard, but I've been intrigued recently by a type of button keyboard found on accordions such as the Russian bayan (which I'm familiar with from the work of Russian composer Sofia Gubaidulina). Because each key is a button rather than a long lever, many more keys can be packed within the limited space of the tablet screen, as shown in **Figure 3**.

The bottom two rows duplicate the keys on the top two rows and are provided to ease the fingering of common chords and melodic sequences. Otherwise, each group of 12 keys in the top three rows provide all the notes of the octave, generally ascending from left to right. The total range here is four octaves, which is about twice what you'd get with a piano keyboard of the same size.

A real bayan has an additional octave, but I couldn't fit it in without making the buttons too small. The source code allows you to set constants to try out that extra octave, or to eliminate another octave and make the buttons even larger.

Because I can't claim that this program sounds like any instrument that exists in the real world, I simply called it ChromaticButtonKeyboard. The keys are instances of a custom control named Key that derives from ContentControl but performs some touch processing to maintain an IsPressed property and generate an IsPressedChanged event. The difference between the touch handling in this control and the touch handling in an ordinary button (which also has an IsPressed property) is noticeable when you sweep your finger across the keyboard: A standard button will set the IsPressed property to true only if the finger press occurs on the surface of the button, while this custom Key control considers the key to be pressed if a finger sweeps in from the side.

The program creates six instances of a SawtoothOscillator class that's virtually identical to the SawtoothOscillator1 class from the earlier project. If your touchscreen supports it, you can play six simultaneous notes. There are no callbacks and the oscillator frequency is controlled by calls to the SetFrequencyRatio method.

To keep track of which oscillators are available and which oscillators are playing, the MainPage.xaml.h file defines two standard collection objects as fields:

```
std::vector<SawtoothOscillator *> availableOscillators;
std::map<int, SawtoothOscillator *> playingOscillators;
```

Originally, each Key object had its Tag property set to the MIDI note code I discussed earlier. That's how the IsPressedChanged handler determines what key is being pressed, and what frequency to calculate. That MIDI code was also used as the map key for the playingOscillators collection. It worked fine until I played a note from the bottom two rows that duplicated a note already playing, which resulted in a duplicate key and an exception. I easily solved that problem by incorporating a value into the Tag property indicating the row in which the key is located: The Tag now equals the MIDI note code plus 1,000 times the row number.

Figure 4 shows the IsPressedChanged handler for the Key instances. When a key is pressed, an oscillator is removed from the availableOscillators collection, given a frequency and non-zero volume, and put into the playingOscillators collection. When a key is released, the oscillator is given a zero volume and moved back to availableOscillators.

That's about as simple as a multi-voice instrument can be, and of course it's flawed: Sounds should not be turned off and on like a switch. The volume should glide up rapidly but smoothly when a note starts, and fall back when it stops. Many real instruments also have a change in volume and timbre as the note progresses. There's still plenty of room for enhancements.

But considering the simplicity of the code, it works surprisingly well and is very responsive. If you compile the program for the ARM processor, you can deploy it on the ARM-based Microsoft Surface and walk around cradling the untethered tablet in one arm while playing on it with the other hand, which I must say is a bit of a thrill. ■

Charles Petzold *is a longtime contributor to* MSDN Magazine *and the author of "Programming Windows, 6th edition" (O'Reilly Media, 2012), a book about writing applications for Windows 8. His Web site is charlespetzold.com.*

Thanks to the following technical experts for reviewing this article: *Tom Mathews and Thomas Petchel*

What's Up, Doc?

The Internet hasn't notably cracked the health-care industry yet. It's nibbled around the edges a little bit—for example, I can renew prescriptions online instead of phoning them in—but it hasn't fundamentally changed the business model or the relationships between the players as in other industries. I'll begin my fourth year as *MSDN Magazine*'s resident Diogenes (bit.ly/Xr3x) by predicting how it soon will.

The forces currently tearing apart the structure of higher education are also gathering in the medical industry. Health care in the United States consumes about $3 trillion per year, approximately one-sixth of the U.S. gross domestic product. The providers are partying like it's 2006 and costs are spiraling. The population is aging and getting fatter; the boomer bulge is making its way through the demographic snake. Disruptive technologies are ready to rock. Unstoppable forces are slamming into immovable objects. Something is about to give.

Why hasn't it happened yet? Partly because of the medical establishment's famed inertia. Consider Ignaz Semmelweis, the 19th century Viennese obstetrician who lowered maternal mortality by 90 percent, simply by insisting that doctors wash their hands before examining childbirth patients. As reward for this spectacular improvement, his colleagues threw him into an insane asylum where he quickly died (see bit.ly/SO3jd4).

Perhaps faster technological change in medicine had to wait until kids who grew up with the Internet had finished medical school. That's starting to happen now. A young doctor who attended a class I taught on Microsoft HealthVault told me: "My kids' babysitter makes better use of the Internet in her business than we do here at [a major teaching hospital]. I'm here to learn how to fix that." The tipping point where these guys accumulate enough power to change things is not far off.

Last month I explored how massive open online courses (MOOCs) are successful in education because they combine higher quality with lower cost. This virtuous combination is now approaching for the medical industry. Consider your child waking up saying, "I don't feel good." Instead of schlepping to the doctor's office, suppose you could talk to a nurse on a Skype video link.

The world's finest doctors would work out the diagnostic protocols for a sick kid, and a software wizard would walk the nurse through it. She would have trained extensively on this specific scenario, using excellent simulators, so she'd be an expert on it. The programs would continuously update the diagnostic probabilities based on the latest results seen in the local area, making her more current and precise than an unaided pediatrician today. Yet her time would cost far less than that of an MD.

What's more, you would own a small instrument to measure and transmit your child's temperature, blood pressure, pulse oxygenation and other vital signs. A camera on the instrument would transmit pictures of the throat or ear canal or skin rashes, and a microphone would transmit breath sounds and heartbeats. All this is under development, with prototypes already emerging and first commercial releases within the year (see econ.st/X5mq3e). Perhaps algorithms could compare the pictures and sounds you transmit to every other captured sample. Doctors wouldn't misdiagnose diseases such as measles (which is rare in the United States) because they had never seen a case.

Patients with viral infections would be told to stay home, keep warm, take Tylenol and call back if they didn't improve. Patients with bacterial infections or more severe symptoms would have prescriptions transmitted to a pharmacy for delivery that day. You wouldn't have to take the whole day off from work to drive your kid to the doctor and exchange germs with everyone in the waiting room.

Patients with more serious conditions, or conditions that can't be evaluated over the wire, would get appointments with the doctor that afternoon. Each doctor's time would be far better utilized as well—mornings for follow-ups, afternoons for new cases from the Web nurses.

Just as the education industry will still need mentor classes for advanced topics, the medical industry will always need specialists and surgeons for when people get really sick. But the mass of day-to-day grunt work will be automated faster than anyone imagines, in the same way and for the same reasons as the teaching of freshman calculus is being transformed today.

The medical industry's dam hasn't yet cracked the way the education industry's has. But there's far more force building up behind the medical dam. The burst will be all the more spectacular when it comes, with concomitantly larger profit opportunities for developers and companies who are thinking forward. Call me if you'd like to discuss it. ■

DAVID S. PLATT *teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.*


Thu
Fri
Sat
30
31
1
6
7
8
12
13
14
15
Save the Date
Final QA
19
20
21
22
26
27
28
29
3
4
5
6
2012
8
58
AM
Wed
Thu
7 Fri
18 Sat
19 Sun
$1M
$800K
$600K
$400K
$200K
$0
Sep 3, 2012
Sep 4, 2012
Sep 5, 2012
Sep 6 2012
Sep 7 2012
TOTALS
Media
Appliances
Games
100s of controls for
HTML5 & JavaScript
WPF & Silverlight
Windows Forms
Windows Phone
Windows 8
100s of UI controls for all .NET platforms including grids, charts, reports & schedulers
Visual Studio 2012 support
Windows 8 Studios for WinRT XAML & WinJS
New Modern UI themes
ComponentOne®
Studio® Enterprise
ComponentOne
a division of GrapeCity®
Download your free trial @
componentone.com/se
© 2013 GrapeCity, inc. All rights reserved. All other product and brand names are trademarks and/or registered trademarks of ther respective holders.