THE MICROSOFT JOURNAL FOR DEVELOPERS

MARCH 2013 VOL 28 NO 3

msdn magazine

XAML with DirectX and C++.....34

Using XAML with DirectX and C++ in Windows Store Apps
Doug Erickson **34**

Exploring the JavaScript API for Office: Data Access and Events
Stephen Oliver and Eric Schmidt **48**

Best Practices in Asynchronous Programming
Stephen Cleary **56**

Migrating ASP.NET Web Forms to the MVC Pattern with the ASP.NET Web API
Peter Vogel **62**

Moving Your Applications to Windows Azure
Alex Homer **68**

Data Clustering Using Naive Bayes Inference
James McCaffrey **74**

COLUMNS

WINDOWS WITH C++
Rendering in a Desktop Application with Direct2D
Kenny Kerr, page 8

DATA POINTS
Playing with the EF6 Alpha
Julie Lerman, page 16

WINDOWS AZURE INSIDER
Real-World Scenarios for Node.js in Windows Azure
Bruno Terkaly and Ricardo Villalobos, page 26

THE WORKING PROGRAMMER
Noda Time
Ted Neward, page 80

MODERN APPS
Data Access and Storage Options in Windows Store Apps
Rachel Appel, page 84

DON'T GET ME STARTED
Grieving
David Platt, page 88

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

MARCH 2013 VOLUME 28 NUMBER 3

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

Developing Apps for Office

The past two issues of *MSDN Magazine* have included feature articles focused on developing applications for Microsoft Office 2013. Last month's lead feature ("Exploring the New JavaScript API for Office") detailed the object model hierarchy of the JavaScript API for Office and explored the object model's asynchronous pattern.

In this issue, authors Stephen Oliver and Eric Schmidt, both programming writers in the Office Division at Microsoft, shift their sights to handling data in apps for Office ("Exploring the JavaScript API for Office: Data Access and Events"). They show how developers can get and set selection data and how to get all of the file data. They also look at the events in the JavaScript API for Office and how to code against them.

The apps for Office platform represents an important change in the way apps are developed for the Office suite. The new platform leverages Web connectivity and standard technologies such as HTML5, XML, CSS3, JavaScript and even server-side technologies like ASP.NET. In essence, an app for Office is a Web page that's hosted inside an Office client application such as Word or Excel, and can extend the functionality of a workbook, presentation, project, e-mail message, or appointment.

Developers can continue to build Office extensions using existing tools and platforms such as Visual Studio Tools for Office (VSTO) and Visual Basic for Applications (VBA), both for older and current versions of Office. However, these apps won't leverage the Web technologies in apps for Office, and aren't eligible for distribution via the Office Store. On the flip side, don't expect to deploy your new apps for Office to older versions of the suite. The new apps are currently compatible with Office 2013 and Office 365 (bit.ly/WRbKkb).

We're already seeing the first apps for Office in the wild. At the end of January Microsoft released Bing Apps for Office (binged.it/XVCGPT), a collection of five free apps powered by the Bing search engine: Bing Finance for Office, Bing Maps for Office, Bing Image Search for Office, Bing News Search for Office and Bing Dictionary for Office. The apps work with Office 2013 and Office 365.

I checked in with Oliver and Schmidt about their experience with apps for Office and the new JavaScript API for Office. Oliver, a Microsoft Certified Professional Developer who writes developer documentation for Excel Services and Word Automation Services, says he's most impressed by the way apps for Office lets developers integrate Web assets and resources into Office applications.

"I'm attracted to the concept inherent in the JavaScript API for Office—that of bringing the Web to the Office application. And since a wide range of Web programming technologies are available to you in this new model, you can use familiar technologies like simple REST calls to bring data to or from the Web right from within your Office application," he says.

Schmidt jokes that he "giggled like a child" when he saw how easy it was, with Internet Explorer 10, to use HTML5 features in an app, such as add an input element with a placeholder attribute. He also says upgrading apps for Office is incredibly easy, as developers need only republish HTML, JavaScript and CSS files to the Web server.

"I also have a special fondness for bindings," Schmidt adds. "Since task pane and content apps travel with the files that they're inserted into, we needed a way for an app to retain a reference to a specific region in the document. Bindings allow us to do that."

So what advice do the authors have for developers intrigued by the new opportunities presented with Office 2013?

"I'd say, jump in and start playing with the API," says Oliver. "While the spectrum of apps you can develop is fairly wide—from the relatively simple app that just grabs some data out of the document to a more-sophisticated app that pushes/pulls data to/from a back-end system—I think it's super easy for a developer new to the platform to get in and explore the API and get a feel for what they might be able to do with it."

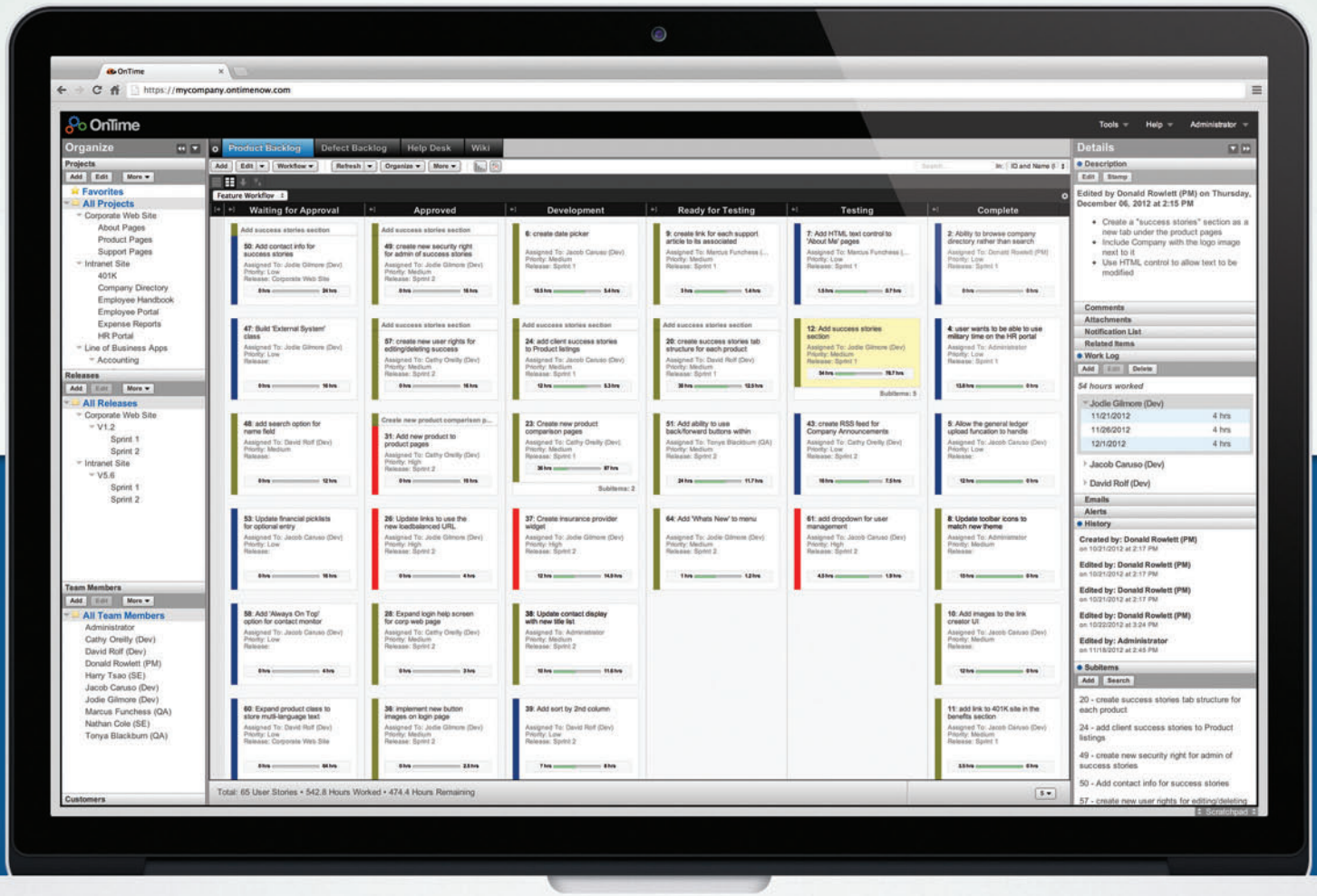
He suggests that developers check out the "Napa" Office 365 Development Tools Web site (bit.ly/Pn2JNr), which makes it easy to start exploring the API. Oh, and one more thing:

"Hey, I'm a doc guy," Oliver says, "so I encourage those getting started to take a look at our documentation (msdn.microsoft.com/library/jj220060) and give us feedback on where we can improve it."

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


RACKSPACE®
just
OPEN-SOURCED
THE CLOUD

KENNY KERR

Rendering in a Desktop Application with Direct2D

In my last column, I showed you how easy it actually is to create a desktop application with C++ without any library or framework. In fact, if you were feeling particularly masochistic, you could write an entire desktop application from within your WinMain function as I've done in **Figure 1**. Of course, that approach simply doesn't scale.

I also showed how the Active Template Library (ATL) provides a nice C++ abstraction for hiding much of this machinery and how the Windows Template Library (WTL) takes this even further, primarily for applications heavily invested in the USER and GDI approaches to application development (see my February column at msdn.microsoft.com/magazine/jj891018).

> The future of application rendering in Windows is hardware-accelerated Direct3D, but that really is impractical to work with directly if all you want to do is render a two-dimensional application or game.

The future of application rendering in Windows is hardware-accelerated Direct3D, but that really is impractical to work with directly if all you want to do is render a two-dimensional application or game. That's where Direct2D comes in. I introduced Direct2D briefly when it was first announced a few years back, but I'm going to spend the next few months taking a much closer look at Direct2D development. Check out my June 2009 column, "Introducing Direct2D" (msdn.microsoft.com/magazine/dd861344), for an introduction to the architecture and fundamentals of Direct2D.

One of the key design underpinnings of Direct2D is that it focuses on rendering and leaves the other aspects of Windows application development to you or other libraries that you might employ. Although Direct2D was designed to render in a desktop window, it's up to you to actually provide this window and optimize it for Direct2D rendering. So this month, I'm going to focus on the unique relationship between Direct2D and the desktop application window. You can do many things to optimize the window handling and rendering process. You want to reduce unnecessary painting and avoid flicker, and just provide the best possible experience for the user. Of course, you'll also want to provide a manageable framework within which to develop your application. I'll be tackling these issues here.

The Desktop Window

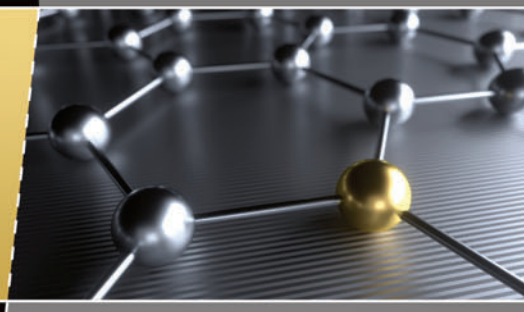
In the ATL example last month, I gave the example of a window class deriving from the ATL CWindowImpl class template. Everything is nicely contained within the application's window class. However, what ends up happening is that a lot of window and rendering plumbing ends up interspersed with the window's application-specific rendering and event handling. To solve this problem, I tend to push as much of this boilerplate code as is

Figure 1 **The Masochist's Window**

```
int __stdcall wWinMain(HINSTANCE module, HINSTANCE, PWSTR, int)
{
  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
  {
    if (WM_DESTROY == message)
    {
      PostQuitMessage(0);
      return 0;
    }

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

  RegisterClass(&wc);

  CreateWindow(wc.lpszClassName, L"Awesome?!",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, nullptr, nullptr, module, nullptr);

  MSG message;
  BOOL result;

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

Powerful Tools for Developers

New! V4.6



Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents programmatically.
- Fast and lightweight 32 and 64-bit components for .Net, COM and WinRT applications.
- Create, fill-out and annotate PDF forms.

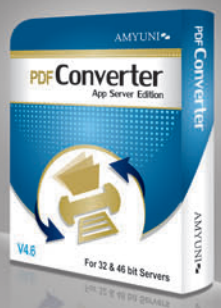
Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package.
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
- Includes our Microsoft certified PDF Converter printer driver.
- Export PDF documents into other formats such as JPeg, Word, Excel or Silverlight.

Advanced HTML to PDF & XAML

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

High Performance PDF Printer Driver



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2008 and Windows 8.
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language.
- Easy licensing and deployment to fit system administrator's requirements.

Windows 7 | CERTIFIED FOR Windows Server 2008 | Ready for Windows 8

AMYUNI

All development tools available at

www.amyuni.com

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

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

practical into a base class, using compile-time polymorphism to reach up to the application's window class when this base class needs its attention. This approach is used quite a lot by ATL and WTL, so why not extend that for your own classes?

Figure 2 illustrates this separation. The base class is the DesktopWindow class template. The template parameter gives the base class the ability to call up to the concrete class without the use of virtual functions. In this case, it's using this technique to hide a bunch of render-specific pre- and post-processing while calling up to the application's window to perform the actual drawing operations. I'll expand on the DesktopWindow class template in a moment, but first its window class registration needs a bit of work.

Optimizing the Window Class

One of the realities of the Windows API for desktop applications is that it was designed to simplify rendering with traditional USER and GDI resources. Some of these "conveniences" need to be disabled to allow Direct2D to take over, to avoid unnecessary painting leading to unsightly flicker. Other of these defaults must also be tweaked to work in a way that better suits Direct2D rendering. Much of this can be achieved by changing the window class information before it's registered, but you may have noticed that ATL hides this from the programmer. Fortunately, there's still a way to achieve this.

One of the realities of the Windows API for desktop applications is that it was designed to simplify rendering with traditional USER and GDI resources.

Last month, I showed how the Windows API expects a window class structure to be registered before a window is created based on its specification. One of the attributes of a window class is its background brush. Windows uses this GDI brush to clear the window's client area before the window begins painting. This was convenient in the days of USER and GDI but is unnecessary and the cause of some flicker for Direct2D applications. A simple way to avoid this is by setting the background brush handle in the window class structure to a nullptr. If you develop on a relatively fast Windows 7 or Windows 8 computer, you might think this is unnecessary because you don't notice any flicker. That's just because the modern Windows desktop is composed so efficiently on the Graphics Processing Unit (GPU) that it's hard to pick it up. However, it's easy enough to strain the rendering pipeline to exaggerate the effect that you might experience on slower machines. If the window's background brush is white and you then paint the window's client area with a contrasting black brush, by adding a little sleep delay—anywhere between 10 ms and 100 ms—you'll have no trouble picking up the striking flicker. So how to avoid it?

Figure 2 **The Desktop Window**

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

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

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

    Render();

    EndPaint(&ps);
    return 0;
  }

  void Render()
  {
    ...
    static_cast<T *>(this)->Draw();
    ...
  }

    ...
};

struct SampleWindow : DesktopWindow<SampleWindow>
{
  void Draw()
  {
    ...
  }
};
```

As I mentioned, if your window class registration lacks a background brush, then Windows won't have any brush with which to clear your window. However, you may have noticed in the ATL examples that the window class registration is completely hidden. A common solution is to handle the WM_ERASEBKGND message that default window processing handles—courtesy of the DefWindowProc function—by painting the window's client area with the window class background brush. If you handle this message, returning true, then no painting occurs. This is a reasonable solution, as this message is sent to a window regardless of whether the window class has a valid background brush or not. Another solution is to just avoid this no-op handler and remove the background brush from the window class in the first place. Fortunately, ATL makes it relatively simple to override this part of window creation. During creation, ATL calls the GetWndClassInfo method on the window to get this window class information. You can provide your own implementation of this method, but ATL provides a handy macro that implements it for you:

```
DECLARE_WND_CLASS_EX(nullptr, CS_HREDRAW | CS_VREDRAW, -1);
```

The last argument for this macro is meant to be a brush constant, but the -1 value tricks it into clearing this attribute of the window

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

Get your 30-day free trial today:
www.telerik.com/all-in-one

telerik

Figure 3 **The DesktopWindow Run Method**

```
int Run()
{
  D2D1_FACTORY_OPTIONS fo = {};

  #ifdef DEBUG
  fo.debugLevel = D2D1_DEBUG_LEVEL_INFORMATION;
  #endif

  HR(D2D1CreateFactory(D2D1_FACTORY_TYPE_SINGLE_THREADED,
                       fo,
                       m_factory.GetAddressOf()));

  static_cast<T *>(this)->CreateDeviceIndependentResources();

  VERIFY(__super::Create(nullptr, nullptr, L"Direct2D"));

  MSG message;
  BOOL result;

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

  return static_cast<int>(message.wParam);
}
```

Figure 4 **The DesktopWindow Render Method**

```
void Render()
{
  if (!m_target)
  {
    RECT rect;
    VERIFY(GetClientRect(&rect));

    auto size = SizeU(rect.right, rect.bottom);
    HR(m_factory->CreateHwndRenderTarget(RenderTargetProperties(),
      HwndRenderTargetProperties(m_hWnd, size),
      m_target.GetAddressOf()));

    static_cast<T *>(this)->CreateDeviceResources();
  }

  if (!(D2D1_WINDOW_STATE_OCCLUDED & m_target->CheckWindowState()))
  {
    m_target->BeginDraw();

    static_cast<T *>(this)->Draw();

    if (D2DERR_RECREATE_TARGET == m_target->EndDraw())
    {
      m_target.Reset();
    }
  }
}
```

Figure 5 **Resizing the Render Target**

```
MESSAGE_HANDLER(WM_SIZE, SizeHandler)

LRESULT SizeHandler(UINT, WPARAM, LPARAM lparam, BOOL &)
{
  if (m_target)
  {
    if (S_OK != m_target->Resize(SizeU(LOWORD(lparam),
      HIWORD(lparam))))
    {
      m_target.Reset();
    }
  }

  return 0;
}
```

class structure. A surefire way to determine whether the window's background has been erased is to check the PAINTSTRUCT filled in by the BeginPaint function inside your WM_PAINT handler. If its fErase member is false, then you know that Windows has cleared your window's background, or at least that some code responded to the WM_ERASEBKGND message and purported to clear it. If the WM_ERASEBKGND message handler doesn't or isn't able to clear the background, then it's up to the WM_PAINT message handler to do so. However, here we can employ Direct2D to completely take over the rendering of the window's client area and avoid this double painting. Just be sure to call the EndPaint function, assuring Windows that you did indeed paint your window, otherwise Windows will continue to pester you with an unnecessary stream of WM_PAINT messages. This, of course, would hurt your application's performance and increase overall power consumption.

The other aspect of the window class information that deserves our attention is the window class styles. This is, in fact, what the second argument to the preceding macro is for. The CS_HREDRAW and CS_VREDRAW styles cause the window to be invalidated every time the window is resized both vertically and horizontally. This certainly isn't necessary. You could, for example, handle the WM_SIZE message and invalidate the window there, but I'm always glad when Windows will save me from writing a few extra lines of code. Either way, if you neglect to invalidate the window, then Windows won't send your window any WM_PAINT messages when the window's size is reduced. This might be fine if you're happy for the window's contents to be clipped, but it's common these days to paint various window assets relative to the window's size. Whatever you prefer, this is an explicit decision you need to make for your application window.

While I'm on the topic of the window background, it's often desirable to invalidate a window explicitly. This allows you to keep your window's rendering rooted in the WM_PAINT message rather than have to handle painting at different places and in different code paths through your application. You might want to paint something in response to a mouse click. You could, of course, do the rendering right there in the message handler. Alternatively, you could simply invalidate the window and let the WM_PAINT handler render the current state of the application. This is the role of the InvalidateRect function. ATL provides the Invalidate method that just wraps up this function. What often confuses developers about this function is how to deal with the "erase" parameter. Conventional wisdom seems to be that saying "yes" to erase will cause the window to be repainted immediately and saying "no" will defer this somehow. This isn't true and the documentation says as much. Invalidating a window will cause it to be repainted promptly. The erase option is in lieu of the DefWindowProc function, which would normally clear the window background. If erase is true, then the subsequent call to BeginPaint will clear the window background. Here, then, is another reason for avoiding the window class background brush entirely rather than relying on a WM_ERASEBKGND message handler. Without a background brush, BeginPaint again has nothing to paint with, so the erase option has no effect. If you let ATL set a background brush for your window class, then you need to be careful when invalidating your window because this will

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

again introduce flicker. I added this protected member to the DesktopWindow class template for this purpose:

```
void Invalidate()
{
  VERIFY(InvalidateRect(nullptr, false));
}
```

It's also a good idea to handle the WM_DISPLAYCHANGE message to invalidate the window. This ensures that the window is properly repainted should something about the display change affect the window's appearance.

Running the App

I like to keep my application's WinMain function relatively simple. To achieve this goal, I added a public Run method to the DesktopWindow class template to hide the entire window and Direct2D factory creation, as well as the message loop. The DesktopWindow's Run method is shown in **Figure 3**. This lets me write my application's WinMain function quite simply:

```
int __stdcall wWinMain(HINSTANCE, HINSTANCE, PWSTR, int)
{
  SampleWindow window;
  return window.Run();
}
```

Before creating the window, I prepare the Direct2D factory options by enabling the debug layer for debug builds. I highly recommend that you do the same, as it allows Direct2D to trace out all kinds of useful diagnostics as you develop your application. The D2D1CreateFactory function returns the factory interface pointer that I hand over to the Windows Runtime Library's excellent ComPtr smart pointer, a protected member of the DesktopWindow class. I then call the CreateDeviceIndependentResources method to create any device-independent resources—things such as geometries and stroke styles that can be reused throughout the life of the application. Although I allow derived classes to override this method, I provide an empty stub in the DesktopWindow class template if this isn't needed. Finally, the Run method concludes by blocking with a simple message loop. Check out last month's column for an explanation of the message loop.

The Render Target

The Direct2D render target should be created on demand inside the Render method called as part of the WM_PAINT message handler. Unlike some of the other Direct2D render targets, it's entirely possible that the device—a GPU in most cases—that provides the hardware-accelerated rendering for a desktop window can disappear or change in some way as to make any resources allocated by the render target invalid. Because of the nature of immediate mode rendering in Direct2D, the application is responsible for tracking what resources are device-specific and might need to be re-created from time to time. Fortunately, this is easy enough to manage. **Figure 4** provides the complete DesktopWindow Render method.

The Render method begins by checking whether the ComPtr managing the render target's COM interface is valid. In this way, it only re-creates the render target when necessary. This will happen at least once the first time the window is rendered. If something happens to the underlying device, or for whatever reason the render target needs to be re-created, then the EndDraw method toward the end of the Render method in **Figure 4** will return the D2DERR_RECREATE_TARGET constant. The ComPtr Reset method is then used to simply release the render target. The next time the window is asked to paint itself, the Render method will go through the motions of creating a new Direct2D render target.

It starts by getting the client area of the window in physical pixels. Direct2D for the most part uses logical pixels exclusively to allow it to support high-DPI displays naturally. This is the point at which it initiates the relationship between the physical display and its logical coordinate system. It then calls the Direct2D factory to create the render target object. It's at this point that it calls out to the derived application window class to create any device-specific resources—things such as brushes and bitmaps that are dependent on the device underlying the render target. Again, an empty stub is provided by the DesktopWindow class if this isn't needed.

> I like to keep my application's WinMain function relatively simple.

Before drawing, the Render method checks that the window is actually visible and not completely obstructed. This avoids any unnecessary rendering. Typically, this only happens when the underlying DirectX swap chain is invisible, such as when the user locks or switches desktops. The BeginDraw and EndDraw methods then straddle the call to the application window's Draw method. Direct2D takes the opportunity to batch geometries in a vertex buffer, coalesce drawing commands, and so on to provide the greatest throughput and performance on the GPU.

The final critical step to integrate Direct2D properly with a desktop window is to resize the render target when the window is resized. I've already talked about how the window is automatically invalidated to ensure that it's promptly repainted, but the render target itself has no idea that the window's dimensions have changed. Fortunately, this is easy enough to do, as **Figure 5** illustrates.

Assuming the ComPtr currently holds a valid render target COM interface pointer, the render target's Resize method is called with the new size, as provided by the window message's LPARAM. If for some reason the render target can't resize all of its internal resources, then the ComPtr is simply reset, forcing the render target to be re-created the next time rendering is requested.

And that's all I have room for in this month's column. You now have everything you need to both create and manage a desktop window as well as to use the GPU to render into your application's window. Join me next month as I continue to explore Direct2D. ■

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*

JULIE LERMAN

Playing with the EF6 Alpha

There's nothing like a shiny new toy to play with, and although it's possible to download nightly builds of Entity Framework 6 (EF6) as it evolves, I waited for the first packaged alpha release (which came on Oct. 30, 2012) to dig in and start playing.

If you're asking yourself, "Huh? Nightly builds?" you might have missed the news that after the EF5 release, Entity Framework became an open source project and subsequent versions are being openly (and communally) developed at entityframework.codeplex.com. I recently wrote a blog post, "Making your way around the Open Source Entity Framework CodePlex Site" (bit.ly/W9eqZS), which I recommend checking out before heading over to those CodePlex pages.

The new version will go a long way toward making EF more flexible and extensible. In my opinion, the three most important features coming to EF in version 6 are:

1. Stored procedure and function support for Code First
2. Support for the .NET 4.5 Async/Await pattern
3. The core Entity Framework APIs that currently live inside the Microsoft .NET Framework

Not only does this last point enable enum and spatial type support for apps that target the .NET Framework 4, but because EF is open source it also means that all of EF now benefits from being open source.

Though it might not have the broad appeal of these three features, there's a lot of other significant functionality on its way as well. For example:

- The custom Code First conventions that got pulled prior to the release of EF4.1 are now in EF6, with a variety of ways to implement.
- Code First migrations support multiple database schemas.
- You can define Entity Framework configurations in code rather than setting them in a web.config or app.config file (which can be complicated).
- The code-based configuration is possible because of new support for extensibility with dependency resolvers.
- You can customize how Code First creates the _MigrationsHistory table so that it's more amenable to a variety of database providers.
- The EF Power Tools are being enhanced and added into the Visual Studio EF Designer. One enhancement will provide a nicer path for choosing a model workflow, including Code First.

This article discusses a prerelease version of Entity Framework 6. All information is subject to change.

Getting EF6 Alpha

Hard-core devs might be interested in downloading the nightly builds. If you prefer to use the released packages, you can have a smooth install experience by grabbing the NuGet Package. Use the NuGet Package Manager and select "Include Prerelease" to get the EF6 package. If you install from the Package Manager Console, be sure to add "-prerelease" to the end of your install-package command.

> As this is such an early alpha, I anticipate some of the details for these features will change with new releases.

Note that the Dec. 10, 2012, release that I'm exploring (with file version 6.0.11025.0 and product version 6.0.0-alpha2-11210) doesn't include the stored procedure or function support, or the tooling consolidation. Also, as this is such an early alpha, I anticipate some of the details for these features will change with new releases. While the overall concepts will remain, some of the syntax or other details are likely to evolve based on feedback from the community. Thanks to the exercise of working on this column, I was able to provide some feedback myself.

.NET 4.5 Async in EF6

Leveraging asynchronous processing in the .NET Framework to avoid blocking when it's waiting for data to be returned has daunted many a developer, especially when using disconnected apps or remote databases. The Background Worker process arrived in the .NET Framework 2.0, but it was still complex. ADO.NET 2.0 helped a bit with methods like BeginExecuteQuery and EndExecuteQuery, but Entity Framework has never had anything like these. One of the major additions to the .NET Framework 4.5 is the new Asynchronous pattern, which has the ability to wait for results from methods that have been defined as Asynchronous, dramatically simplifying Async processing.

In EF6, a slew of methods have been added that support the .NET 4.5 Asynchronous pattern. Following the guidelines for the new pattern, the new methods all have Async appended to their names, such as SaveChangesAsync, FindAsync and ExecuteSqlCommandAsync. For LINQ to Entities, a new namespace

1&1 WEB HOSTING

1&1 Windows hosting packages now feature 1 GB (up from 200 MB) **MSSQL 2012** databases. In addition, we've updated **ASP.NET to version 4.5** and added Dedicated Application Pools to achieve top flexibility and performance of your Windows-based web applications. As always, 1&1 data centers offer top security, featuring Cisco firewall protection and maximum uptime due to our geo-redundancy.

- **Dual Hosting for Maximum Reliability**
  Your website hosted across multiple servers in two different data centers, and in two geographic locations.
- **Unlimited Bandwidth** (Traffic)
- **IpV6 Ready**

1&1 Starter Windows	1&1 Starter Linux
50 GB Webspace	
Unlimited Bandwidth (Traffic)	
250 E-Mail Accounts (2 GB each)	
24/7 Phone and E-mail Support	
NEW! ASP.NET/ .NET Framework **4, 4.5**	**NEW! PHP 5.4** and Host multiple websites
NEW! 10 MSSQL **2012** Databases (**1 GB** each)	**10** MySQL Databases (**1 GB** each)
NEW! ASP.NET MVC	**NEW!** Webspace Recovery and daily server backups
NEW! Dedicated App Pools	**2** Click & Build Applications, like Wordpress, Joomla!, TYPO3
~~$4.99~~ **$0.99** per month first year **A $48 DOLLAR SAVINGS**	~~$4.99~~ **$0.99** per month first year **A $48 DOLLAR SAVINGS**

MEMBER OF united internet

1and1.com

1&1®

* Offers valid for a limited time only. Visit www.1and1.com for billing information and full promotional offer details. Program and pricing specifications and availability subject to change without notice. 1&1 and the 1&1 logo are trademarks of 1&1 Internet, all other trademarks are the property of their respective owners.

called System.Data.Entity.IQueryableExtensions contains Async versions of the many LINQ methods, including ToListAsync, FirstOrDefaultAsync, MaxAsync and SumAsync. And to explicitly load data from entities managed by a DbContext, LoadAsync is now available.

What follows is a little experiment I've conducted with this feature, first without using the Asynchronous methods and then with them. I do highly recommend reading up on the new Async pattern. "Asynchronous Programming with Async and Await (C# and Visual Basic)," available at bit.ly/U8FzhP, is a good starting point.

In EF6, a slew of methods have been added that support the .NET 4.5 Asynchronous pattern.

My example contains a Casino class that includes a rating for the casino. I've created a repository method that will find a given casino, increment its rating using my UpdateRating method (which is inconsequential for this explanation, and therefore not listed) and save the change back to the database:

```
public void IncrementCasinoRating(int id)
{
  using (var context = new CasinoSlotsModel())
  {
    var casino =  context.Casinos.Find(id);
    UpdateRating(casino);
    context.SaveChanges();
  }
}
```

There are two points in this method where a thread can get blocked. The first is when calling Find, which causes the context to search its in-memory cache for the requested casino and query the database if it's not found. The second is when asking the context to save the modified data back to the database.

I've structured my UI code solely to demonstrate the relevant behavior. The UI includes a method that calls the repository's IncrementCasinoRating and, when it's finished, writes out a notification in the console:

```
private static void UI_RequestIncrement (SimpleRepository repo)
{
  repo.IncrementCasinoRating(1);
  Console.WriteLine("Synchronous Finish ");
}
```

In another method, I trigger the test by calling UI_ IncrementCasinoRating and follow that with another notification:

```
UI_RequestIncrement (repo);
Console.WriteLine(" After sync call");
```

When I run this, I'll see the following in the console output:

```
Synchronous Finish
After sync call
```

That's because everything stopped while waiting for each of the steps in IncrementCasinoRating to complete—finding the casino, updating the rating and saving to the database.

Now I'll change the repository method so that it uses the new FindAsync and SaveChangesAsync methods. Following the Asynchronous pattern, I also need to make the method asynchronous by:

- adding the async keyword to its signature
- appending Async to the method name
- returning a Task; if the method returns results, then you would return Task<resulttype>

Within the method, I call the new Async methods—FindAsync and SaveChangesAsync—as prescribed, using the await keyword:

```
public async Task IncrementCasinoRatingAsync(int id)
{
  using (var context = new CasinoSlotsModel())
  {
    var casino=await context.Casinos.FindAsync(id);
    // Rest is delayed until await has received results
    UpdateRating(casino);
    await context.SaveChangesAsync();
    // Method completion is delayed until await has received results
  }
}
```

Because the method is marked async, as soon as the first await is hit, the method returns control to the calling process. But I'll need to modify that calling method. There's a waterfall path when building behavior for the Asynchronous pattern, so not only does the method make a special call to my new Asynchronous method, it also needs to be Asynchronous itself because it's being called by yet another process:

```
private static async void UI_RequestIncrementAsync(
  SimpleRepository repo)
{
  await repo.IncrementCasinoRatingAsync(1);
  // Rest is delayed until await has received results
  Console.WriteLine(" Asynchronous Finish ");
}
```

Notice that I'm using await to call the repository method. That lets the caller know this is an Asynchronous method. As soon as that await is hit, control will be returned to the caller. I've also modified the startup code:

```
UI_RequestIncrementAsync(repo);
Console.WriteLine(" After asynchronous call");
```

Code First has a set of built-in conventions that drive its default behavior when it builds a model along with database mappings from your classes.

Now when I run this code, the UI_RequestIncrementAsync method returns control to the caller as it's also calling the repository method. That means I'll immediately get to the next line of startup code, which prints out "After asynchronous call." When the repository method finishes saving to the database, it returns a Task to the method that called it, UI_RequestIncrementAsync, which then executes the rest of its code, printing out a message to the console:

```
After asynchronous call
Asynchronous Finish
```

So my UI was able to finish without waiting for EF to complete its work. If the repository method had returned results, they would have bubbled up in the Task when they were ready.

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

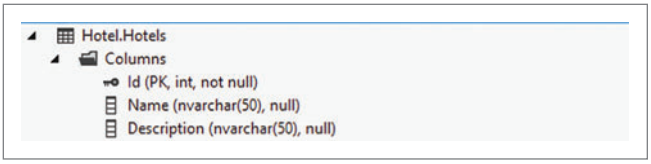


Figure 1 **A Custom Convention Made the Max Length of These nvarchars 50**

This little exercise helped me see the new Asynchronous methods in action. Whether you're writing client-side or disconnected applications that rely on asynchronous processing, it's a great benefit that EF6 now supports the new Asynchronous pattern with such simplicity.

Custom Conventions

Code First has a set of built-in conventions that drive its default behavior when it builds a model along with database mappings from your classes. You can override those conventions with explicit configurations using DataAnnotations or the Fluent API. In the early betas of Code First you could also define your own conventions—for example, a convention that sets all strings to map to database fields with a max length of 50. Unfortunately, the team wasn't able to get this feature to a satisfactory state without holding up Code First, so it didn't make it into the final release. It has now made its return in EF6.

There are several ways to define your own conventions.

> EF6 gives Code First migrations the ability to handle multi-tenant databases.

Using a lightweight convention is the simplest method. It allows you to specify conventions fluently in the OnModelCreating overload of the DbContext. Lightweight conventions are applied to your classes and are limited to configuring properties that have a direct correlation in the database, such as length to MaxLength. Here's an example of a class that has no special configurations and therefore, by default, its two string fields would map to nvarchar(max) data types in a SQL Server database:

```
public class Hotel
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
  }
```

I've added a lightweight convention in the model specifying that the API should check the properties of any entity it's processing and set the MaxLength of strings to 50:

```
modelBuilder.Properties<string>()
  .Configure(p => p.HasColumnType("nvarchar"));
```

You can see in **Figure 1** that Code First ensured the Name and Description fields do have a max length of 50.

You can also define a convention by implementing an existing interface, such as the convention interface for handling DateTime properties—the DateTimePropertyConfiguration class in the System.Data.Entity.ModelConfiguration.Configuration.Properties.Primitive namespace. **Figure 2** shows an example in which I forced DateTime properties to map to the SQL Server date type instead of the default datetime. Note that this sample follows Microsoft guidance—I won't apply my configuration if the attribute (ColumnType in this case) has already been configured.

Conventions have a specific pecking order, which is why you have to be sure that the column type hasn't already been configured before applying the new ColumnType.

The model builder needs to know how to find this new convention. Here's how to do that, again in the OnModelCreating overload method:

```
modelBuilder.Conventions.Add(new DateTimeColumnTypeConvention());
```

There are two other ways to customize conventions. One method allows you to create custom attributes you can use in your classes as easily as DataAnnotations. The other is more granular: rather than building a convention that depends on what the ModelBuilder learns from your classes, this method allows you to affect the metadata directly. You'll find examples of all four styles of custom conventions in the MSDN Data Developer Center documentation, "Custom Code First Conventions" (msdn.microsoft.com/data/jj819164). As EF6 evolves, this document will either gain a link to a more-current version or be modified to align with the most-recent version.

Multiple Schema Support for Migrations

EF6 gives Code First migrations the ability to handle multiple schemas in databases. For more about this feature, take a look at the detailed blog post I wrote shortly after the alpha was released: "Digging in to Multi-Tenant Migrations with EF6 Alpha" (bit.ly/Rrz1MD). Do keep in mind, however, that the name of the feature has changed from "Multi-Tenant Migrations" to "Multiple Contexts per Database."

Code-Based Configurations

You can already specify database-relevant configurations for Entity Framework in application config files (app.config and web.config), freeing you from having to supply the configurations at application startup or in the constructor of the context. Now, in EF6, it's possible to create a class that inherits from a new DbConfiguration class where you can specify details such as the default database

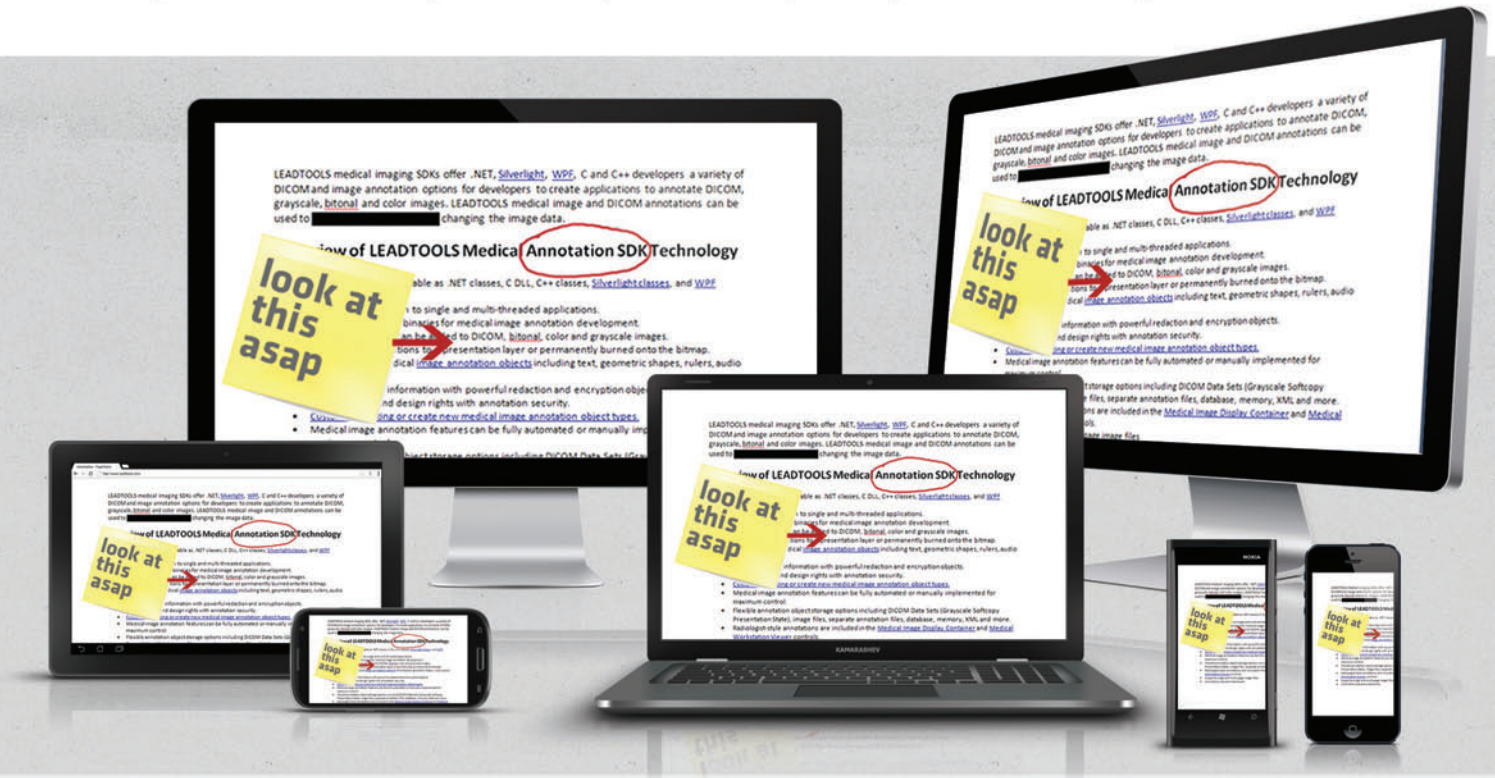
Figure 2 **Mapping DateTime Properties to the SQL Server Date Type**

```
public  class DateTimeColumnTypeConvention :
  IConfigurationConvention<PropertyInfo, DateTimePropertyConfiguration>
  {
    public void Apply(
      PropertyInfo propertyInfo,
      Func<DateTimePropertyConfiguration> configuration)
    {
      // If ColumnType hasn't been configured ...
      if (configuration().ColumnType == null)
      {
        configuration().ColumnType = "date";
      }
    }
  }
```

LEADTOOLS V18
THE WORLD LEADER IN IMAGING SDKs ANYWHERE

DOCUMENT SDKs

.NET, WIN 32/64, WinRT, HTML5, iOS, ANDROID, OS X & LINUX



The world's leading Imaging SDK
NOW **RUNS ANYWHERE**

OCR	**BARCODE**	**PDF & PDF/A**
FORMS RECOGNITION	**VIRTUAL PRINTER**	**ANNOTATIONS & MARKUP**
150+ FORMATS	**SCANNING**	**DOCUMENT CLEANUP & PREPROCESSING**
ZERO FOOTPRINT VIEWERS	**RICH CLIENT VIEWERS**	**RUNS ON DESKTOP, MOBILE & TABLET**

COMPREHENSIVE IMAGING SDK FOR **C++, C#, VB, JavaScript, OBJECTIVE-C & JAVA**

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

provider for Code First, the database-initialization strategy (for example, DropCreateDatabaseIfModelChanges) and others. You can create this DbConfiguration class in the same project as your context or in a separate project, allowing multiple contexts to benefit from a single configuration class. The code-based configuration overview at msdn.microsoft.com/data/jj680699 provides examples of the various options.

Core EF Is Now in EF6 and It's Open Source, Too

While the Code First and DbContext APIs have always been disconnected from the .NET release cycle, the core of EF has been embedded in the .NET Framework. This is the core functionality—the ObjectContext API, querying, change-tracking, the EntityClient provider and so much more. This is why support for enums and spatial data had to wait for the .NET Framework 4.5 to be released—those changes had to be made deep within the core APIs.

> One of the great benefits of having the core APIs inside of EF6 is that it removes some of the dependency on .NET versions for EF-specific features.

With EF6, all of those core APIs have been pulled into the open source project and will be deployed via the NuGet package. It's interesting to see the EF5 and EF6 namespaces side-by-side, as shown in **Figure 3**. As you can see, in EF6 there are many more namespaces.

Enum and Spatial Support for .NET 4 Apps

As I mentioned earlier, one of the great benefits of having the core APIs inside of EF6 is that it removes some of the dependency on .NET versions for EF-specific features—most notably the enum and spatial data support added to EF in the .NET Framework 4.5. Existing apps that targeted the .NET Framework 4 with EF were not able to take advantage of this with EF5. Now that limitation is gone because EF6 includes the enum and spatial support, and the features are therefore no longer dependent on the .NET Framework. The existing documentation on these features can help you use them in apps that target the .NET Framework 4.

Help Drive the Evolution of EF

With Entity Framework now an open source project, developers can help themselves and others by getting involved with its design and development. You can share your thoughts, comments and feedback, whether from reading the specs, taking part in discussions and issues, playing with the latest NuGet package, or grabbing nightly builds and banging on them. You can also contribute code, either for one of the issues listed on the site that nobody's working on yet or something of your own that you can't live without in EF. ■

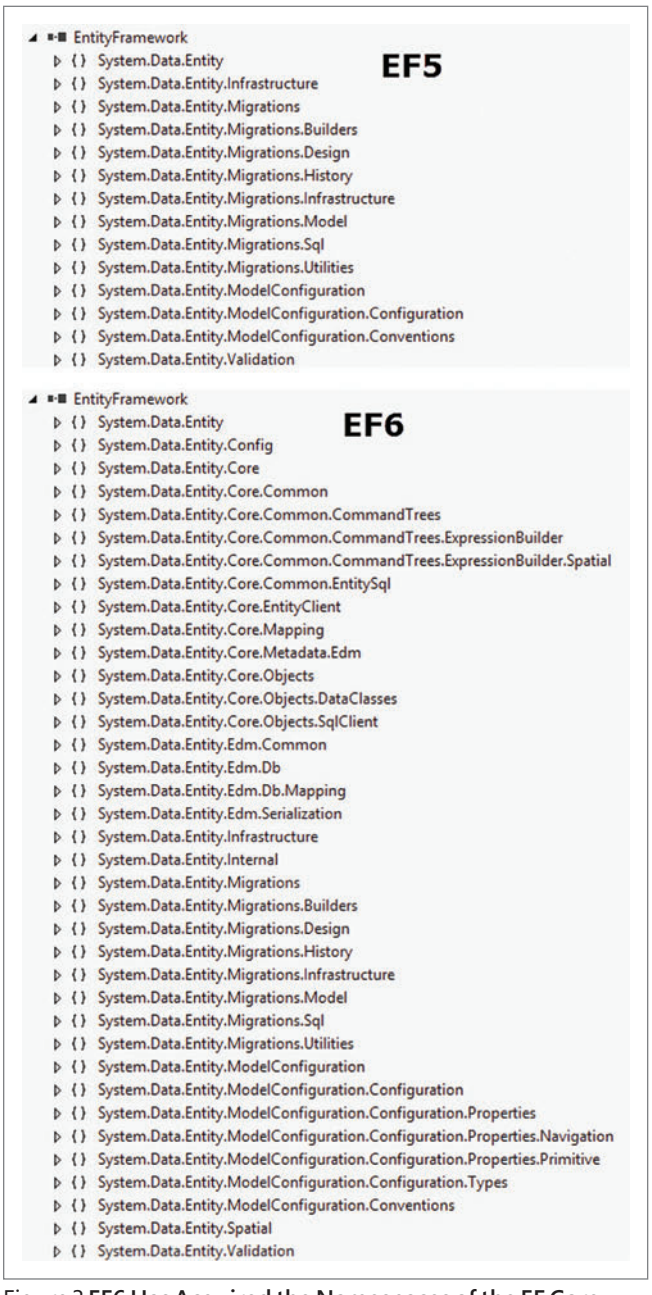


Figure 3 **EF6 Has Acquired the Namespaces of the EF Core APIs from the .NET Framework**

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

Thanks to the following technical expert for reviewing this article: *Glenn Condron*

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 4 days of unbiased .NET training led by industry experts and Microsoft insiders.

SUPPORTED BY
Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY
1105 MEDIA

CHICAGO | MAY 13-16, 2013

HILTON CHICAGO

REGISTER TODAY!

USE PROMO CODE CHMAR2

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

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

linkedin.com – Join the "VSLive" group!

vslive.com/chicago

BRUNO TERKALY AND
RICARDO VILLALOBOS

Real-World Scenarios for Node.js in Windows Azure

The popular quote, "If all you have is a hammer, everything looks like a nail," certainly applies to software architecture. The best developers, however, understand a wide variety of frameworks, programming languages and platforms so they can engineer solutions that not only fulfill immediate business requirements, but also result in solutions that are scalable, maintainable, extensible and reusable. Node.js burst onto the scene three years ago, offering yet another tool for creating server-side software systems that support scalable Internet applications. Like all development tools, Node.js is not a magic hammer, and its capabilities should be fully understood before deciding if it's the right fit for the solution at hand.

In case you're new to Node.js, it's a platform for building scalable network applications, based on the Google V8 JavaScript engine. It provides a single-threaded evented-io model, which allows orchestration of tasks running in parallel using an asynchronous/event-callback/non-blocking approach as shown in **Figure 1**. Node.js can be seen as a lightweight server that supports multiple connections without requiring a large memory footprint.

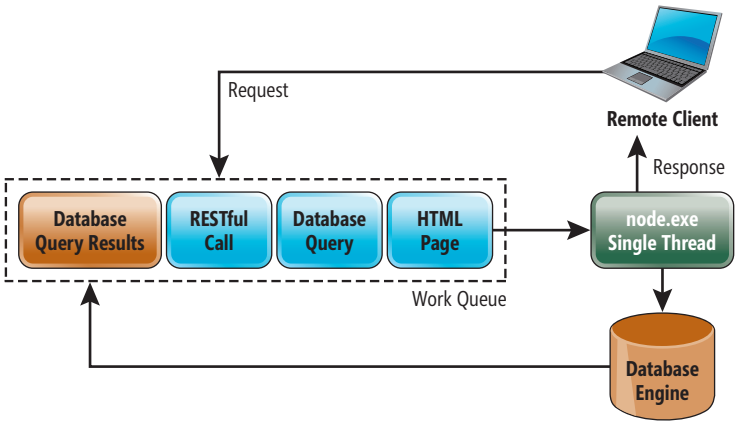


Figure 1 **Node.js—Single-Threaded Model Based on an Asynchronous/Event-Callback/Non-Blocking Approach**

> The best way to evaluate a new technology is in the real world.

From a deployment perspective, the full Node.js engine is contained in a small executable—less than 5MB—that can be installed in Windows, Linux or Mac OS X. It implements a highly modularized architecture, including a few built-in components, such as those for listening to HTTP and TCP ports, making requests, or accessing the file system. Additional modules, provided by a strong open community, can be downloaded as needed, using the Node package manager utility (npm). Thanks to this approach, it's possible to have an HTTP server up and running with five lines of code:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');}).listen(8080);
console.log('Server running on port 8080');
```

Having said this, there are a few significant challenges Node.js developers face. First, it forces a programming model based on asynchronous calls, running on a single thread. This is a paradigm shift from conventional programming, where tasks can be assigned to multiple threads. If care is not taken, programmers can get in trouble, either blocking the server or creating unexpected code behaviors. The second challenge is related to callbacks: Code can become unwieldy and difficult to maintain due to deep nesting. Last, debugging is not simple, particularly for complex scenarios. Even though there are approaches for working through each of these challenges, it takes conscious effort and learning. Keep in mind that Node.js is young; fortunately, there's a robust community of programmers ready to help, as well as online resources such as howtonode.org.

At the end of the day, the best way to evaluate a new technology is in the real world, where problems are encountered and solved. In this article we present two specific cloud-based scenarios using Node.js, with Windows Azure as the deployment platform. If you're already a client-side JavaScript developer, you'll hit the ground running from a language point of view.

Code download available at archive.msdn.microsoft.com/mag201303AzureInsider.

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

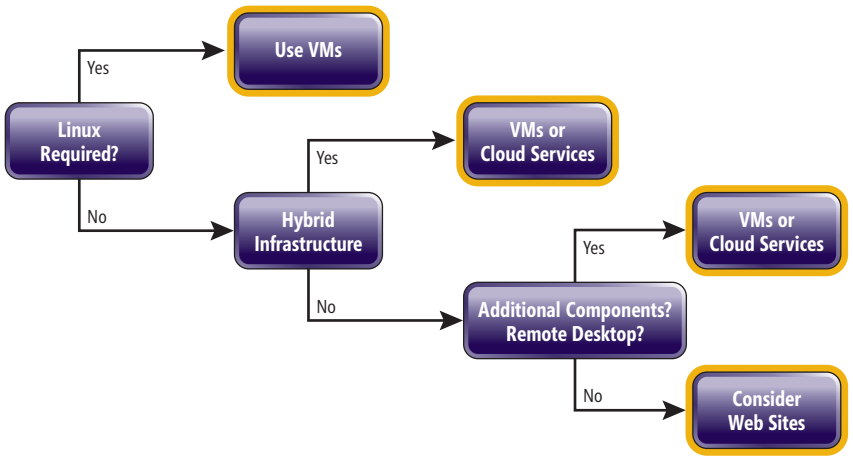


Figure 2 **Decision Tree for Deploying Node.js Applications to Windows Azure**

Prerequisites

Before you start testing the solutions presented in this article, make sure you download the Node.js Windows installer, which can be found at nodejs.org/download. The installer will place two main files in your Windows Program Files folder: node.exe, which is the Node.js runtime, and the npm, which allows you to download third-party modules. For deploying Node.js solutions to Windows Azure, download the corresponding command-line tools, which you'll find at windowsazure.com/en-us/develop/downloads. Prerequisites that are specific to the use cases are included in each section.

Deploying Node.js Applications to Windows Azure

Windows Azure offers three cloud deployment models for applications to be deployed in any of the eight Microsoft global datacenters: Virtual Machines (VMs), Cloud Services and Web Sites. The most appropriate deployment model depends on the level of scale, control and flexibility you require. The price you pay for more scale, control and flexibility is that more work is involved to deploy and maintain your Node.js application. The Web Sites model frees the developer from worrying about firewall rules, virtual networks and OSes. Naturally, you give up fine-grained control of your deployment when you select this option.

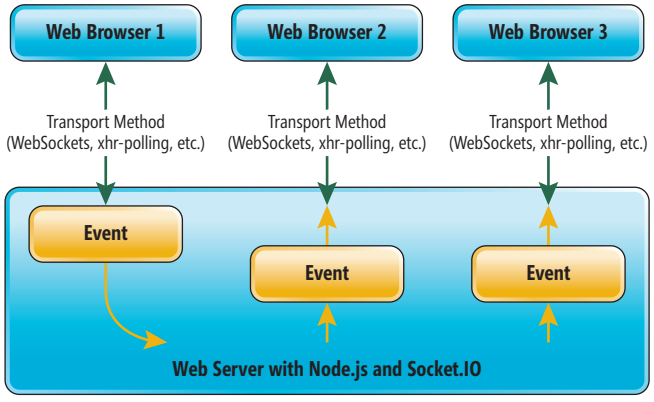


Figure 3 **Real-Time Communication Between Web Browsers and HTTP Servers Using Node.js**

Even though it's impossible to cover all types of scenarios in a single diagram, **Figure 2** shows a decision tree for determining where to deploy your Node.js solution, based on infrastructure and software components required by the application. We'll use it to determine the Windows Azure deployment model for the real-world examples in this article.

Real-World Scenario 1: Real-Time Web Communication Between HTTP Clients and Servers The first scenario illustrates how Node.js makes sense for applications that require real-time communication between Web browsers and HTTP servers, such as chat solutions, social media, news tickers and video games. Traditionally, developers have achieved this type of communication by using different long-term connection mechanisms, including long-polling and streaming. More recently, the HTML5 specification has included a communication protocol named WebSockets that provides full-duplex communications channels over a single TCP connection, but this is only supported by the latest versions of the most common Web browsers. Node.js applications can support real-time communication scenarios through a third-party module called socket.io, which supports multiple types of transports, including xhr-polling and WebSockets. Socket.io is based on an event-driven approach between the server and the Web browser clients, as depicted in **Figure 3**. It is well-documented at bit.ly/NIDOv7.

Figure 4 **Server-Side Code for Establishing Real-Time Communication**

```
// Include needed packages (socket.io and express)
var express = require('express');
var app = express()
  , http = require('http')
  , server = http.createServer(app)
  , io = require('socket.io').listen(server);

// REPLACE BELOW var port = var port = process.env.PORT || 8080;
// Allow connections on port 8080, or the environment port number
var port = process.env.PORT || 8080;

// At the time of this writing, WebSockets is not supported
// in Windows Azure Web Sites, which will force socket.io
// to fall back to a different communication protocol
// Prevent potential problems by specifying one, in this case, xhr-polling
io.set('transports', ['xhr-polling']);

// Listen for incoming requests
server.listen(port);

// Redirect request to index.html
app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

// When connected and sendmessage is called by client,
// broadcast data sent by one client to all connected clients
io.sockets.on('connection', function (socket) {
  // When the client emits 'sendmessage,' the following method is triggered
  socket.on('sendmessage', function (data) {
    // Message is broadcast to all clients
    socket.broadcast.emit('displaymessage', data);
  });
});
```

Figure 5 **Client-Side Code for Establishing Real-Time Communication**

```
<html>
<head>
<script src="/socket.io/socket.io.js"></script>
<script>
  // Initialize the socket connection
  var socket = io.connect();

  // Ask client (browser input box) to enter text
  function sendMessage(){
    socket.emit('sendmessage', prompt("Message to broadcast?"));
  }

  // Displaymessage event received at all clients
  // display in alert dialog box
  socket.on('displaymessage', function(data){
    alert(data);
  });
</script>
</head>
<body>
<!—Client sends user input to node.js server through the sendMessage
JavaScript function-->
<input type="button" value="Broadcast new message" onClick="sendMessage();">
</body>
</html>
```

The basic flow is as follows:

1. The Web client connects to the server and agrees on a protocol for communication (such as WebSockets, XMLHttpRequest (XHR), long-polling or flash sockets).
2. The Web client sends an event to the Node.js server via JavaScript, using the socket.emit method.
3. The server captures the event by matching the name of the function sent by the client to the one defined in any of its socket.on definitions.
4. The server can respond to the client by using the socket.emit method, or broadcast messages to all the connected clients using socket.broadcast.emit.

This is illustrated in **Figure 4** and **Figure 5**, which show the server-side code and client-side code, respectively. The code simply broadcasts a message to all connected clients.

Testing Your Application Locally

Follow these steps to test your application locally:

1. Create a local folder called {*drive letter*}://nodejs/sockets.
2. Using your preferred text editor, create a file called server.js.
3. Copy and paste the code listed in **Figure 4** for the server side.
4. In the same directory, create a file called index.html.
5. Copy and paste the code listed in **Figure 5** for the client side.
6. Open a command prompt, and change the directory to {*drive letter*}://nodejs/sockets.
7. Make sure you're connected to the Internet, and type *npm install socket.io*. This will install the required socket.io module.
8. Type *npm install express*. Express is a module that simplifies access to the HTTP server functions, and can be easily integrated with socket.io.
9. Type *node server.js*.
10. Open a Web browser compatible with WebSockets and enter the URL http://localhost:8080.
11. Open a second Web browser tab or window, pointing to the same URL.
12. A message sent from the first client will be broadcast to all the other clients connected to the server.

Deploying Your Solution to Windows Azure

Based on the decision tree in **Figure 2**, Windows Azure Web Sites is a good option for our application (Linux isn't required; no additional components in the OS are needed; and a hybrid infrastructure isn't necessary to run the solution). Keep in mind that at the time of this writing, WebSockets is not supported in Windows Azure Web Sites, so we've added a line in our code that specifies the communication protocol to be used by socket.io—we want to use the xhr-polling transport for the communication between the server and the Web browser clients. The easiest way to deploy an application to Windows Azure Web Sites is by using Git, which can be downloaded at git-scm.com/download. Once you have it installed, go to the Windows Azure portal at manage.windowsazure.com and create a new empty Web Site. Enable Git publishing by clicking on the corresponding option, as shown in **Figure 6**.

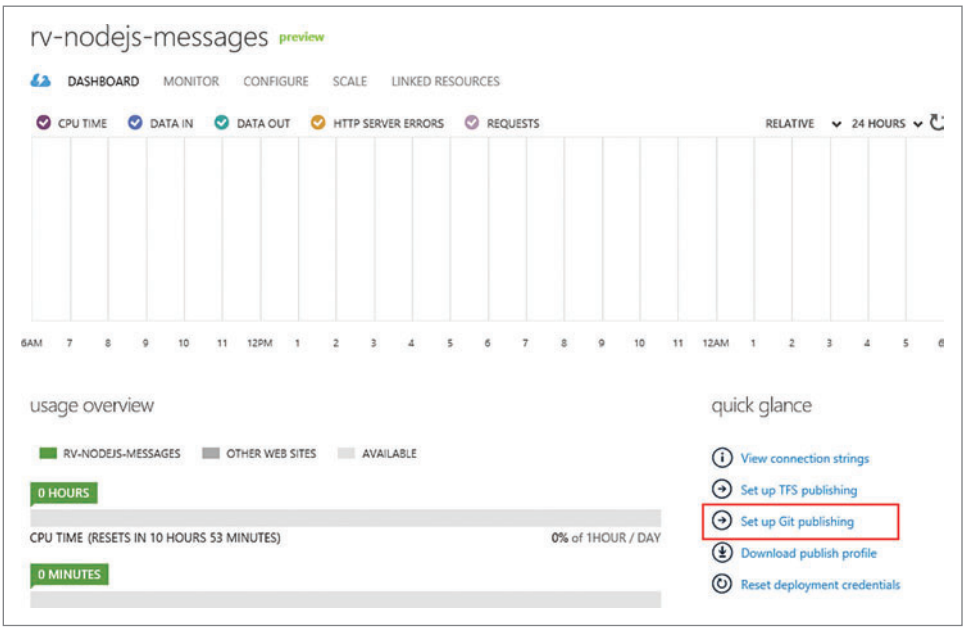


Figure 6 **Enable Git Publishing for Your New Web Site**

After a few seconds, a new screen will appear, showing the URL to the Git repository for your Web Site. If you haven't set up any credentials for your account yet, you'll need to provide them before continuing.

TURNING THE WORLD OF .NET APPLICATION DEVELOPMENT UPSIDE DOWN

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

FAT

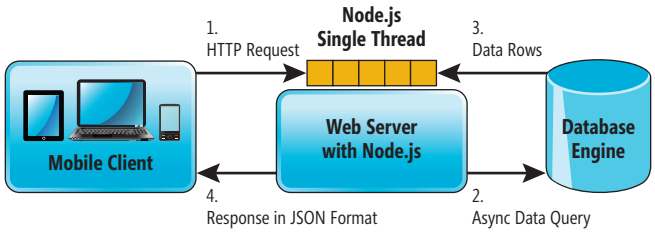


Figure 7 **Node.js Provides a Web Service Layer to Data Apps**

The URL is in the form https://WindowsAzureAccount@WebSiteName.scm.azurewebsites.net/WebSiteName.git. Record it, because it will be used in the next deployment steps. Be sure you executed the local test before continuing, because the modules need to be downloaded first. (Note that Web Sites can also be created using the command-line tool for Mac and Linux, found at bit.ly/RGCc3A.) Once your Web Site has been created, follow these steps to deploy your Node.js solution to Windows Azure:

1. Open a command prompt and change your current folder to {*drive letter*}://nodejs/sockets (where you created your application).
2. Type *git init*. This creates a local Git repository for your solution.
3. Type *git add*. This adds the solution to your new local repository.
4. Type *git commit –m "Initial commitment."* This creates a pending Git commitment.
5. Type *git remote azure {URLforGITRepository}*. Use the Git URL found in the Windows Azure portal, which you previously recorded.
6. Type *git push azure master*. Enter your password when requested.
7. Wait until progress gets to 100 percent, and then your solution is ready to be used in the cloud.

Figure 8 **Exposing a Full RESTful API for a SQL Server Database Running on Windows Azure**

```
var sql = require('msnodesql');
var express = require('express');
var conn_str = "Driver={SQL Server Native Client 11.0};
  Server=[serverName].database.windows.net,1433;Database=AdventureWorks2012;
  Trusted_Connection={No};
  Uid=[LoginForDb]@[serverName];Pwd=[Password];Encrypt=yes";
var app = express();
var port = process.env.PORT || 8080;
app.get('/', function(req,res) {sql.query(conn_str, "SELECT FirstName,
  LastName FROM Person.Person", function (err, results) {
    if (err)
      console.log(err);
    else
      res.json(results);
  })
});
app.get('/lastname/:lastname', function(req,res) {sql.query(conn_str,
  "SELECT FirstName, LastName FROM Person.Person WHERE LastName LIKE ?",
  [req.params.lastname], function (err, results) {
    if (err)
      console.log(err);
    else
    {
      for (var i = 0; i < results.length; i++) {
        res.json(results[i]);
      }
    }
  })
});
app.listen(port);
console.log("Server listening on port 8080");
```

Real-World Scenario 2: Creating a Quick and Robust RESTful Web Service Layer for Data Applications As we've discussed in previous articles, mobile solutions running on different devices (iOS, Android, Windows Phone) can be unified by making them talk to Web services that provide access to data operations in the back end. This usually requires a data layer that extracts or inserts information from or to the database, as well as a service layer that maps internal objects to UI objects (usually in JSON format). Even though this can be achieved by using traditional Web servers such as IIS and frameworks such as the ASP.NET Web API, Node.js offers a simpler solution for this scenario, acting as a simple orchestrator that delegates queries to the database engine and sends back responses in native JSON format, due to its JavaScript nature (see **Figure 7**).

Any objects obtained from the database can be easily returned in JSON format by using the res.json method. The example in **Figure 8** defines a couple of RESTful calls for the HTTP server, reads data from a SQL Server database and returns the results in JSON format.

In order to test this solution, deploy the AdventureWorks database to Windows Azure by following the instructions at bit.ly/dOapaC. You'll need to modify the connection string in the example in **Figure 8** accordingly. To deploy the example to Windows Azure, follow the same steps explained in the first scenario for testing locally and deploying to the cloud.

This is the simplest way to expose a full RESTful API for your data layer running on Windows Azure. Even though we used a SQL Server database to illustrate this scenario, many other data engines are supported in Node.js, including Windows Azure Table Storage, MongoDB and Cassandra, among others. Some of them, like MongoDB, are offered in an as-a-service model in the Windows Azure store, which facilitates integration with applications deployed in the Microsoft cloud, including Node.js applications.

Wrapping Up

We've shown you two real-world scenarios where Node.js can be used for simple connectivity tasks, taking advantage of its single-threaded approach and modules created by the community. The important thing to remember is that any synchronous blocking operation disrupts this model, and applications should be written with this in mind. In many cases, Node.js can be installed side-by-side with other engines, acting as an offload server for specific functionality inside the solution. Also, Node.js can be deployed using three different models in Windows Azure, depending on the level of scalability and control required. ■

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

Ricardo Villalobos *is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in the supply chain management industry. Holding different technical certifications, as well as a master's degree in business administration from the University of Dallas, he works as a cloud architect in the Windows Azure CSV incubation group for Microsoft.*

Thanks to the following technical expert for reviewing this article: *Glenn Block*

Using XAML with DirectX and C++ in Windows Store Apps

Doug Erickson

Since Windows Vista, DirectX has been the core graphics API for the Windows platform, enabling graphics processing unit (GPU) acceleration for all OS screen-drawing operations. However, until Windows 8, DirectX developers had to roll their own UI frameworks from the ground up in native C++ and COM, or license a middleware UI package such as Scaleform.

In Windows 8, you can bridge the gap between native DirectX and a proper UI framework with the DirectX-XAML interop feature of the Windows Runtime (WinRT). To take advantage of the API support for XAML in DirectX, you're required to use "native" C++ (although you have access to smart pointers and the C++ component extensions). A little basic knowledge of COM helps as well, although I'll spell out the specific interop you must perform to bring the XAML framework and DirectX operations together.

In this two-part series of articles, I'll look at two approaches to DirectX-XAML interop: one where I draw surfaces into XAML framework elements with the DirectX graphics APIs; and one where I draw XAML controls hierarchy atop a DirectX swap chain surface.

This article discusses the first scenario, where you render to images or primitives displayed within your XAML framework.

But first, here's a quick overview of your API options. Right now, there are three XAML types in the Windows Runtime that support DirectX interop:

- **Windows::UI::Xaml::Media::Imaging::SurfaceImageSource** (SurfaceImageSource hereafter): This type lets you draw relatively static content to a shared XAML surface using DirectX graphics APIs. The view is entirely managed by the WinRT XAML framework, which means that all presentation elements are likewise managed by it. This makes it ideal for drawing complex content that doesn't change every frame, but less ideal for complex 2D or 3D games that update at a high frequency.
- **Windows::UI::Xaml::Media::Imaging::VirtualSurfaceImageSource** (VirtualSurfaceImageSource hereafter): Like SurfaceImageSource, this uses the graphics resources defined for the XAML framework. Unlike SurfaceImage-

This article discusses:

- DirectX interop
- Limitations of using SurfaceImageSource
- SurfaceImageSource and DirectX image composition
- VirtualSurfaceImageSource and interactive control rendering

Technologies discussed:

DirectX, C++, Windows 8, Windows Runtime

Figure 1 **Deriving from SurfaceImageSource**

```
public ref class MyImageSourceType sealed : Windows::UI::Xaml::
  Media::Imaging::SurfaceImageSource
{
  // ...
  MyImageSourceType::MyImageSourceType(
    int pixelWidth,
    int pixelHeight,
    bool isOpaque
  ) : SurfaceImageSource(pixelWidth, pixelHeight, isOpaque)
  {
    // Global variable that contains the width,
    // in pixels, of the SurfaceImageSource.
    m_width = pixelWidth;
    // Global variable that contains the height,
    // in pixels, of the SurfaceImageSource.
    m_height = pixelHeight;

    CreateDeviceIndependentResources();
    CreateDeviceResources();
  }
  // ...
}
```

Source, VirtualSurfaceImageSource supports logically large surfaces in an optimized, region-based way, such that DirectX only draws the regions of the surface that change between updates. Choose this element if you're creating a map control, for example, or a large, image-dense document viewer. Again, like SurfaceImageSource, this isn't a good choice for complex 2D or 3D games, especially ones that rely on real-time visuals and feedback.

- **Windows::UI::Xaml::Controls::SwapChainBackgroundPanel** (SwapChainBackgroundPanel hereafter): This XAML control element and type allows your app to use a custom DirectX view provider (and swap chain) on top of which you can draw XAML elements, and which allows for better performance in scenarios that require very low-latency presentation or high-frequency feedback (for example, modern games). Your app will manage the DirectX device context for the SwapChainBackgroundPanel separately from the XAML framework. Of course, this means that both the SwapChainBackgroundPanel and the XAML frames aren't synchronized with each other for refresh. You can also render to a SwapChainBackgroundPanel from a background thread.

This time, I'll take a look at the SurfaceImageSource and VirtualSurfaceImageSource APIs, and how you can incorporate them into your rich image and media XAML controls (SwapChainBackgroundPanel is special and gets its own article).

Note: SurfaceImageSource and VirtualSurfaceImageSource can be used from C# or Visual Basic .NET, although the DirectX rendering component must be written in C++ and compiled to a separate DLL accessed from the C# project. There are also third-party managed WinRT DirectX frameworks, such as SharpDX (sharpdx.org) and MonoGame (monogame.net), which you can use instead of SurfaceImageSource or VirtualSurfaceImageSource.

So, let's get started. This article assumes that you understand the basics of DirectX, specifically Direct2D, Direct3D and Microsoft DirectX Graphics Infrastructure (DXGI). Of course, you know XAML and C++; this is an intermediate subject for Windows app developers. Thus girded: onward!

SurfaceImageSource and DirectX Image Composition

The Windows::UI::Xaml::Media::Imaging namespace contains SurfaceImageSource type, alongside many of the other XAML imaging types. In fact, the SurfaceImageSource type provides a way to dynamically draw to the shared surfaces of many XAML graphics and imaging primitives, effectively filling them with the contents you render with DirectX graphics calls and applying them as a brush. (Specifically, it's an ImageSource that you use as an ImageBrush.) Think of it like a bitmap that you're generating on the fly with DirectX, and consider that you can use this type in many places where you could apply a bitmap or other image resource.

For the purposes of this section, I'll draw into an <Image> XAML element that contains a blank PNG image as a placeholder. I provide a height and width for the <Image> element, because this information is passed to the SurfaceImageSource constructor in my code (if I don't provide a height and width, the content I render will be stretched to fit the <Image> tag parameters):

```
<Image x:Name="MyDxImage" Width="300" Height="200" Source="blank-image.png" />
```

In this example, my target is the <Image> tag, which will display the surface into which I'm drawing. I could use a XAML primitive as well, such as a <Rectangle> or an <Ellipse>, both of which can be filled by a SurfaceImageSource brush. This is possible because the drawings for these primitives and images are performed with DirectX by the Windows Runtime; all I'm doing is hooking up a different rendering source under the covers, as it were.

In my code, I include the following:

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

#include <dxgi.h>
#include <dxgi1_2.h>
#include <d2d1_1.h>
#include <d3d11_1.h>

#include "windows.ui.xaml.media.dxinterop.h"
```

These are the headers for the Windows Runtime Library (WRL), some key DirectX components and, most important, the native DirectX interop interfaces. The need for the latter to be included will become apparent soon.

I also import the corresponding libraries: dxgi.lib, d2d1.lib and d3d11.lib.

And for convenience, I'll also include the following namespaces:

```
using namespace Platform;
using namespace Microsoft::WRL;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Media::Imaging;
```

Now, in the code, I create a type, MyImageSourceType, that inherits from the base SurfaceImageSource type and calls its constructor, as shown in **Figure 1**.

Note: You don't need to inherit from SurfaceImageSource, although it makes things a bit easier from a code organization perspective. You can simply instantiate a SurfaceImageSource object as a member, and use it instead. Just mentally substitute the name of your member for the object self-reference (*this*) in the code examples.

The CreateDeviceResources and CreateDeviceIndependentResources methods are user implementations that are a convenient way to logically separate the setup specific to the DirectX graphics hardware interface and the more general DirectX app-specific

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

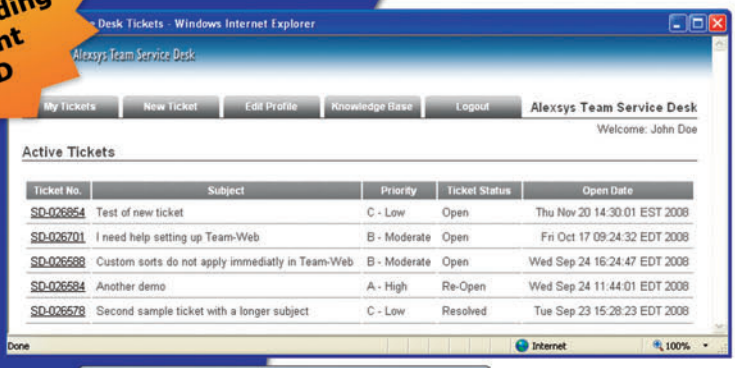
Service Desk Features

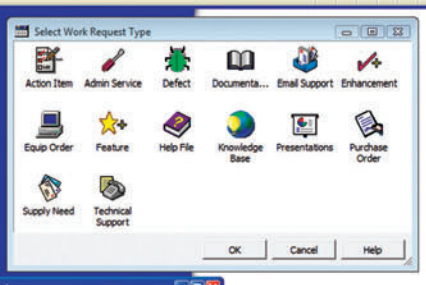
- Fully Secure
- Unlimited Users Self Registered or Active Directory
- Integrated into Your Web Site
- Fast/AJAX Dynamic Content
- Unlimited Service Desks
- Visual Service Desk Builder

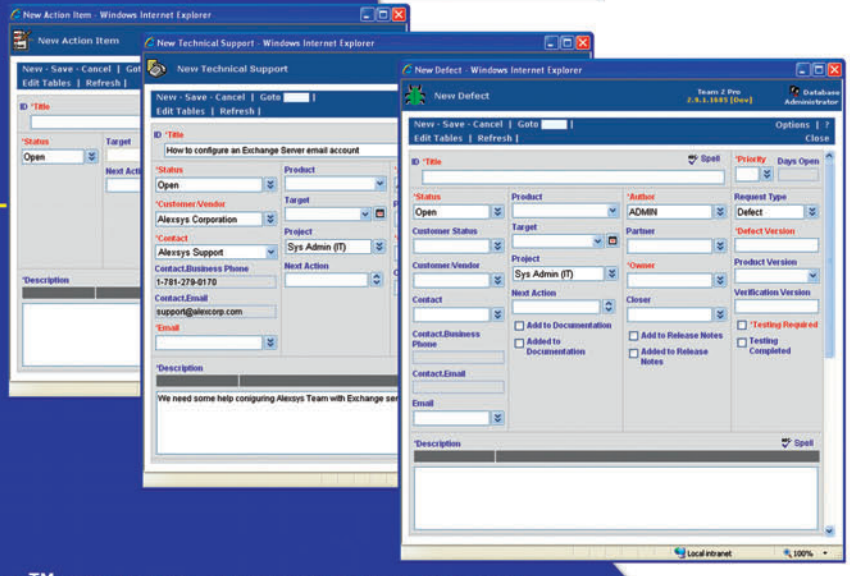
Team 2 Features

- 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.

setup. The actions taken in both methods are essential. However, it's good (and necessary) design to separate them, as there are times when you might want to re-create the device resources without affecting the device-independent resources, and vice versa.

CreateDeviceResources should look similar to the code in **Figure 2**, at least in a basic form.

At this point, I've created a hardware device context and tied it to a ... wait, what's ISurfaceImageSourceNative? That's not a WinRT type! What's going on here?

This is the interop bit. This is where I sneak into the "Jeffries tube" of the WRL and do a little rewiring. It's also where I get into the COM that sits behind much of the WRL.

In order to enable this interop behavior, I need to essentially plug in that DirectX source under the hood. To do so, I need to hook my type into implementation of the methods defined on the WRL-specific COM interface, ISurfaceImageSourceNative. When I've done that, I'll attach the type to the <Image> element (in this example), and when the app pushes an update to the XAML framework, it'll use my DirectX implementations of the draw calls instead of the default ones.

ISurfaceImageSourceNative is defined in the interop header I specified earlier. See what's going on here?

Now, in my app-specific CreateDeviceIndependentResources method, I break out the COM and query for the native methods defined on SurfaceImageSource. Because these methods aren't exposed directly, they must be obtained with a call to IUnknown::QueryInterface on the SurfaceImageSource or SurfaceImageSource-derived type. To do this, I recast my SurfaceImageSource-derived type as IUnknown, the base interface for any COM interface (I could also cast it as IInspectable, the "base" interface for any WinRT type, which inherits from IUnknown). Then, to get a list of the ISurfaceImageSourceNative methods, I query that interface, like so:

```
void MyImageSourceType::CreateDeviceIndependentResources()
{
  // Query for ISurfaceImageSourceNative interface.
  reinterpret_cast<IUnknown*>(this)->QueryInterface(
    IID_PPV_ARGS(&m_sisNative));
}
```

(IID_PPV_ARGS is a helper macro for the WRL that retrieves an interface pointer. Very convenient! If you aren't inheriting from SurfaceImageSource, substitute your SurfaceImageSource object's member name for *this*.)

Finally, this part of the CreateDeviceResources method makes sense:

```
m_sisNative->SetDevice(dxgiDevice.Get());
```

ISurfaceImageSourceNative::SetDevice takes the configured graphics interface and couples it to the surface for any drawing operations. Note, though, that it also means I should call CreateDeviceResources *after* calling CreateDeviceIndependentResources at least once prior—or I won't have a configured device to attach.

I've now exposed the underlying ISurfaceImageSourceNative implementation of the SurfaceImageSource type from which my MyImageSourceType type derives. I've effectively flipped open the hood and moved the wires to the SurfaceImageSource type, albeit to the base implementation of the draw calls and not my own. Now, I'll implement *my* calls.

To do this, I implement the following methods:

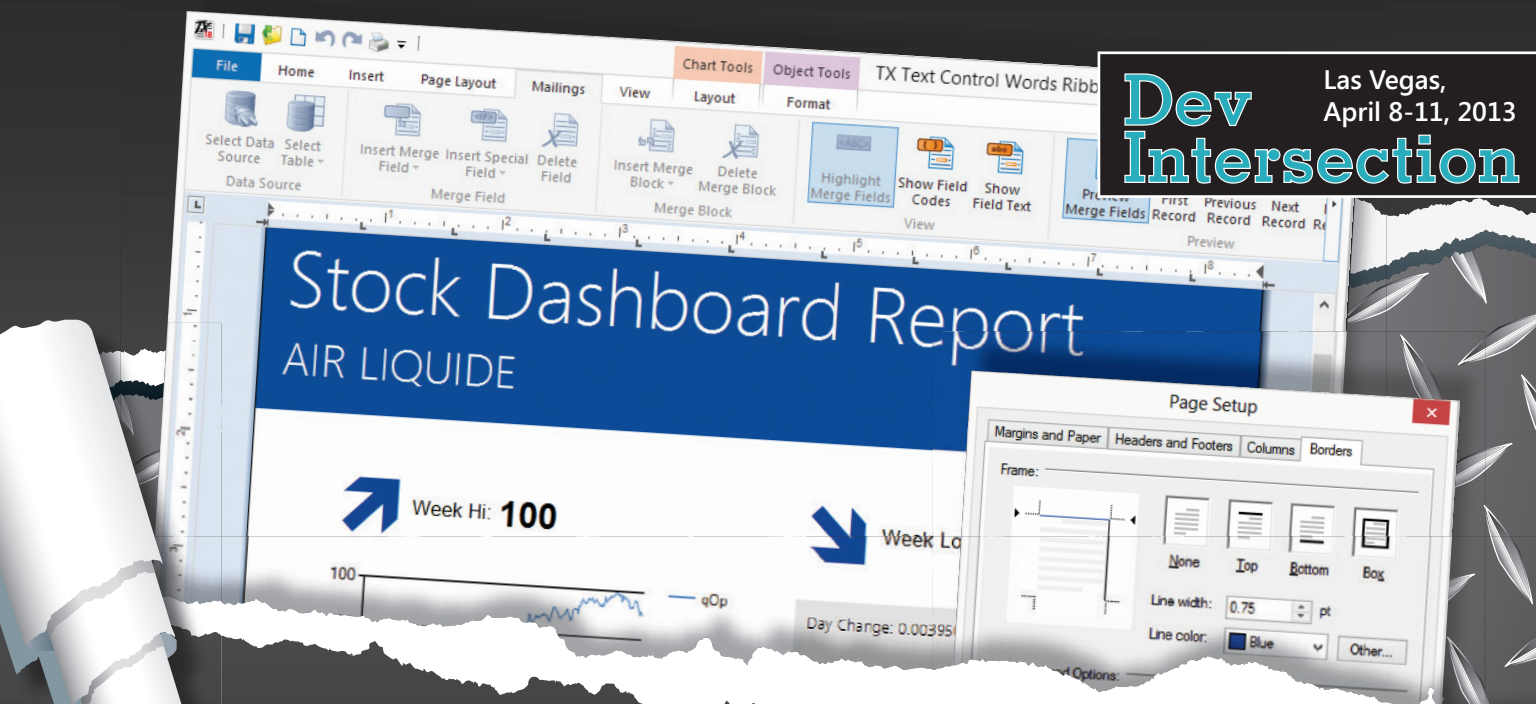
- BeginDraw: This opens the device context for drawing.
- EndDraw: This closes the device context.

Figure 2 **Creating the DirectX Device-Specific Resources**

```
// Somewhere in a header you have defined the following:
Microsoft::WRL::ComPtr<ISurfaceImageSourceNative> m_sisNative;

// Direct3D device.
Microsoft::WRL::ComPtr<ID3D11Device> m_d3dDevice;

// Direct2D objects.
Microsoft::WRL::ComPtr<ID2D1Device> m_d2dDevice;
Microsoft::WRL::ComPtr<ID2D1DeviceContext> m_d2dContext;
// ...

void MyImageSourceType::CreateDeviceResources()
{
  // This flag adds support for surfaces with a different color channel ordering
  // from the API default. It's required for compatibility with Direct2D.
  UINT creationFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;

#if defined(_DEBUG)
  // If the project is in a debug build, enable debugging via SDK Layers.
  creationFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

  // This array defines the set of DirectX hardware feature levels this
  // app will support. Note the ordering should be preserved.
  // Don't forget to declare your application's minimum required
  // feature level in its description. All applications are assumed
  // to support 9.1 unless otherwise stated.
  const D3D_FEATURE_LEVEL featureLevels[] =
  {
    D3D_FEATURE_LEVEL_11_1,
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_10_0,
    D3D_FEATURE_LEVEL_9_3,
    D3D_FEATURE_LEVEL_9_2,
    D3D_FEATURE_LEVEL_9_1,
  };

  // Create the Direct3D 11 API device object.

  D3D11CreateDevice(
    nullptr,
    D3D_DRIVER_TYPE_HARDWARE,
    nullptr,
    creationFlags,
    featureLevels,
    ARRAYSIZE(featureLevels),

    // Set this to D3D_SDK_VERSION for Windows Store apps.
    D3D11_SDK_VERSION,

    // Returns the Direct3D device created in a global var.
    &m_d3dDevice,
      nullptr,
      nullptr);

    // Get the Direct3D API device.
    ComPtr<IDXGIDevice> dxgiDevice;
    m_d3dDevice.As(&dxgiDevice);

    // Create the Direct2D device object and a
    // corresponding device context.
    D2D1CreateDevice(
      dxgiDevice.Get(),
      nullptr,
      &m_d2dDevice);

    m_d2dDevice->CreateDeviceContext(
      D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
      &m_d2dContext);

    // Associate the DXGI device with the SurfaceImageSource.
    m_sisNative->SetDevice(dxgiDevice.Get());
}
```

WINDOWS FORMS | WPF | ASP.NET

FLOW TYPE LAYOUT REPORTING



Dev Intersection
Las Vegas, April 8-11, 2013

- 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 3 **Drawing to the DirectX Surface**

```
void MyImageSourceType::BeginDraw(Windows::Foundation::Rect updateRect)
{
  POINT offset;
  ComPtr<IDXGISurface> surface;

  // Express target area as a native RECT type.
  RECT updateRectNative;
  updateRectNative.left = static_cast<LONG>(updateRect.Left);
  updateRectNative.top = static_cast<LONG>(updateRect.Top);
  updateRectNative.right = static_cast<LONG>(updateRect.Right);
  updateRectNative.bottom = static_cast<LONG>(updateRect.Bottom);

  // Begin drawing - returns a target surface and an offset
  // to use as the top-left origin when drawing.
  HRESULT beginDrawHR = m_sisNative->BeginDraw(
    updateRectNative, &surface, &offset);

  if (beginDrawHR == DXGI_ERROR_DEVICE_REMOVED ||
    beginDrawHR == DXGI_ERROR_DEVICE_RESET)
  {
    // If the device has been removed or reset, attempt to
    // re-create it and continue drawing.
    CreateDeviceResources();
    BeginDraw(updateRect);
  }


  // Create render target.
  ComPtr<ID2D1Bitmap1> bitmap;
  m_d2dContext->CreateBitmapFromDxgiSurface(
    surface.Get(),
    nullptr,
    &bitmap);

  // Set context's render target.
  m_d2dContext->SetTarget(bitmap.Get());

  // Begin drawing using D2D context.
  m_d2dContext->BeginDraw();

  // Apply a clip and transform to constrain updates to the target update
  // area. This is required to ensure coordinates within the target surface
  // remain consistent by taking into account the offset returned by
  // BeginDraw, and can also improve performance by optimizing the area
  // that's drawn by D2D. Apps should always account for the offset output
  // parameter returned by BeginDraw, because it might not match the passed
  // updateRect input parameter's location.
  m_d2dContext->PushAxisAlignedClip(
    D2D1::RectF(
      static_cast<float>(offset.x),
      static_cast<float>(offset.y),
      static_cast<float>(offset.x + updateRect.Width),
      static_cast<float>(offset.y + updateRect.Height)),
      D2D1_ANTIALIAS_MODE_ALIASED);

    m_d2dContext->SetTransform(
      D2D1::Matrix3x2F::Translation(
        static_cast<float>(offset.x),
        static_cast<float>(offset.y)
        )
    );
}

// End drawing updates started by a previous BeginDraw call.
void MyImageSourceType::EndDraw()
{
  // Remove the transform and clip applied in BeginDraw because
  // the target area can change on every update.
  m_d2dContext->SetTransform(D2D1::IdentityMatrix());
  m_d2dContext->PopAxisAlignedClip();

  // Remove the render target and end drawing.
  m_d2dContext->EndDraw();

  m_d2dContext->SetTarget(nullptr);

  m_sisNative->EndDraw();
}
```

Note: I've chosen the method names BeginDraw and EndDraw for a sort of loose correspondence with the ISurfaceImageSourceNative methods. This pattern is for convenience and isn't enforced.

My BeginDraw method (or other draw-initialization method I define on the derived type) must, at some point, call ISurfaceImageSourceNative::BeginDraw. (For optimizations, you can add a parameter for a sub-rectangle with the region of the image to update.) Likewise, the EndDraw method should call ISurfaceImageSourceNative::EndDraw.

The BeginDraw and EndDraw methods, in this case, could look something like the code shown in **Figure 3**.

Note that my BeginDraw method takes a Rect primitive as input, which is mapped to a native RECT type. This RECT defines the region of the screen that I intend to draw into with a corresponding SurfaceImageSource. BeginDraw, however, can only be called once at a time; I'll have to queue up the BeginDraw calls for each SurfaceImageSource, one after the other.

Also notice that I initialize a reference to an IDXGISurface, and an offset POINT struct that contains the (x, y) offset coordinates of the RECT I'll draw into the IDXGISurface with respect to the upper left. This surface pointer and offset are returned from ISurfaceImageSourceNative::BeginDraw to provide the IDXGISurface for drawing. Future calls in the example create a bitmap from the received surface pointer and draw into it with Direct2D calls. When ISurfaceImageSourceNative::EndDraw is called in the EndDraw overload, a completed image is the final result—an image that will be available to draw to the XAML image element or primitive.

Let's take a look at what I've got:

- A type that I derived from SurfaceImageSource.
- Methods on my derived type that define its drawing behavior to a provided RECT on the screen.

Figure 4 **Inheriting from VirtualSurfaceImageSource**

```
public ref class MyImageSourceType sealed :
Windows::UI::Xaml::Media::Imaging::VirtualSurfaceImageSource
{
  // ...
  MyImageSourceType::MyImageSourceType(
    int pixelWidth,
    int pixelHeight,
    bool isOpaque
    ) : VirtualSurfaceImageSource(pixelWidth, pixelHeight, isOpaque)
  {
    // Global variable that contains the width, in pixels,
    // of the SurfaceImageSource.
    m_width = pixelWidth;

    // Global variable that contains the height, in pixels,
    // of the SurfaceImageSource.
    m_height = pixelHeight;

    CreateDeviceIndependentResources(); // See below.
    CreateDeviceResources(); //Set up the DXGI resources.
  }
  // ...

  void MyImageSourceType::CreateDeviceIndependentResources()
  {
    // Query for IVirtualSurfaceImageSourceNative interface.
    reinterpret_cast<IUnknown*>(this)->QueryInterface(
      IID_PPV_ARGS(&m_vsisNative));
  }
  // ...
}
```

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.

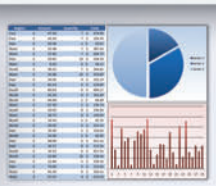
Excel Reporting for ASP.NET, WinForms, WPF and Silverlight

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

Figure 5 **Setting up a Callback for a VirtualSurfaceImageSource**

```
class MyVisibleSurfaceDrawingType :
  public IVirtualSurfaceUpdatesCallbackNative
{
// ...
private:
  virtual HRESULT STDMETHODCALLTYPE UpdatesNeeded() override;
}

// ...

HRESULT STDMETHODCALLTYPE MyVisibleSurfaceDrawingType::UpdatesNeeded()
{
  // ... perform drawing here ...
}
void MyVisibleSurfaceDrawingType::Initialize()
{
  // ...
  m_vsisNative->RegisterForUpdatesNeeded(this);
  // ...
}
```

- The DirectX graphics resources I need to perform the drawing.
- An association between the DirectX graphics device and the SurfaceImageSource.

What I still need is:

- Some code that does the *actual* image rendering into a RECT.
- A connection between the specific <Image> instance (or primitive) in the XAML and the SurfaceImageSource instance, to be invoked by the app.

The code for the drawing behavior is up to me, and it's probably simplest to implement it on my SurfaceImageSource type as a specific public method that can be called from the codebehind.

The rest is easy. In the codebehind for my XAML, I add this code to my constructor:

```
// An image source derived from SurfaceImageSource,
// used to draw DirectX content.
MyImageSourceType^ _SISDXsource = ref new
  MyImageSourceType((int)MyDxImage->Width, (int)MyDxImage->Height, true);
// Use MyImageSourceType as a source for the Image control.
MyDxImage->Source = _SISDXsource;
```

And add an event handler in the same codebehind, similar to this:

```
private void MainPage::MyCodeBehindObject_Click(
  Object^ sender, RoutedEventArgs^ e)
{
  // Begin updating the SurfaceImageSource.
  SISDXsource->BeginDraw();

  // ... Your DirectX drawing/animation calls here ...
  // such as _SISDXsource->
  // DrawTheMostAmazingSpinning3DShadedCubeEver();
  // ...

  // Stop updating the SurfaceImageSource and draw its contents.
  SISDXsource->EndDraw();
}
```

(Alternatively, if I'm not deriving from SurfaceImageSource, I could place the calls to BeginDraw and EndDraw inside a method—such as DrawTheMostAmazingSpinning3DShadedCubeEver from the previous code snippet—on my drawing object.)

Now, if I'm using a XAML primitive, such as Rect or Ellipse, I create an ImageBrush and attach the SurfaceImageSource to it, like this (where MySISPrimitive is a XAML graphics primitive):

```
// Create a new image brush and set the ImageSource
// property to your SurfaceImageSource instance.
ImageBrush^ myBrush = new ImageBrush();
myBrush->ImageSource = _SISDXsource;
MySISPrimitive->Fill = myBrush;
```

And that's it! To recap, the process in my example is:

1. Choose a XAML imaging element, such as an Image, ImageBrush or graphics primitive (Rect, Ellipse and so on), that I intend to render into. Also, determine if the surface will provide an animated image. Place it in my XAML.
2. Create the DirectX device and device contexts (typically Direct2D or Direct3D, or both) that will be used for drawing operations. Also, using COM, acquire a reference to the ISurfaceImageSourceNative interface that underpins the SurfaceImageSource runtime type and associate the graphics device with it.
3. Create a type that derives from SurfaceImageSource, and which has code that calls ISurfaceImageSource::BeginDraw and ISurfaceImageSource::EndDraw.
4. Add any specific drawing operations as methods on the SurfaceImageSource type.
5. For Image surfaces, connect the Source property to a SurfaceImageSource type instance. For graphics primitive surfaces, create an ImageBrush and assign a SurfaceImageSource instance to the ImageSource property, and then use that brush with the primitive's Fill property (or any property that accepts an ImageSource or ImageBrush).
6. Call the draw operations on the SurfaceImageSource instances from event handlers. For animated images, ensure that the frame-draw operations are interruptible.

I can use SurfaceImageSource for 2D and 3D game scenarios if the scene and shaders are simple enough. For example, a graphically middleweight strategy game (think "Civilization 4") or simple dungeon crawler could render the visuals to a SurfaceImageSource.

Also, note that I can create the derived SurfaceImageSource type in C++ in a separate DLL and use that type from a different, non-C++ language projection. In this case, I could confine my renderer and methods to C++, and build my app infrastructure and codebehinds in, say, C#. Model-View-ViewModel (MVVM) rules!

Which brings us to the limitations:

- The control that displays the SurfaceImageSource is designed for fixed-size surfaces.
- The control that displays the SurfaceImageSource isn't performance-optimized for arbitrarily large surfaces, especially surfaces that can be dynamically panned or zoomed.
- The control refresh is handled by the WinRT XAML framework view provider, which occurs when the framework refreshes. For real-time, high-fidelity graphics scenarios, this can impact performance noticeably (meaning it's not well-suited for your hot new shader-intensive intergalactic battle game).

This brings us to VirtualSurfaceImageSource (and, eventually, SwapChainBackgroundPanel). Let's take a look at the former.

VirtualSurfaceImageSource and Interactive Control Rendering

VirtualSurfaceImageSource is an extension of SurfaceImageSource, but it's designed for image surfaces that might be resized by the user, especially images that can be sized larger than the screen or moved partially offscreen, or that might have other images or

XAML elements obscuring part or all of them. It works particularly well with apps in which the user regularly pans or zooms an image that's potentially larger than the screen, for example, a map control or an image viewer.

The process for VirtualSurfaceImageSource is identical to that for SurfaceImageSource as presented earlier, only you use the VirtualSurfaceImageSource type instead of SurfaceImageSource, and the IVirtualImageSourceNative interface implementation instead of the ISurfaceImageSourceNative one.

> The graphics interface can have only one operation on the UI thread at a time.

That means I change my code from the prior example to:

- Use VirtualSurfaceImageSource instead of SurfaceImageSource. In the code samples following, I'll have my base image source type class, MyImageSourceType, derive from VirtualSurfaceImageSource.
- Query for the method implementation on the underlying IVirtualSurfaceImageSourceNative interface.

See **Figure 4** for an example.

Oh, and there's one other very important difference: I must implement a callback that's invoked whenever a "tile" (a defined rectangular region, not to be confused with Windows 8 UI tiles) of the surface becomes visible and needs to be drawn. These tiles are managed by the framework when an app creates an instance of VirtualSurfaceImageSource, and you don't control their parameters. Rather, behind the scenes, a large image is subdivided into these tiles, and the callback is invoked whenever a portion of one of these tiles becomes visible to the user and requires an update.

To use this mechanism, I first need to implement an instantiable type that inherits from the IVirtualSurfaceUpdatesCallbackNative interface, and register an instance of that type by passing it to IVirtualSurfaceImageSource::RegisterForUpdatesNeeded, as shown in **Figure 5**.

Figure 6 **Handling Updates to the Control Size or Visibility**

```
POINT offset;
ComPtr<IDXGISurface> dynamicSurface;

// Set offset.

// Call the following code once for each tile RECT that
// needs to be updated.

HRESULT beginDrawHR = m_vsisNative->
  BeginDraw(updateRect, &dynamicSurface, &offset);

if (beginDrawHR == DXGI_ERROR_DEVICE_REMOVED ||
  beginDrawHR == DXGI_ERROR_DEVICE_RESET)
{
  // Handle the change in the graphics interface.
}
else
{
  // Draw to IDXGISurface for the updated RECT at the provided offset.
}
```

The drawing operation is implemented as the UpdatesNeeded method from the IVirtualSurfaceUpdatesCallbackNative interface. If a specific region has become visible, I must determine which tiles should be updated. I do this by calling IVirtualSurfaceImageSourceNative::GetRectCount and, if the count of updated tiles is greater than zero, getting the specific rectangles for those updated tiles with IVirtualSurfaceImageSourceNative::GetUpdateRects and updating each one:

```
HRESULT STDMETHODCALLTYPE MyVisibleSurfaceDrawingType::UpdatesNeeded()
{
  HRESULT hr = S_OK;

  ULONG drawingBoundsCount = 0;

  m_vsisNative->GetUpdateRectCount(&drawingBoundsCount);
  std::unique_ptr<RECT[]> drawingBounds(new RECT[drawingBoundsCount]);
  m_vsisNative->GetUpdateRects(drawingBounds.get(), drawingBoundsCount);

  for (ULONG i = 0; i < drawingBoundsCount; ++i)
  {
    // ... per-tile drawing code here ...
  }

}
```

I can get the VirtualSurfaceImageSource-defined parameters for these tiles as RECT objects. In the preceding example, I get an array of RECT objects for all the tiles that need updates. Then I use the values for the RECTs to redraw the tiles by supplying them to VirtualSurfaceImageSource::BeginDraw.

Again, as with SurfaceImageSource, I initialize a pointer to the IDXGISurface, and I call the BeginDraw method on IVirtualSurfaceImageSourceNative (the underlying native interface implementation) to get the current surface into which to draw. The offset, however, refers to the (x, y) offset for the target RECT, rather than the image element as a whole.

For each RECT to update, I call code that looks like **Figure 6**.

Again, I can't parallelize these calls, because the graphics interface can have only one operation on the UI thread at a time. I can process each tile RECT in serial, or I can call IVirtualSurfaceImageSourceNative::BeginDraw with a unioned area of all RECTs for a single draw update. This is up to the developer.

Finally, I call IVirtualSurfaceImageSourceNative::EndDraw after I update each changed tile RECT. When the last updated tile is processed, I'll have a completely updated bitmap to provide to the corresponding XAML image or primitive, just as I did in the SurfaceImageSource example.

And that's it! This form of DirectX-XAML interop is great when users don't care about low-latency input for real-time 3D graphics, as they might in a detailed, real-time game. It's also awesome for graphics-rich apps and controls and more asynchronous (read: turn-based) games.

In the follow-up article, I'll take a look at the flipside of this approach: drawing XAML on top of the DirectX swap chain and the work needed to make the XAML framework play nice with a custom DirectX view provider. Stay tuned! ■

Doug Erickson *is a senior programming writer at Microsoft, working in Windows Content Services and specializing in DirectX and Windows Store game development. He hopes you'll make lots of amazing Windows Store DirectX games and become famous.*

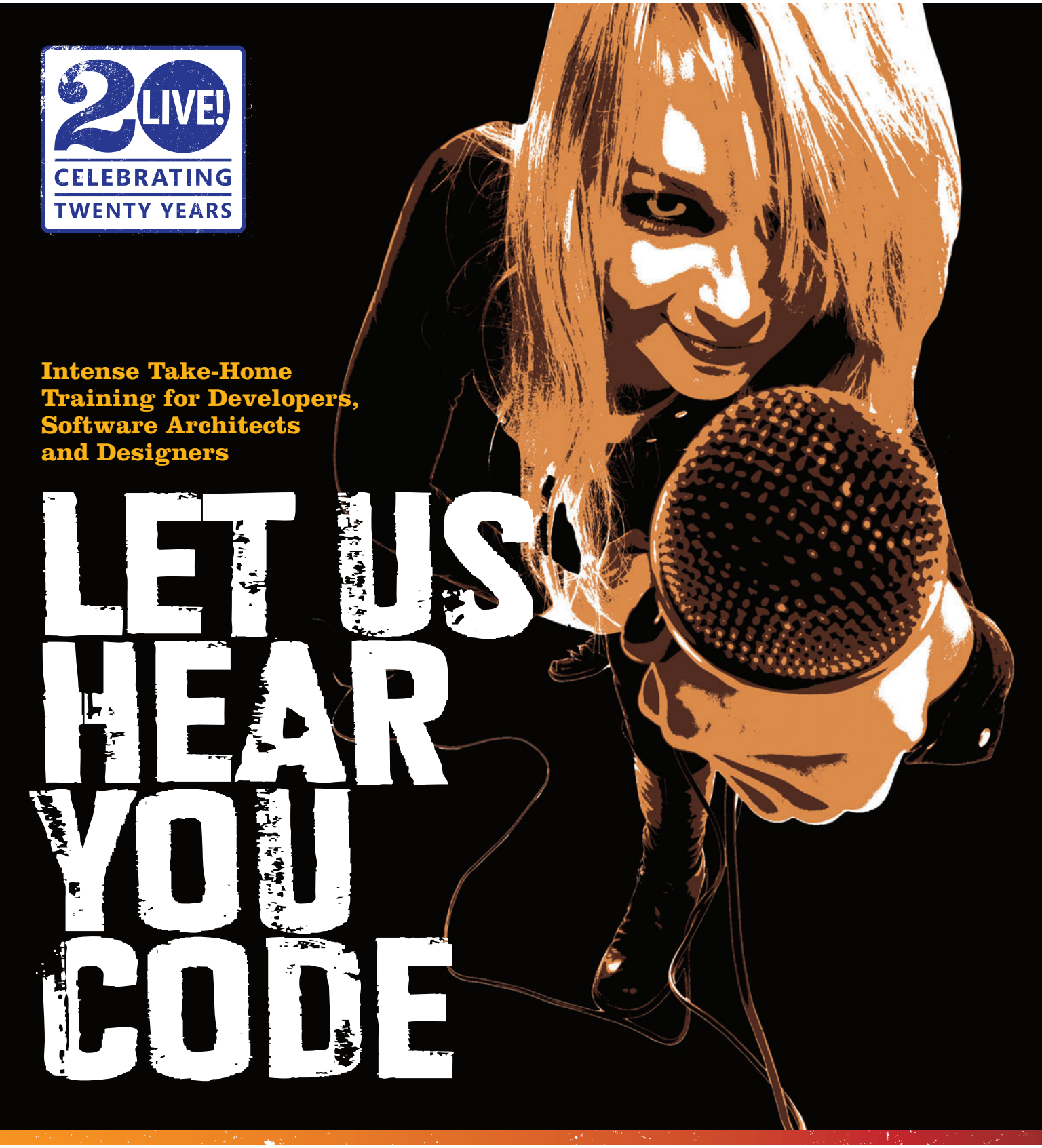
Thanks to the following technical experts for reviewing this article: *Jesse Bishop and Bede Jordan*

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

Seats are filling up fast... register today!

Use Promo Code LVMAR4

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
- Developer Deep Dive: SQL Server
- Windows 8 / WinRT
- WPF / Silverlight
- Visual Studio 2012 / .NET 4.5

CO-LOCATED WITH

Modern Apps LIVE!

MODERN APPS FROM START TO FINISH

www.modernappslive.com

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 - What's New in Azure for Developers - *Vishwas Lele*
10:45 AM	12:00 PM	**W06** - Cross Win8/WP8 Apps - *Ben Dewey*
		W07 - In Depth Azure IaaS - *Vishwas Lele*
12:00 PM	2:30 PM	Round Table 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** - Start Developing for Microsoft Office 365 SharePoint Online - *Darrin Bishop*
T13 - Chalk Talk: Beyond Hello World - A Practical Introduction to Node.js - *Rick Garibay*	**T14** - Chalk Talk: SharePoint 2013 Search - A Developer's Perspective - *Ryan McIntyre*	
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** - WCF Data Services - Getting Started Guide - *Sergey Barskiy*	**W10** - Use 2012 (and Beyond) Technology with SharePoint 2010 - *Ryan McIntyre*
W13 - Chalk Talk: NoSQL for the SQL Guy - *Ted Neward*	**W14** - Chalk Talk: Demystifying the Microsoft UI Technology Roadmap - *Brian Noyes*	
W17 - ASP.NET MVC - AJAX in your Views - *Walt Ritscher*	**W18** - Entity Framework Code First End to End - *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 HTML5 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 LVMAR4

Exploring the JavaScript API for Office: Data Access and Events

Stephen Oliver and Eric Schmidt

This article is the second in a series of in-depth walk-throughs of the JavaScript API for Office. Part 1 (available at msdn.microsoft.com/magazine/jj891051) provides a broad overview of the object model. This article picks up where part 1 left off, with a detailed walk-through on how to access file content and a review of the event model.

Throughout this series, we often make reference to the JavaScript API for Office reference 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).

Accessing Office File Content from an App for Office

The JavaScript API for Office provides several basic ways for accessing data in an Office file: You can either get or set the currently selected data, or you can get the entire file. This level of data access might sound simple and, in truth, both techniques are pretty simple to use. However, there's a wide degree of flexibility and customization within both techniques, providing you with a lot of possibilities for your apps.

In addition to access to selected data or the entire file, the JavaScript API for Office also allows you to bind to data or to manipulate custom XML parts in the document. We'll look more closely at these techniques for working with Office content.

This article discusses:

- Accessing Office file content
- Getting and setting selected data
- Getting the entire content of a file
- Getting different types of data from a Project application
- Events in an app for Office
- Key scenarios in the event model
- Document-level selection and data changed events
- Setting changed events

Technologies discussed:

JavaScript API for Office

Getting and Setting Selected Data

As we mentioned previously, the Document object gives an app access to the data in the file. For task pane and content apps, we can get or set selected content in an Office file using the Document.getSelectedDataAsync and Document.setSelectedDataAsync methods.

The two methods can manipulate several types of data formats that we control when we call them. Both the getSelectedDataAsync and setSelectedDataAsync methods have a parameter, coercionType, which takes a constant from the Office.CoercionType enumeration. The coercionType parameter specifies the data format of the content to get or set. Depending on the value of the coercionType parameter, we can select data as plain text, a table, a matrix, HTML or even "raw" Office Open XML (OOXML). (Note that getting and setting text as HTML or OOXML are only supported in Word 2013 as of press time.)

You don't always have to specify a coercionType when using getSelectedDataAsync and setSelectedDataAsync. The coercionType

is inferred from context whenever possible. For example, if you pass a string literal into a call to setSelectedDataAsync, then the default coercionType is "text." If you passed the same data in as an array of arrays, then the default coercionType would be "matrix."

We'll give some examples of how powerful these simple methods can be, primarily using the setSelectedDataAsync method. We'll start with some code that inserts some simple text into a Word document:

```
// Define some data to set in the document.
var booksToRead = "Anabasis by Xenophon; \n" +
  "Socrates' Apology by Plato; \n" +
  "The Illiad by Homer.";

// Set some data to the document as simple text.
Office.context.document.setSelectedDataAsync(
  booksToRead,
  { coercionType: Office.CoercionType.Text },
  function (result) {

    // Access the results, if necessary.
});
```

Figure 1 shows the result.

Now we'll change the example so that we insert the text as a "matrix" coercion type. A matrix is an array of arrays that's inserted as a simple range of cells (Excel) or simple table (Word).

When inserted into Word, the code inserts an unformatted, two-column table without a header. Each item in the first-level array represents a row in the resulting table; each item in a subarray contains data for a cell in the row:

```
// Define a matrix of data to set in the document.
var booksToRead = [["Xenophon", "Anabasis"],
  ["Plato", "Socrates' Apology"],
  ["Homer", "The Illiad"]];

// Set some data to the document as an unformatted table.
Office.context.document.setSelectedDataAsync(
  booksToRead,
  { coercionType: Office.CoercionType.Matrix },
  function (result) {

    // Access the results, if necessary.
});
```

Figure 2 shows the result.

In addition to the matrix coercion type, we can get or set data as a table using a TableData object. This allows us to provide a little bit more formatting to the results—in this particular case, a header row. We access the header row and content of a TableData object using the headers and rows properties, respectively.

Also, with the TableData object, you can specify a subset of data to insert using the startRow and startColumn parameters. This allows you to set data into a single column of an existing five-column table, as an example. We'll look at the startRow and startColumn parameters in greater depth in the next article in this series.

Note: If the selection in the document is a table, the selection shape must match the data being inserted (unless you specify the startRow and startColumn parameters). That is, if the data being inserted is a 2 x 2 table and the selection in the document is 3 x 2 cells in a table, then the method will fail. This also applies to inserting data as a matrix.

Like the matrix coercion type, the headers and rows properties return an array of arrays, where each item in the first array is a row of data and each item in a subarray contains one cell of data in the table, as you see in **Figure 3**.

Figure 4 shows the result of the code in **Figure 3**.

For the next example, we'll insert the same data, this time formatted as HTML and using the Office.CoercionType.HTML coercion. Now we can add additional formatting to the inserted data, such as CSS styles, as shown in **Figure 5**.

Figure 6 shows the result of the code in **Figure 5**.

Finally, we can also insert text into the document as OOXML, which lets us customize the data greatly and use many more advanced content types in Word (SmartArt or inline pictures, as two examples).

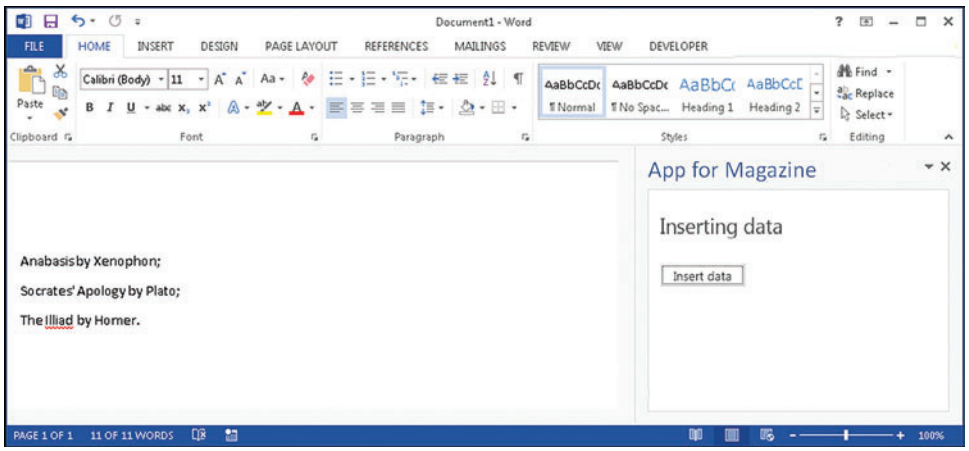


Figure 1 **Results of Inserting Data as Simple Text**

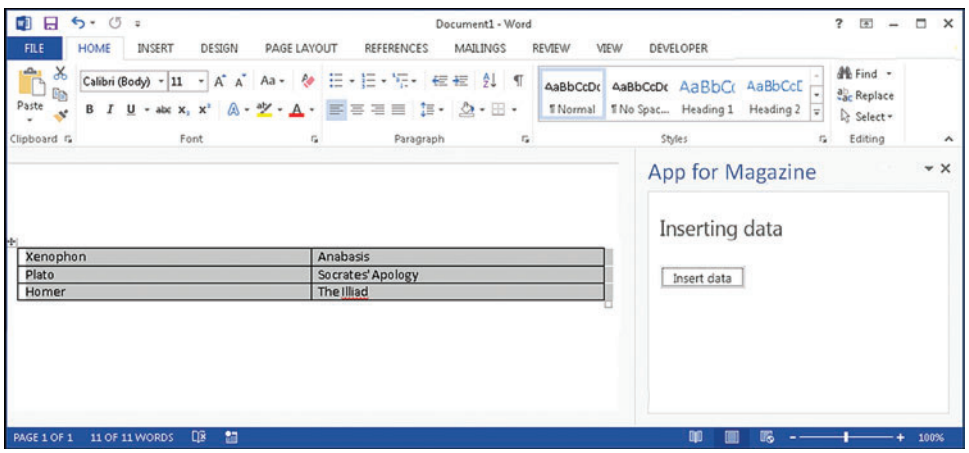


Figure 2 **Results of Inserting Data as a Matrix**

Figure 3 **Inserting Data into a Document as a Table**

```
// Define some tabular data to set in the document,
// including a header row.
var booksToRead = new Office.TableData();
booksToRead.headers = [["Author", "Title"]];
booksToRead.rows = [["Xenophon", "Anabasis"],
  ["Plato", "Socrates' Apology"],
  ["Homer", "The Illiad"]];

// Set some data to the document as a table with a header.
Office.context.document.setSelectedDataAsync(
  booksToRead,
  { coercionType: Office.CoercionType.Table },
  function (result) {

    // Access the results, if necessary.
});
```

The table of data that we've been working with, when represented as OOXML and stored in string literal, looks like the code in **Figure 7** (note: only part of the table is presented, for brevity).

This technique also requires a high degree of familiarity with XML, and the structures described by the OOXML standard (ECMA-376) in particular. When setting OOXML into a document, the data must be stored as a string (HTML Document objects can't be inserted) that contains all the necessary information—including relationships and related document parts in the file format package. Thus, when inserting a more advanced content type into Word using OOXML, you must remember to manipulate the OOXML data in accordance with the best practices of using OOXML and the Open Packaging Conventions.

In **Figure 8**, we've stepped around this issue by getting the data as OOXML *first*, concatenating our data with the OOXML from the document (by manipulating the received data and the new data as strings) and then inserting the OOXML back into the document. (Granted, part of the reason that this code works is because we haven't added any content that requires adding or changing any relationships or document parts in the file.)

Figure 9 shows the results of the code in **Figure 8**.

Note: One good way to learn about how to manipulate OOXML from an app is to add the content that you want to work with using the UI (for example, inserting SmartArt by clicking Insert | Illustrations | SmartArt), getting the OOXML for the content using getSelectedDataAsync and then reading the results. See the blog post, "Inserting images with apps for Office," at bit.ly/SeU3MS for more details.

Getting All of the Content in the File

Getting or setting data at the selection point is fine, but there are scenarios where it's necessary to get all of the content from a file. For example, an app might need to get all of the content in the document as text, parse it and then represent it in a bubble chart. As another example, an app might need to send all of the content in the file to a remote Web service for remote printing or faxing.

The JavaScript API for Office provides just the functionality for these scenarios. Using the JavaScript API, an app can create a copy of the file into which it's inserted, break the copy into sized chunks of data or "slices" (up to 4MB), and then read the data inside the slices.

The process for getting all of the content in the file essentially includes three steps:

1. For apps inserted into Word or PowerPoint, the app calls the Document.getFileAsync method, which returns a File object that corresponds to a copy of the file.
2. Once the app has a reference to the file, it can call the File.getSliceAsync method to access specific slices within the file, passing in the index of the slice to get. If this is done using a for loop, the calling code must be careful about how it handles closures.

One good way to learn about how to manipulate OOXML from an app is to add the content that you want to work with using the UI.

3. Finally, the app should close the File object once it's done with it, by calling the File.closeAsync method. Only two files are allowed to remain in memory at any time; attempting to open a third file using Document.getFileAsync raises the "An internal error has occurred" error.

In **Figure 10**, we get a Word document in 1KB chunks, iterate over each chunk in the file and then close the file when we're done with it.

For more information about how to get all of the file content from within an app for Office, see the documentation page, "How to: Get the whole document from an app for PowerPoint," at bit.ly/12Asi4x.

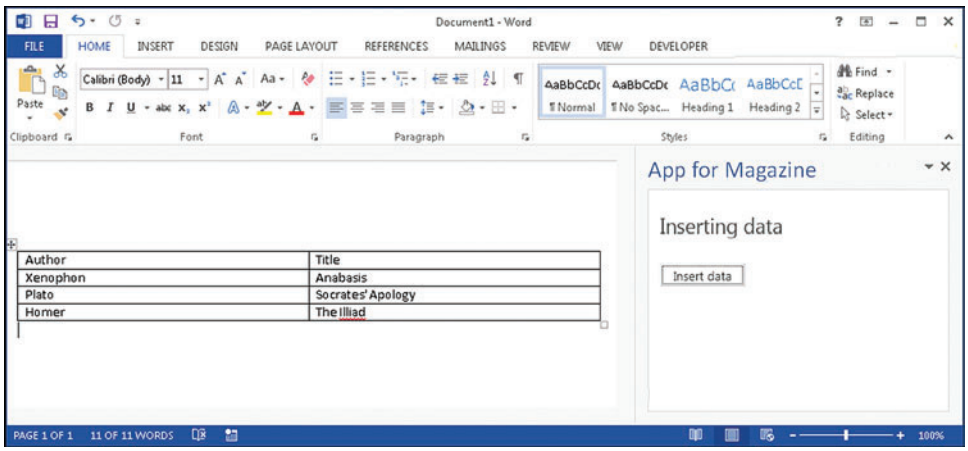


Figure 4 **Results of Inserting Data as a Table**

Figure 5 **Inserting Data into a Document as HTML**

```
// Define some HTML data to set in the document,
// including header row, text formatting and CSS styles.
var booksToRead =
  "<table style='font-family:Segoe UI'>" +
    "<thead style='background-color:#283E75;color:white'>" +
      "<tr><th>Authors</th><th>Books</th></tr>" +
    "</thead>" +
    "<tbody>" +
      "<tr><td>Xenophon</td><td><u>Anabasis</u></td></tr>" +
      "<tr><td>Plato</td><td><u>Socrates' Apology</u></td></tr>" +
      "<tr><td>Homer</td><td><u>The Iliad</u></td></tr>" +
    "</tbody>" +
  "</table>";

// Set some data to the document as a table with styles applied.
Office.context.document.setSelectedDataAsync(
  booksToRead,
  { coercionType: Office.CoercionType.Html },
    function (result) {

    // Access the results, if necessary.
});
```

Getting Task Data, View Data and Resource Data from a Project

For task pane apps inserted into Project, the JavaScript API for Office includes additional methods to read data for the active project and the selected task, resource or view. The project-15.js script extends office.js and also adds selection-change events for tasks, resources and views. For example, when the user selects a task in the Team Planner view, an app can integrate and display in one place the remaining work scheduled for that task, who is available to work on it, and related projects in other SharePoint task lists or in Project Server that can affect scheduling.

A task pane app inserted into a project only has read access to the content in the project.

A task pane app inserted into a project only has read access to the content in the project. But, because a task pane app is at heart a Web page, it can read from and write to external applications by using JavaScript and protocols such as Representational State Transfer (REST). For example, the Apps for Office and SharePoint documentation includes a sample app for Project Professional that uses jQuery with the OData reporting service in Project to compare total cost and work data for the active project with the averages for all projects in a Project Web App (see **Figure 11**).

For more information, see the documentation page, "How to: Create a Project app that uses REST with an on-premises Project Server OData service," at bit.ly/T80W2H.

Because ProjectDocument extends the Document object, the Office.context.document object captures a reference to the active project—similar to apps inserted into other host applications. The asynchronous methods available in Project have similar signatures to the other methods in the JavaScript API for Office. For example, the getProjectFieldAsync method has three parameters:

- **fieldId:** specifies the field to return in the object for the callback parameter. The Office.ProjectProjectFields enumeration includes 12 fields, such as the project GUID, start date, finish date and (if any) the Project Server URL or SharePoint task list URL.
- **asyncContext:** (optional) is any user-defined type returned in the asyncResult object.
- **callback:** contains a reference to a function that runs when the call returns, and contains options to handle success or failure.

As you can see in **Figure 12**, the methods specific to apps in Project are used similarly to apps hosted in other applications. In the script fragment, a locally defined function calls a routine that displays an error message in the app. The script doesn't use the asyncContext parameter.

Although the getProjectFieldAsync method can get only 12 fields for the general project, the getTaskFieldAsync method can get any one of 282 different fields for a task by using the ProjectTaskFields enumeration. And the getResourceFieldAsync method can get any one of 200 fields for a resource by using the ProjectResourceFields enumeration. More general methods in the ProjectDocument object include getSelectedDataAsync, which returns selected text data in any of the supported views, and getTaskAsync, which returns several items of general data for a selected task. Task pane apps can work with 16 different views in Project.

In addition, task pane apps in Project can add or remove event handlers when the user changes a view, selects a task or selects a resource.

Events in an App for Office

The JavaScript API for Office enables you to create more responsive apps through events. The event model for the API supports

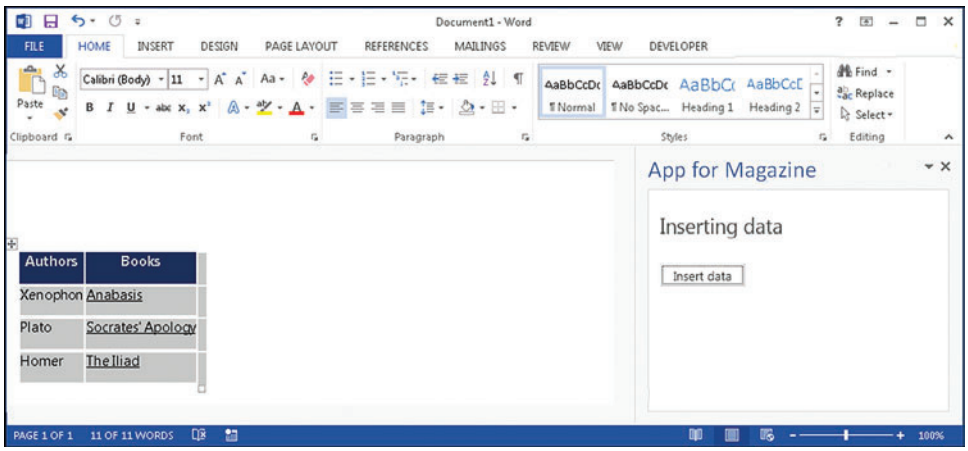


Figure 6 **Results of Inserting Data as HTML**

four key event scenarios fundamental to developing apps for Office (discussed later). Understanding these four scenarios will give you a solid grasp of the event model for the API.

That said, the event model for apps for Office is consistent throughout, so knowing the common design for event handling will complete your understanding of this important concept.

Concerning common design for events in the apps for Office API, the following objects have events associated with them:

- Binding
- CustomXMLPart
- Document
- RoamingSettings (mail apps)
- Settings

In addition to the events associated with it, each of the objects listed has two methods for dealing with its events:

- addHandlerAsync
- removeHandlerAsync

Because the removeHandlerAsync method simply unsubscribes a handler from an event and because its signature is nearly identical to that of addHandlerAsync, in the next section, we'll focus our attention on addHandlerAsync only.

Note: There's one very important difference between the removeHandlerAsync and addHandlerAsync methods. The handler parameter is *optional* for removeHandlerAsync. If unspecified, *all handlers* for the given event type are removed.

The AddHandlerAsync Method

The addHandlerAsync method wires up an event handler to the specified event and has the same signature for each object that implements it:

```
objectName.addHandlerAsync(eventType, handler [, options], callback);
```

Now we'll discuss the parameters for this method.

EventType Parameter The required eventType parameter takes an EventType enumeration, which tells the method which type of event to wire up.

Handler Parameter The eventType parameter is followed by the handler parameter. The handler can be either a named function or an anonymous inline function. Note that just like the event model

Figure 7 **An OOXML Snippet that Represents a Word Table, Stored as a JavaScript String Literal**

```
var newTable = "<w:tbl>" +
  "<w:tblPr>" +
    "<w:tblStyle w:val=\"TableGrid\"/>" +
    "<w:tblW w:w=\"0\" w:type=\"auto\"/>" +
    "<w:tblBorders>" +
      "<w:top w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "<w:left w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "<w:bottom w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "<w:right w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "<w:insideH w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "<w:insideV w:val=\"single\" w:sz=\"4\" w:space=\"0\"" +
        "w:color=\"283E75\"/>" +
      "</w:tblBorders>" +
    "<w:tblLook w:val=\"04A0\" w:firstRow=\"1\" w:lastRow=\"0\"" +
      "w:firstColumn=\"1\" w:lastColumn=\"0\""  +
      "w:noHBand=\"0\" w:noVBand=\"1\"/>" +
  "</w:tblPr>" +
  "<w:tblGrid>" +
    "<w:gridCol w:w=\"4675\"/>" +
    "<w:gridCol w:w=\"4675\"/>" +
  "</w:tblGrid>" +
  "<w:tr w:rsidR=\"00431544\" w:rsidTr=\"00620187\">" +
    "<w:tc>" +
      "<w:tcPr>" +
        "<w:tcW w:w=\"4675\" w:type=\"dxa\"/>" +
        "<w:shd w:val=\"clear\" w:color=\"auto\" w:fill=\"283E75\"/>" +
      "</w:tcPr>" +
      "<w:p w:rsidR=\"00431544\" w:rsidRPr=\"00236B94\""  +
        "w:rsidRDefault=\"00431544\" w:rsidP=\"00620187\">" +
        "<w:pPr>" +
          "<w:rPr>" +
            "<w:b/>" +
            "<w:color w:val=\"FEFEFE\"/>" +
          "</w:rPr>" +
        "</w:pPr>" +
        "<w:r w:rsidRPr=\"00236B94\">" +
          "<w:rPr>" +
            "<w:b/>" +
            "<w:color w:val=\"FEFEFE\"/>" +
          "</w:rPr>" +
          "<w:t>Authors</w:t>" +
        "</w:r>" +
      "</w:p>" +
    "</w:tc>" +
    "<w:tc>" +
      "<w:tcPr>" +
        "<w:tcW w:w=\"4675\" w:type=\"dxa\"/>" +
        "<w:shd w:val=\"clear\" w:color=\"auto\" w:fill=\"283E75\"/>" +
      "</w:tcPr>" +
      "<w:p w:rsidR=\"00431544\" w:rsidRPr=\"00236B94\"" +
        "w:rsidRDefault=\"00431544\" w:rsidP=\"00620187\">" +
        "<w:pPr>" +
          "<w:rPr>" +
            "<w:b/>" +
            "<w:color w:val=\"FEFEFE\"/>" +
          "</w:rPr>" +
        "</w:pPr>" +
        "<w:r w:rsidRPr=\"00236B94\">" +
          "<w:rPr>" +
            "<w:b/>" +
            "<w:color w:val=\"FEFEFE\"/>" +
          "</w:rPr>" +
          "<w:t>Books</w:t>" +
        "</w:r>" +
      "</w:p>" +
    "</w:tc>" +
  "</w:tr>" +
  "<w:tr w:rsidR=\"00431544\" w:rsidTr=\"00620187\">" +
    "<w:tc>" +
      "<w:tcPr>" +
        "<w:tcW w:w=\"4675\" w:type=\"dxa\"/>" +
      "</w:tcPr>" +
      "<w:p w:rsidR=\"00431544\" w:rsidRDefault=\"00431544\"" +
        "w:rsidP=\"00620187\">" +
        "<w:r>" +
          "<w:t>Xenophon</w:t>" +
        "</w:r>" +
      "</w:p>" +
    "</w:tc>" +
    "<w:tc>" +
      "<w:tcPr>" +
        "<w:tcW w:w=\"4675\" w:type=\"dxa\"/>" +
      "</w:tcPr>" +
      "<w:p w:rsidR=\"00431544\" w:rsidRDefault=\"00431544\"" +
        "w:rsidP=\"00620187\">" +
        "<w:r>" +
          "<w:t>Anabasis</w:t>" +
        "</w:r>" +
      "</w:p>" +
    "</w:tc>" +
  "</w:tr>" +

  // The rest of the code has been omitted for the sake of brevity.

"</w:tbl>";
```

Figure 8 **Inserting Data into a Document as a Table Using OOXML**

```
// Get the OOXML for the data at the point of insertion
// and add a table at the beginning of the selection.
Office.context.document.getSelectedDataAsync(
  Office.CoercionType.Ooxml,
  {
    valueFormat: Office.ValueFormat.Formatted,
    filterType: Office.FilterType.All
  },
  function (result) {
    if (result.status == "succeeded") {

      // Get the OOXML returned from the getSelectedDataAsync call.
      var selectedData = result.value.toString();

      // Define the new table in OOXML.
      var newTable = "<!--Details omitted for brevity.-->";

      // Find the '<w:body>' tag in the returned data—the tag
      // that represents the body content of the selection, contained
      // within the main document package part (/word/document.xml)—
      // and then insert the new table into the OOXML at that point.
      var newString = selectedData.replace(
        "<w:body>",
        "<w:body>" + newTable,
        "gi");

        // Insert the data back into the document with the table added.
        Office.context.document.setSelectedDataAsync(
          newString,
          { coercionType: Office.CoercionType.Ooxml },
          function () {
        });
    }
});
```

for many programming languages, the apps for Office runtime invokes the handler and passes in an event object argument as the only parameter. Also, if you use an inline anonymous function for the handler parameter, the only way to remove the handler is to remove *all* handlers from the event by calling removeHandlerAsync and not specifying the handler parameter.

Options Parameter Like all asynchronous functions in the apps for Office API, you can specify an object that contains optional parameters, but for all addHandlerAsync methods, the only optional parameter that you can specify is asyncContext. It's provided as a way to pass whatever data you want through the asynchronous method that you can retrieve inside the callback.

Callback Parameter The callback acts just as it does elsewhere throughout the apps for Office API with one significant exception: the value property of the AsyncResult object. As discussed earlier in this article, when the runtime invokes a callback, it passes in an AsyncResult object and you use the value property of the AsyncResult object to get the return value of the asynchronous call. In the case of callbacks in the addHandlerAsync method, the value of the AsyncResult object is always *undefined*.

Figure 13 demonstrates how to code the addHandlerAsync method for the DocumentSelectionChanged event (the code assumes you have a element with an id attribute value of "message").

When the app is initialized, the code in **Figure 13** wires up the onDocSelectionChanged and onDocSelectionChanged2 handler functions to the DocumentSelectionChanged event, showing that you can have more than one event handler for the same event. Both handlers simply write to the , "message," when the DocumentSelectionChanged event fires.

The calls to addHandlerAsync also include callbacks addHandlerCallback and addHandlerCallback2, respectively. The callbacks also write to the , "message," but are only called once when addHandlerAsync completes.

In the same way, you can use the addHandlerAsync method to wire up event handlers for any event in the JavaScript API for Office.

Key Scenarios in the Event Model for Apps for Office

As mentioned, in the JavaScript API for Office, there are four key event scenarios around which to order your understanding as you consider the event model for apps for Office. All events in the API will fall into one of these four key event scenarios:

- Office.initialize events
- Document-level selection change events
- Binding-level selection and data changed events
- Settings changed events

Office.initialize Events By far the most common event in the JavaScript API for Office that you'll encounter is the Office.initialize event. The initialize event happens for every app for Office that you create. In fact, it's the first part of your code the runtime executes.

If you look at the starter code that Visual Studio 2012 provides for any new app for Office project, you'll see the first lines of the starter code in the ProjectName.js file for your app wire up an event handler for the Office.initialize event, as shown here:

```
// This function is run when the app is ready to
// start interacting with the host application;
// it ensures the DOM is ready before adding click handlers to buttons.
Office.initialize = function (reason) { /* handler code */ };
```

As you know from the Object model hierarchy section in the previous article, the Office object is the topmost object in the JavaScript API for Office and represents the instance of your app at run time. The Initialize event fires when the apps for Office runtime is completely

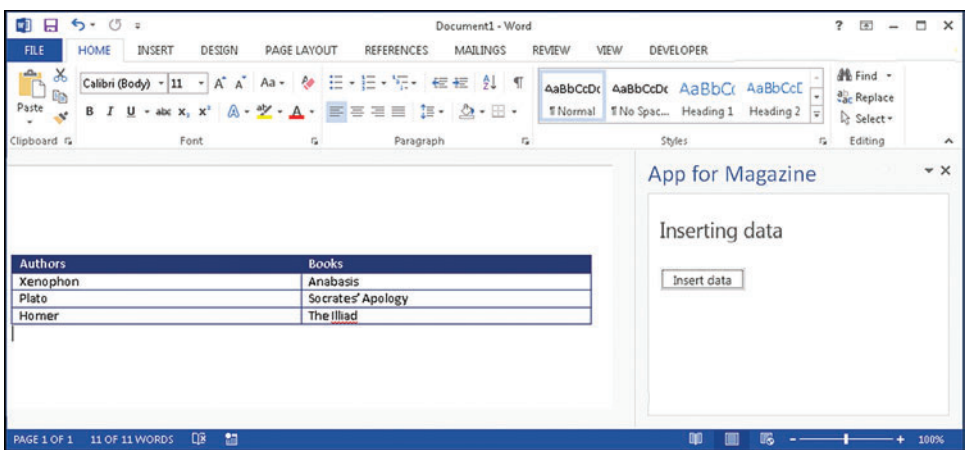


Figure 9 **Results of Inserting Data as OOXML**

Figure 10 **Getting All of the Content from a File as Text and Iterating over the Slices**

```
// Get all of the content from a Word document in 1KB chunks of text.
function getFileData() {
  Office.context.document.getFileAsync(
  Office.FileType.Text,
  {
    sliceSize: 1000
  },
  function (asyncResult) {
    if (asyncResult.status === 'succeeded') {

      var myFile = asyncResult.value,
        state = {
          file: myFile,
          counter: 0,
          sliceCount: myFile.sliceCount
        };

      getSliceData(state);
    }
  });
}

// Get a slice from the file, as specified by
// the counter contained in the state parameter.
function getSliceData(state) {
  state.file.getSliceAsync(
    state.counter,
    function (result) {

      var slice = result.value,
        data = slice.data;

      state.counter++;

      // Do something with the data.

      // Check to see if the final slice in the file has
      // been reached—if not, get the next slice;
      // if so, close the file.
      if (state.counter < state.sliceCount) {
        getSliceData(state);
      }
      else {
        closeFile(state);
      }
    });
}

// Close the file when done with it.
function closeFile(state) {
  state.file.closeAsync(
    function (results) {

      // Inform the user that the process is complete.
    });
}
```

loaded and ready for interaction with your app. So, the Initialize event event handler is essentially the "handshake" between your app and the runtime that has to occur before the rest of your code runs.

The function that you have to provide as the handler for the Office.initialize event takes a single argument—an InitializationReason enumeration. The InitializationReason enumeration has just two enumerations—Inserted and documentOpened:

- Inserted indicates that the app is being initialized because it was just inserted into the document.
- documentOpened means the app is being initialized because the document that already had the app inserted was just opened.

The runtime will pass in the InitializationReason enumeration as the only argument to your handler function. From there, you can branch how your code will react depending on the reason.

Here's an idea of how this might work:

```
Office.initialize = function (reason) {
  // Display initialization reason.
  if (reason == "inserted")
  write("The app was just inserted.");

  if (reason == "documentOpened")
  write(
    "The app is already part of the document.");
}

// Function that writes to a div with
// id='message' on the page.
function write(message){
  document.getElementById(
  'message').innerText += message;
}
```

Note: The preceding code snippet assumes you have a element with an id attribute value of "message."

Interestingly, you don't have to include anything inside the function that you supply as a handler, but the function *must* be present or your app will throw an error when it starts.

By the way, the event handler for the Office.initialize event is a good place to initialize other frameworks that you might use in your app, such as jQuery. Again, in the starter code that Visual Studio provides for new apps for Office projects, you'll see something like the code in **Figure 14**.

The jQuery .ready event is handled inside the Office.initialize event handler. This ensures the JavaScript API for Office is loaded and ready before the JQuery code calls into it.

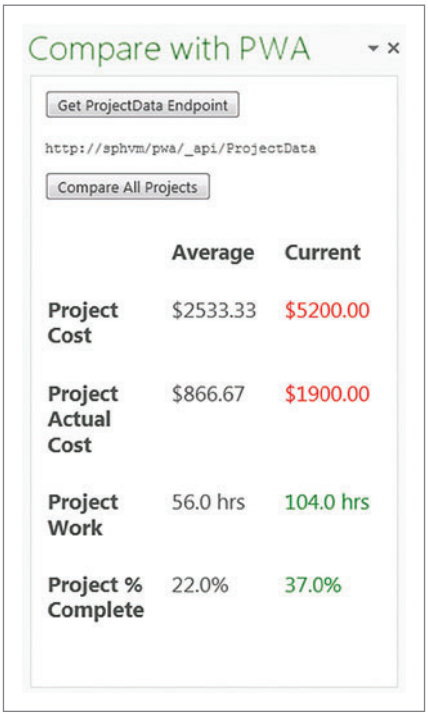


Figure 11 **A Task Pane App that Uses jQuery with an OData Reporting Service**

Document-Level Selection Change Events Document-level selection change events occur when the selection in the document moves from one selection to another. For example, when the user clicks away from the current selection in a Word document to a range of text, or object or location elsewhere in the document, an event is fired at the document level for the change in selection.

The following code illustrates how to respond to a change to the current selection:

```
function addEventHandlerToDocument() {
  Office.context.document.addHandlerAsync(
    Office.EventType.DocumentSelectionChanged,
    MyHandler);
}

function MyHandler(eventArgs) {
  doSomethingWithDocument(eventArgs.document);
```

Binding-Level Selection and Data Changed Events Bindings in the apps for Office object model are a way to consistently access a specific area in a document (or spreadsheet) by establishing a linkage, or binding, to a uniquely named region in the document. To work with a binding, you first create one

using one of the provided methods of the API. You can then refer to the specific binding that you created using its unique identifier.

Bindings also trigger events, and your app can respond to those events as needed. In particular, bindings fire an event when the *selection changes within* the binding area and when *data changes within* the binding area. The following two code snippets show how to handle changes to the selection and changes to data in a given binding (both assume you have a element with an id attribute value of "message").

Responding to the Binding.bindingSelectionChanged event:

```
function addEventHandlerToBinding() {
  Office.select("bindings#MyBinding").addHandlerAsync(
    Office.EventType.BindingSelectionChanged,
    onBindingSelectionChanged);
}

function onBindingSelectionChanged(eventArgs) {
  write(eventArgs.binding.id + " has been selected.");
}
// Function that writes to a div with id='message' on the page.
function write(message){
  document.getElementById('message').innerText += message;
}
```

Responding to the Binding.bindingDataChanged event:

```
function addEventHandlerToBinding() {
  Office.select("bindings#MyBinding").addHandlerAsync(
    Office.EventType.BindingDataChanged, onBindingDataChanged);
}

function onBindingDataChanged(eventArgs) {
  write("Data has changed in binding: " + eventArgs.binding.id);
}

// Function that writes to a div with id='message' on the page.
function write(message){
  document.getElementById('message').innerText += message;
}
```

Settings Changed Events The apps for Office object model provides a way for developers to persist settings that relate to their app. The Settings object acts as a property bag where custom app settings are stored as key/value pairs. The Settings object also has an event associated with it—Settings.settingsChanged—that fires when a stored setting is changed.

For more information about the Settings.settingsChanged event, see the MSDN documentation for the JavaScript API for Office at bit.ly/U92Sbe.

Next Up: More-Advanced Topics

In this second article of the series, we reviewed the basics of getting and setting Office file content from an app for Office. We showed how to get and set selection data and how to get all of the file data. We looked at how to get project, task, view and resource data from an app for Project. Finally, we looked at the events in the JavaScript API for Office and how to code against them.

Note: We'd like to thank Jim Corbin, programming writer in the Office Division, for contributing much of the content concerning apps for Project.

Next, we'll take a closer look at some more-advanced topics in the JavaScript API for Office: data bindings and custom XML parts. ■

Figure 12 **Getting the GUID of a Field from a Task Pane Inserted into a Project**

```
var _projectUid = "";

// Get the GUID of the active project.
function getProjectGuid() {
  Office.context.document.getProjectFieldAsync(
    Office.ProjectProjectFields.GUID,
    function (asyncResult) {
      if (asyncResult.status == Office.AsyncResultStatus.Succeeded) {
        _projectUid = asyncResult.value.fieldValue;
        }
      else {
        // Display error message to user.
      }
    }
  );
}
```

Figure 13 **Wiring up an Event Handler for the DocumentSelectionChanged Event, Using the Document.addHandlerAsync Method**

```
Office.initialize = function (reason) {
  $(document).ready(function () {
    Office.context.document.addHandlerAsync(
      Office.EventType.DocumentSelectionChanged, onDocSelectionChanged,
        addHandlerCallback);
      Office.context.document.addHandlerAsync(
        Office.EventType.DocumentSelectionChanged, onDocSelectionChanged2,
        addHandlerCallback2);
  });
};
function onDocSelectionChanged(docSelectionChangedArgs) {
  write("onDocSelectionChanged invoked each event.");
}
function onDocSelectionChanged2(docSelectionChangedArgs) {
  write("onDocSelectionChanged2 invoked each event.");
}
function addHandlerCallback(asyncResult) {
  write("addHandlerCallback only called once on app initialize.");
}
function addHandlerCallback2(asyncResult) {
  write("addHandlerCallback2 only called once on app initialize.");
}
function write(message) {$('#message').append(message + "\n");
```

Figure 14 **Initializing Other Frameworks Within the Office.initialize Event Handler**

```
Office.initialize = function (reason) {
  $(document).ready(function () {
    $('#getDataBtn').click(function () { getData('#selectedDataTxt'); });

    // If setSelectedDataAsync method is supported
    // by the host application, setDatabtn is hooked up
    // to call the method, else setDatabtn is removed.
    if (Office.context.document.setSelectedDataAsync) {
        $('#setDataBtn').click(function () { setData('#selectedDataTxt'); });
    }
    else {
      $('#setDataBtn').remove();
    }
  });
};
```

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*

Best Practices in Asynchronous Programming

Stephen Cleary

These days there's a wealth of information about the new async and await support in the Microsoft .NET Framework 4.5. This article is intended as a "second step" in learning asynchronous programming; I assume that you've read at least one introductory article about it. This article presents nothing new, as the same advice can be found online in sources such as Stack Overflow, MSDN forums and the async/await FAQ. This article just highlights a few best practices that can get lost in the avalanche of available documentation.

The best practices in this article are more what you'd call "guidelines" than actual rules. There are exceptions to each of these guidelines. I'll explain the reasoning behind each guideline so that it's clear when it does and does not apply. The guidelines are summarized in **Figure 1**; I'll discuss each in the following sections.

Avoid Async Void

There are three possible return types for async methods: Task, Task<T> and void, but the *natural* return types for async methods are just Task and Task<T>. When converting from synchronous to asynchronous code, any method returning a type T becomes an async method returning Task<T>, and any method returning void becomes an async method returning Task. The following code snippet illustrates a synchronous void-returning method and its asynchronous equivalent:

```
void MyMethod()
{
  // Do synchronous work.
  Thread.Sleep(1000);
}

async Task MyMethodAsync()
{
  // Do asynchronous work.
  await Task.Delay(1000);
}
```

Void-returning async methods have a specific purpose: to make asynchronous event handlers possible. Event handlers naturally return void, so async methods are allowed to return void so that you can have an asynchronous event handler. However, some semantics of an async void method are subtly different than the semantics of an async Task or async Task<T> method.

Async void methods have different error-handling semantics. When an exception is thrown out of an async Task or async Task<T> method, that exception is captured and placed on the Task object. With async void methods, there is no Task object, so any exceptions thrown out of an async Task method will be raised directly on the SynchronizationContext that was active when the async void method started. **Figure 2** illustrates that exceptions thrown from async void methods can't be caught naturally.

This article discusses:

- Avoiding async void methods
- Avoiding mixing of synchronous and asynchronous code
- Using ConfigureAwait for context-free code
- Solutions to common problems when using async/await

Technologies discussed:

Microsoft .NET Framework 4.5; Async/Await Keywords

Figure 1 **Summary of Asynchronous Programming Guidelines**

Name	Description	Exceptions
Avoid async void	Prefer async Task methods over async void methods	Event handlers
Async all the way	Don't mix blocking and async code	Console main method
Configure context	Use ConfigureAwait(false) when you can	Methods that require context

These exceptions *can* be observed using AppDomain.UnhandledException or a similar catch-all event for GUI/ASP.NET applications, but using those events for regular exception handling is a recipe for unmaintainability.

Async void methods have different composing semantics. Async methods returning Task or Task<T> can be easily composed using await, Task.WhenAny, Task.WhenAll and so on. Async methods returning void don't provide an easy way to notify the calling code that they've completed. It's easy to start several async void methods, but it's not easy to determine when they've finished. Async void methods *will* notify their SynchronizationContext when they start and finish, but a custom SynchronizationContext is a complex solution for regular application code.

Async void methods are difficult to test. Because of the differences in error handling and composing, it's difficult to write unit tests that call async void methods. The MSTest asynchronous testing support only works for async methods returning Task or Task<T>. It's possible to install a SynchronizationContext that detects when all async void methods have completed and collects any exceptions, but it's much easier to just make the async void methods return Task instead.

It's clear that async void methods have several disadvantages compared to async Task methods, but they're quite useful in one particular case: asynchronous event handlers. The differences in semantics make sense for asynchronous event handlers. They raise their exceptions directly on the SynchronizationContext, which is similar to how synchronous event handlers behave. Synchronous event handlers are usually private, so they can't be composed or directly tested. An approach I like to take is to minimize the code in my asynchronous event handler—for example, have it await an async Task method that contains the actual logic. The following code illustrates this approach, using async void methods for event handlers without sacrificing testability:

```
private async void button1_Click(object sender, EventArgs e)
{
  await Button1ClickAsync();
}

public async Task Button1ClickAsync()
{
  // Do asynchronous work.
  await Task.Delay(1000);
}
```

Async void methods can wreak havoc if the caller isn't expecting them to be async. When the return type is Task, the caller knows it's dealing with a future operation; when the return type is void, the caller might assume the method is complete by the time it returns. This problem can crop up in many unexpected ways. It's usually wrong to provide an async implementation (or override) of a void-returning method on an interface (or base class). Some events also assume that their handlers are complete when they return. One subtle trap is passing an async lambda to a method taking an Action parameter; in this case, the async lambda returns void and inherits all the problems of async void methods. As a general rule, async lambdas should only be used if they're converted to a delegate type that returns Task (for example, Func<Task>).

To summarize this first guideline, you should prefer async Task to async void. Async Task methods enable easier error-handling, composability and testability. The exception to this guideline is asynchronous event handlers, which *must* return void. This exception includes methods that are logically event handlers even if they're not literally event handlers (for example, ICommand.Execute implementations).

Async All the Way

Asynchronous code reminds me of the story of a fellow who mentioned that the world was suspended in space and was immediately challenged by an elderly lady claiming that the world rested on the back of a giant turtle. When the man enquired what the turtle was standing on, the lady replied, "You're very clever, young man, but it's turtles all the way down!" As you convert synchronous code to asynchronous code, you'll find that it works best if asynchronous code calls and is called by other asynchronous code—all the way down (or "up," if you prefer). Others have also noticed the spreading behavior of asynchronous programming and have called it "contagious"

Figure 2 **Exceptions from an Async Void Method Can't Be Caught with Catch**

```
private async void ThrowExceptionAsync()
{
  throw new InvalidOperationException();
}

public void AsyncVoidExceptions_CannotBeCaughtByCatch()
{
  try
  {
    ThrowExceptionAsync();
  }
  catch (Exception)
  {
    // The exception is never caught here!
    throw;
  }
}
```

Figure 3 **A Common Deadlock Problem When Blocking on Async Code**

```
public static class DeadlockDemo
{
  private static async Task DelayAsync()
  {
    await Task.Delay(1000);
  }

  // This method causes a deadlock when called in a GUI or ASP.NET context.
  public static void Test()
  {
    // Start the delay.
    var delayTask = DelayAsync();

    // Wait for the delay to complete.
    delayTask.Wait();
  }
}
```

Figure 4 **The Main Method May Call Task.Wait or Task.Result**

```
class Program
{
  static void Main()
  {
    MainAsync().Wait();
  }

  static async Task MainAsync()
  {
    try
    {
      // Asynchronous implementation.
      await Task.Delay(1000);
    }
    catch (Exception ex)
    {
      // Handle exceptions.
    }
  }
}
```

or compared it to a zombie virus. Whether turtles or zombies, it's definitely true that asynchronous code tends to drive surrounding code to also be asynchronous. This behavior is inherent in all types of asynchronous programming, not just the new async/await keywords.

"Async all the way" means that you shouldn't mix synchronous and asynchronous code without carefully considering the consequences. In particular, it's usually a bad idea to block on async code by calling Task.Wait or Task.Result. This is an especially common problem for programmers who are "dipping their toes" into asynchronous programming, converting just a small part of their application and wrapping it in a synchronous API so the rest of the application is isolated from the changes. Unfortunately, they run into problems with deadlocks. After answering many async-related questions on the MSDN forums, Stack Overflow and e-mail, I can say this is by far the most-asked question by async newcomers once they learn the basics: "Why does my partially async code deadlock?"

Figure 3 shows a simple example where one method blocks on the result of an async method. This code will work just fine in a console application but will deadlock when called from a GUI or ASP.NET context. This behavior can be confusing, especially considering that stepping through the debugger implies that it's the await that never completes. The actual cause of the deadlock is further up the call stack when Task.Wait is called.

The root cause of this deadlock is due to the way await handles contexts. By default, when an incomplete Task is awaited, the current "context" is captured and used to resume the method when the Task completes. This "context" is the current SynchronizationContext unless it's null, in which case it's the current TaskScheduler. GUI and ASP.NET applications have a SynchronizationContext that permits only one chunk of code to run at a time. When the await completes, it attempts to execute the remainder of the async method within the captured context. But that context already has a thread in it, which is (synchronously) waiting for the async method to complete. They're each waiting for the other, causing a deadlock.

Figure 5 **The "Async Way" of Doing Things**

To Do This ...	Instead of This ...	Use This
Retrieve the result of a background task	Task.Wait or Task.Result	await
Wait for any task to complete	Task.WaitAny	await Task.WhenAny
Retrieve the results of multiple tasks	Task.WaitAll	await Task.WhenAll
Wait a period of time	Thread.Sleep	await Task.Delay

Note that console applications don't cause this deadlock. They have a thread pool SynchronizationContext instead of a one-chunk-at-a-time SynchronizationContext, so when the await completes, it schedules the remainder of the async method on a thread pool thread. The method is able to complete, which completes its returned task, and there's no deadlock. This difference in behavior can be confusing when programmers write a test console program, observe the partially async code work as expected, and then move the same code into a GUI or ASP.NET application, where it deadlocks.

The best solution to this problem is to allow async code to grow naturally through the codebase. If you follow this solution, you'll see async code expand to its entry point, usually an event handler or controller action. Console applications can't follow this solution fully because the Main method can't be async. If the Main method were async, it could return before it completed, causing the program to end. **Figure 4** demonstrates this exception to the guideline: The Main method for a console application is one of the few situations where code may block on an asynchronous method.

Allowing async to grow through the codebase is the best solution, but this means there's a lot of initial work for an application to see real benefit from async code. There are a few techniques for incrementally converting a large codebase to async code, but they're outside the scope of this article. In some cases, using Task.Wait or Task.Result can help with a partial conversion, but you need to be aware of the deadlock problem as well as the error-handling problem. I'll explain the error-handling problem now and show how to avoid the deadlock problem later in this article.

Every Task will store a list of exceptions. When you await a Task, the first exception is re-thrown, so you can catch the specific exception type (such as InvalidOperationException). However, when you synchronously block on a Task using Task.Wait or Task.Result, all of the exceptions are wrapped in an AggregateException and thrown. Refer again to **Figure 4**. The try/catch in MainAsync will catch a specific exception type, but if you put the try/catch in Main, then it will always catch an AggregateException. Error handling is much easier to deal with when you don't have an AggregateException, so I put the "global" try/catch in MainAsync.

So far, I've shown two problems with blocking on async code: possible deadlocks and more-complicated error handling. There's also a problem with using blocking code within an async method. Consider this simple example:

```
public static class NotFullyAsynchronousDemo
{
  // This method synchronously blocks a thread.
  public static async Task TestNotFullyAsync()
  {
    await Task.Yield();
    Thread.Sleep(5000);
  }
}
```

This method isn't fully asynchronous. It will immediately yield, returning an incomplete task, but when it resumes it will synchronously block whatever thread is running. If this method is called

from a GUI context, it will block the GUI thread; if it's called from an ASP.NET request context, it will block the current ASP.NET request thread. Asynchronous code works best if it doesn't synchronously block. **Figure 5** is a cheat sheet of async replacements for synchronous operations.

To summarize this second guideline, you should avoid mixing async and blocking code. Mixed async and blocking code can cause deadlocks, more-complex error handling and unexpected blocking of context threads. The exception to this guideline is the Main method for console applications, or—if you're an advanced user—managing a partially asynchronous codebase.

Figure 6 **Handling a Returned Task that Completes Before It's Awaited**

```
async Task MyMethodAsync()
{
  // Code here runs in the original context.

  await Task.FromResult(1);

  // Code here runs in the original context.

  await Task.FromResult(1).ConfigureAwait(continueOnCapturedContext: false);

  // Code here runs in the original context.

  var random = new Random();
  int delay = random.Next(2); // Delay is either 0 or 1
  await Task.Delay(delay).ConfigureAwait(continueOnCapturedContext: false);

  // Code here might or might not run in the original context.
  // The same is true when you await any Task
  // that might complete very quickly.
}
```

Figure 7 **Having an Async Event Handler Disable and Re-Enable Its Control**

```
private async void button1_Click(object sender, EventArgs e)
{
  button1.Enabled = false;
  try
  {
    // Can't use ConfigureAwait here ...
    await Task.Delay(1000);
  }
  finally
  {
    // Because we need the context here.
    button1.Enabled = true;
  }
}
```

Figure 8 **Each Async Method Has Its Own Context**

```
private async Task HandleClickAsync()
{
  // Can use ConfigureAwait here.
  await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
}

private async void button1_Click(object sender, EventArgs e)
{
  button1.Enabled = false;
  try
  {
    // Can't use ConfigureAwait here.
    await HandleClickAsync();
  }
  finally
  {
    // We are back on the original context for this method.
    button1.Enabled = true;
  }
}
```

Configure Context

Earlier in this article, I briefly explained how the "context" is captured by default when an incomplete Task is awaited, and that this captured context is used to resume the async method. The example in **Figure 3** shows how resuming on the context clashes with synchronous blocking to cause a deadlock. This context behavior can also cause another problem—one of performance. As asynchronous GUI applications grow larger, you might find many small parts of async methods all using the GUI thread as their context. This can cause sluggishness as responsiveness suffers from "thousands of paper cuts."

To mitigate this, await the result of ConfigureAwait whenever you can. The following code snippet illustrates the default context behavior and the use of ConfigureAwait:

```
async Task MyMethodAsync()
{
  // Code here runs in the original context.

  await Task.Delay(1000);

  // Code here runs in the original context.

  await Task.Delay(1000).ConfigureAwait(
    continueOnCapturedContext: false);

  // Code here runs without the original
  // context (in this case, on the thread pool).
}
```

By using ConfigureAwait, you enable a small amount of parallelism: Some asynchronous code can run in parallel with the GUI thread instead of constantly badgering it with bits of work to do.

Aside from performance, ConfigureAwait has another important aspect: It can avoid deadlocks. Consider **Figure 3** again; if you add "ConfigureAwait(false)" to the line of code in DelayAsync, then the deadlock is avoided. This time, when the await completes, it attempts to execute the remainder of the async method within the thread pool context. The method is able to complete, which completes its returned task, and there's no deadlock. This technique is particularly useful if you need to gradually convert an application from synchronous to asynchronous.

If you can use ConfigureAwait at some point within a method, then I recommend you use it for every await in that method after that point. Recall that the context is captured only if an *incomplete* Task is awaited; if the Task is already complete, then the context isn't captured. Some tasks might complete faster than expected in different hardware and network situations, and you need to graciously handle a returned task that completes before it's awaited. **Figure 6** shows a modified example.

You should *not* use ConfigureAwait when you have code after the await in the method that needs the context. For GUI apps, this includes any code that manipulates GUI elements, writes data-bound properties or depends on a GUI-specific type such as Dispatcher/CoreDispatcher. For ASP.NET apps, this includes any code that uses HttpContext.Current or builds an ASP.NET response, including return statements in controller actions. **Figure 7** demonstrates one common pattern in GUI apps—having an async event handler disable its control at the beginning of the method, perform some awaits and then re-enable its

Figure 9 **Solutions to Common Async Problems**

Problem	Solution
Create a task to execute code	Task.Run or TaskFactory.StartNew (*not* the Task constructor or Task.Start)
Create a task wrapper for an operation or event	TaskFactory.FromAsync or TaskCompletionSource<T>
Support cancellation	CancellationTokenSource and CancellationToken
Report progress	IProgress<T> and Progress<T>
Handle streams of data	TPL Dataflow or Reactive Extensions
Synchronize access to a shared resource	SemaphoreSlim
Asynchronously initialize a resource	AsyncLazy<T>
Async-ready producer/consumer structures	TPL Dataflow or AsyncCollection<T>

control at the end of the handler; the event handler can't give up its context because it needs to re-enable its control.

Each async method has its own context, so if one async method calls another async method, their contexts are independent. **Figure 8** shows a minor modification of **Figure 7**.

Context-free code is more reusable. Try to create a barrier in your code between the context-sensitive code and context-free code, and minimize the context-sensitive code. In **Figure 8**, I recommend putting all the core logic of the event handler within a testable and context-free async Task method, leaving only the minimal code in the context-sensitive event handler. Even if you're writing an ASP.NET application, if you have a core library that's potentially shared with desktop applications, consider using ConfigureAwait in the library code.

To summarize this third guideline, you should use ConfigureAwait when possible. Context-free code has better performance for GUI applications and is a useful technique for avoiding deadlocks when working with a partially async codebase. The exceptions to this guideline are methods that require the context.

Know Your Tools

There's a lot to learn about async and await, and it's natural to get a little disoriented. **Figure 9** is a quick reference of solutions to common problems.

The first problem is task creation. Obviously, an async method can create a task, and that's the easiest option. If you need to run code on the thread pool, use Task.Run. If you want to create a task wrapper for an existing asynchronous operation or event, use TaskCompletionSource<T>. The next common problem is how to handle cancellation and progress reporting. The base class library (BCL) includes types specifically intended to solve these issues: CancellationTokenSource/CancellationToken and IProgress<T>/Progress<T>. Asynchronous code should use the Task-based Asynchronous Pattern, or TAP (msdn.microsoft.com/library/hh873175), which explains task creation, cancellation and progress reporting in detail.

Another problem that comes up is how to handle streams of asynchronous data. Tasks are great, but they can only return one object and only complete once. For asynchronous streams, you can use either TPL Dataflow or Reactive Extensions (Rx). TPL Dataflow creates a "mesh" that has an actor-like feel to it. Rx is more powerful and efficient but has a more difficult learning curve. Both TPL Dataflow and Rx have async-ready methods and work well with asynchronous code.

Figure 10 **SemaphoreSlim Permits Asynchronous Synchronization**

```
SemaphoreSlim mutex = new SemaphoreSlim(1);
int value;
Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()
{
  await mutex.WaitAsync().ConfigureAwait(false);
  try
  {
    value = await GetNextValueAsync(value);
  }
  finally
  {
    mutex.Release();
  }
}
```

Just because your code is asynchronous doesn't mean that it's safe. Shared resources still need to be protected, and this is complicated by the fact that you can't await from inside a lock. Here's an example of async code that can corrupt shared state if it executes twice, even if it always runs on the same thread:

```
int value;
Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()
{
  value = await GetNextValueAsync(value);
}
```

The problem is that the method reads the value and suspends itself at the await, and when the method resumes it assumes the value hasn't changed. To solve this problem, the SemaphoreSlim class was augmented with the async-ready WaitAsync overloads. **Figure 10** demonstrates SemaphoreSlim.WaitAsync.

Asynchronous code is often used to initialize a resource that's then cached and shared. There isn't a built-in type for this, but Stephen Toub developed an AsyncLazy<T> that acts like a merge of Task<T> and Lazy<T>. The original type is described on his blog (bit.ly/dEN178), and an updated version is available in my AsyncEx library (nitoasyncex.codeplex.com).

Finally, some async-ready data structures are sometimes needed. TPL Dataflow provides a BufferBlock<T> that acts like an async-ready producer/consumer queue. Alternatively, AsyncEx provides AsyncCollection<T>, which is an async version of BlockingCollection<T>.

I hope the guidelines and pointers in this article have been helpful. Async is a truly awesome language feature, and now is a great time to start using it! ■

Stephen Cleary *is a husband, father and programmer living in northern Michigan. He has worked with multithreading and asynchronous programming for 16 years and has used async support in the Microsoft .NET Framework since the first CTP. His home page, including his blog, is at stephencleary.com.*

Thanks to the following technical expert for reviewing this article: *Stephen Toub*

FOSE
EXPERIENCE TECHNOLOGY

MAY 14-16, 2013
WASHINGTON, DC
WALTER E. WASHINGTON CONVENTION CENTER

Keynotes

General Stan McChrystal

Joe Thiesmann

FREE EXPO*
Hands-on demos, free education, networking and more!

CYBERSECURITY
The Business of Cybersecurity
Detect, determine and develop your strategy.

Government Tech Talks
Valuable technology implementation—3 strategies in 15 minutes!

First @ FOSE
See industry giants' latest products and services.

RECORDS & INFORMATION MANAGEMENT
Comply & Implement
Take advantage of others' successful RIM programs.

CLOUD & VIRTUALIZATION
Benefit Today & Plan for Tomorrow
Do more with less. Obtain new guidance on cloud initiatives.

EXPERIENCE TECHNOLOGY

SEE IT. HEAR IT. TRY IT. KNOW IT.
Experience NEW Solutions at FOSE.

App Arcade
Experience the latest apps for government.

MOBILE GOVERNMENT
Digital Government Strategy
Achieve your BYOD, security, API and Enterprise System goals.

BIG DATA & BUSINESS INTELLIGENCE
Extract Meaning
Use your bits and bytes to reach objectives.

REGISTER TODAY!

Early Bird Special—SAVE $100!
USE PRIORITY CODE: FOSEAD2

FOSE.com

- 3-DAY PAID CONFERENCE delivering best practices & case studies from some of the biggest names in government.
- FREE EXPO* showcasing industry partners & their solutions to help you reach your mission goals!

*Expo is free for government; $50 for industry suppliers.

PLATINUM SPONSOR
Project Management Institute

TECHNOLOGY SPONSOR
BOXLIGHT Presenting a Better Way

GOLD SPONSORS
BIG BANG LLC
THE CENTECH GROUP, Inc. TRUST · QUALITY · SERVICE · VALUE
ClearanceJobs.com A Dice Service
Government Transaction Services, Inc.
IRONKEY by imation
mka mitchell kwon & associates llc
MOBILEGOV

SILVER SPONSOR
Schneider Electric

PRODUCED BY
1105 GOVERNMENT EVENTS

Migrating ASP.NET Web Forms to the MVC Pattern with the ASP.NET Web API

Peter Vogel

While ASP.NET MVC tends to get most of the attention these days, ASP.NET Web Forms and its related controls allow developers to generate powerful, interactive UIs in a short period of time—which is why there are so many ASP.NET Web Forms applications around. What ASP.NET Web Forms doesn't support is implementing the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) patterns, which can enable test-driven development (TDD).

The ASP.NET Web API ("Web API" hereafter) provides a way to build or refactor ASP.NET Web Forms applications to the MVC pattern by moving code from the codebehind file to a Web API controller. This process also enables ASP.NET applications to leverage Asynchronous JavaScript and XML (AJAX), which can be used to create a more responsive UI and improve an application's scalability by moving logic into the client and reducing communication with the server. This is possible because the Web API leverages the HTTP protocol and (through coding by convention) automatically takes care of several low-level tasks. The Web API paradigm for ASP.NET that this article proposes is to let ASP.NET generate the initial set of markup sent to the browser but handle all of the user's interactions through AJAX calls to a standalone, testable controller.

This article discusses:

- Adding Web API infrastructure
- Routing Web Forms
- Refactoring a Web Form
- Using AJAX
- Workflow processing
- Replacing events
- Handling transactions

Technologies discussed:

ASP.NET, AJAX, ASP.NET Web API

The Web API leverages the HTTP protocol and (through coding by convention) automatically takes care of several low-level tasks.

Setting up the infrastructure to have a Web Forms application interact with the server through a set of AJAX calls isn't difficult. But I won't mislead you: Refactoring the code in the Web Forms application code file to work in a Web API controller might not be a trivial task. You have to give up the various events fired by the controls, auto-generated server-side validation and the ViewState. However, as you'll see, there are some workarounds for living without these features that can reduce the pain.

Adding Web API Infrastructure

To use the Web API in an ASP.NET project, all you need to do (after adding the NuGet Microsoft ASP.NET Web API package) is right-click and select Add | New Item | Web API Controller Class. However, adding the controller this way creates a class with a lot of default code that you'll just have to delete later. You might prefer to simply add an ordinary class file and have it inherit from the System.Web.Http.ApiController class. To work with the ASP.NET routing infrastructure, your class name must end with the string "Controller."

> A Web API controller class supports a great deal of coding by convention.

This example creates a Web API controller called Customer:

```
public class CustomerController : ApiController
{
```

A Web API controller class supports a great deal of coding by convention. For example, to have a method called whenever a form is posted back to the server, you need only have a method named "Post" or with a name that begins with "Post" (under the hood, a page that's posted back to the server is sent to the server with the HTTP POST verb; the Web API picks methods based on the request's HTTP verb). If that method name violates your organization's coding convention, you can use the HttpPost attribute to flag the method to use when data is posted to the server. The following code creates a method called UpdateCustomer in the Customer controller to handle HTTP posts:

```
public class CustomerController : ApiController
{
  [HttpPost]
  public void UpdateCustomer()
  {
```

Post methods accept, at most, a single parameter (a post method with multiple parameters is ignored). The simplest data that can be sent to a post method is a single value in the body of the post, prefixed with an equal sign (for example, "=ALFKI"). The Web API will automatically map that data to the post method's single parameter, provided the parameter is decorated with the FromBody attribute, as in this example:

```
[HttpPost]
public HttpResponseMessage UpdateCustomer([FromBody] string CustID)
{
```

This is, of course, almost useless. If you want to post back more than a single value—the data from a Web Form, for example—you'll need to define a class to hold the values from the Web Form: a Data Transfer Object (DTO). The Web API coding convention standards help out here. You need only define a class with property names that match the names associated with the controls in the Web Form to have your DTO properties automatically populated with data from the Web Form by the Web API.

As an example of data that can be posted back to a Web API controller, the (admittedly simple) example Web Form shown in **Figure 1** has only three TextBoxes, a RequiredFieldValidator and a Button.

To have the post method accept the data from the TextBoxes in this Web Form, you'd create a class with properties with names that match the ID properties of the TextBoxes, as this class does (any controls in the Web Form that don't have a matching property are ignored by the Web API):

```
public class CustomerDTO
{
  public string CustomerID { get; set; }
  public string CompanyName { get; set; }
  public string City { get; set; }
}
```

A more complex Web Form might require a DTO that you can't live with (or is beyond the abilities of the Web API to bind to). If so, you can create your own Model Binder to map data from the Web Form controls to the DTO properties. In a refactoring scenario, the code in your Web Form will already be working with the names of the ASP.NET controls—having identically named properties on the DTO reduces the work required when you move that code into the Web API controller.

Routing the Web Form

The next step in integrating the Web API into an ASPX Web Form processing cycle is to provide a routing rule in the Application_Start event of the application's Global.asax file that will direct the form's postback to your controller. A routing rule consists of a template that specifies URLs to which the rule applies and which controller is to handle the request. The template also specifies where in the URL to find values that are to be used by the Web API (including values to be passed to methods in the controller).

There are some standard practices here that can be ignored. The standard routing rule can match almost any URL, which can lead to unexpected results when the rule is applied to URLs that you didn't intend the rule to be used with. To avoid that, a Microsoft best practice is to have URLs associated with the Web API begin with the string "api" to prevent collisions with URLs used elsewhere in the application. That "api" performs no other useful function and just pads out all of your URLs.

Putting that together, you end up with a generalized routing rule in the Application_Start event that looks like this (you need to add *using* statements for both System.Web.Routing and System.Web.Http to the Global.asax to support this code):

```
RouteTable.Routes.MapHttpRoute(
  "API Default",
  "api/{controller}/{id}",
  new { id = RouteParameter.Optional })
);
```

This routing extracts the controller name from the second parameter in the template, so URLs become tightly coupled to controllers. If you rename the controller, any clients using the URL stop working. (I also prefer that any parameters mapped in the URL by the template have more meaningful names than "id.") I've come to prefer more-specific routing rules that don't require the controller name in the template but, instead, specify the controller name in the defaults passed in the third parameter to the MapHttpRoute method. By making the templates in my routing rules more specific, I also bypass the need for a special prefix for URLs used with Web API controllers, and I'm less frequently surprised by the results of my routing rules.

My routing rules look like the following code, which creates a route called CustomerManagementPost that applies only to URLs beginning with "CustomerManagement" (following the server and site name):

```
RouteTable.Routes.MapHttpRoute(
  "CustomerManagementPost",
  "CustomerManagement",
  new { Controller = "Customer" },
  new { httpMethod = new HttpMethodConstraint("Post") }
);
```

This rule would, for example, apply only to a URL like www.phivs.com/CustomerManagement. In the defaults, I tie this URL to the Customer controller. Just to make sure the route is only used when I intend it, I use the fourth parameter to specify that this route is to be used only when data is being sent back as an HTTP POST.

> The first step in eliminating the request/response cycle is to insert some JavaScript into the process.

Refactoring to the Controller

If you're refactoring an existing Web Form, the next step is to get the Web Form to post its data to this newly defined route rather than back to itself. This is the first change to existing code—everything else done so far has been added code, leaving existing processing in place. The revised form tag should look something like this:

```
<form id="form1" runat="server" action="CustomerManagement"
   method="post" enctype="application/x-www-form-urlencoded">
```

The key change here is setting the form tag's action attribute to use the URL specified in the route ("CustomerManagement"). The method and enctype attributes help ensure cross-browser compatibility. When the page posts back to the controller, the Web API will automatically call the post method, instantiate the class being passed to the method and map data from the Web Form to the properties on the DTO—and then pass the DTO to the post method.

Figure 1 **A Basic Sample Web Form**

```
<form id="form1" runat="server">
<p>
  Company Id: <asp:TextBox ID="CustomerID"
    ClientIDMode="Static" runat="server">
    </asp:TextBox> <br/>
  Company Name: <asp:TextBox ID="CompanyName"
    ClientIDMode="Static" runat="server">
    </asp:TextBox>
  <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
    runat="server" ControlToValidate="CompanyName"
    Display="Dynamic"
    ErrorMessage="Company Name must be provided">
  </asp:RequiredFieldValidator><br/>
  City: <asp:TextBox ID="City"
    ClientIDMode="Static" runat="server">
    </asp:TextBox><br/>
</p>
<p>
  <asp:Button ID="PostButton" runat="server" Text="Update" />
</p>
</form>
```

With all the pieces in place, you can now write code in your controller's post method to work with the data in the DTO. The following code updates a matching Entity Framework entity object for a model based on the Northwind database using the data passed from the Web Form:

```
[HttpPost]
public void UpdateCustomer(CustomerDTO custDTO)
{
  Northwind ne = new Northwind();

  Customer cust = (from c in ne.Customers
                   where c.CustomerID == custDTO.CustomerID
                   select c).SingleOrDefault();
  if (cust != null)
  {
    cust.CompanyName = custDTO.CompanyName;
  }
  ne.SaveChanges();
```

When processing is complete, something should be sent back to the client. Initially, I'll just return an HttpResponseMessage object configured to redirect the user to another ASPX page in the site (a later refactoring will enhance this). First, I need to modify the post method to return an HttpResponseMessage:

```
[HttpPost]
public HttpResponseMessage UpdateCustomer(CustomerDTO custDTO)
```

Then I need to add the code to the end of the method that returns the redirect response to the client:

```
  HttpResponseMessage rsp = new HttpResponseMessage();
  rsp.StatusCode = HttpStatusCode.Redirect;
  rsp.Headers.Location = new Uri("RecordSaved.aspx", UriKind.Relative);
  return rsp;
}
```

The real work now begins, including:

- Moving whatever code was in the ASPX code file into the new controller method
- Adding in any server-side validation performed by the Validation controls
- Detaching the code from the events fired by the page

These aren't trivial tasks. However, as you'll see, you have some options that might simplify this process by continuing to AJAX-enable the page. One of those options, in fact, allows you to leave code in the Web Form if it can't be moved to the controller (or if it's to be moved later).

At this point in refactoring an existing Web Form, you've moved to the MVC pattern but you haven't moved to the AJAX paradigm. The page is still using the classic request/response cycle rather than eliminating the page's postback. The next step is to create a genuinely AJAX-enabled page.

Moving to AJAX

The first step in eliminating the request/response cycle is to insert some JavaScript into the process by setting the button's OnClientClick property to call a client-side function. This example has the button call a JavaScript function named UpdateCustomer:

```
<asp:Button ID="PostButton" runat="server" Text="Update"
  OnClientClick="return UpdateCustomer();" />
```

In this function, you'll intercept the postback triggered by the user clicking the button and replace it with an AJAX call to your service's method. Using the *return* keyword in OnClientClick and having UpdateCustomer return false will suppress the postback triggered by the button. Your intercept function should also invoke any client-side validation code generated by the ASP.NET Validation controls by

calling the ASP.NET-provided Page_ClientValidate function (in a refactoring process, calling the validators' client-side code might let you avoid having to recreate the validators' server-side validation).

If you're refactoring an existing Web Form, you can now remove the action attribute on the form tag that uses your route. Removing the action attribute allows you to implement a hybrid/staged approach to moving your Web Form's code to your Web API controller. For code that you don't want to move to your controller (yet), you can continue to let the Web Form post back to itself. For example, in your intercept function, you can check to see which changes have taken place in the Web Form and return true from the intercept function to let the postback continue. If there are multiple controls on the page that trigger postbacks, you can choose which controls you want to process in your Web API controller and write intercept functions just for those. This lets you implement a hybrid approach when refactoring (leaving some code in the Web Form) or a staged approach (migrating code over time).

The UpdateMethod now needs to call the Web API service to which the page was formerly posting back. Adding jQuery to the project and to the page (I've used jQuery 1.8.3) lets you use its post function to call your Web API service. The jQuery serialize function will convert the form into a set of name/value pairs that the Web API will map to the property names on the CustomerDTO object. Integrating this call into the UpdateCustomer function—so that the post only happens if no client-side errors are found—gives this code:

```
function UpdateCustomer() {
  if (Page_ClientValidate()){
    $.post('CustomerManagement', $('#form1').serialize())
    .success(function () {
      // Do something to tell the user that all went well.
    })
    .error(function (data, msg, detail) {
      alert(data + '\n' + msg + '\n' + detail)
    });
  }
  return false;
}
```

Serializing the form sends a lot of data to the controller, not all of which might be necessary (for example, the ViewState). I'll walk through sending just the necessary data later in this article.

The final step (at least for this simple example) is to rewrite the end of the post method in the controller so the user stays on the current page. This version of the post method just returns an HTTP OK status using the HttpResponseMessage class:

```
    ...
    ne.SaveChanges();

    HttpResponseMessage rsp = new HttpResponseMessage();
    rsp.StatusCode = HttpStatusCode.OK;
    return rsp;
}
```

Workflow Processing

You must now decide where the responsibility for any further processing should lie. As shown earlier, if the user is to be sent to a different page, you can handle that in your controller. However, if the controller is now just returning an OK message to the client, you might want to perform some additional processing in the client. For example, adding a label to the Web Form to display the result of the server-side processing would be a good start:

```
Update Status: <asp:Label ID="Messages" runat="server" Text=""></asp:Label>
```

In the success method for your AJAX call, you'd update the label with the status of your AJAX call:

```
success: function (data, status) {
  $("#Messages").text(status);
},
```

It's not unusual, as part of processing a posted page, for the Web Form's server-side code to update the controls on the page before returning the page to the user. To handle that, you'll need to return data from the service's post method and update the page from your JavaScript function.

The first step in that process is to set the Content property of the HttpResponseMessage object to hold the data that you're returning. Because the DTO created to pass data to the post method from the form is already available, using it to send data back to the client makes sense. However, there's no need to mark your DTO class with the Serializable attribute to use it with the Web API. (In fact, if you do mark the DTO with the Serializable attribute, the backing fields for the DTO properties will be serialized and sent to the client, giving you odd names to work with in your client-side version of the DTO.)

This code updates the DTO City property and moves it to the HttpResponseMessage Content property, formatted as a JSON object (you'll need to add a *using* statement for System.Net.Http.Headers to your controller to make this code work):

```
HttpResponseMessage rsp = new HttpResponseMessage();
rsp.StatusCode = HttpStatusCode.OK;
custDTO.City = cust.City;
rsp.Content = new ObjectContent<CustomerDTO>(custDTO,
              new JsonMediaTypeFormatter(),
              new MediaTypeWithQualityHeaderValue("application/json"));
```

The final step is to enhance the intercept function to have its success method move the data into the form:

```
.success(function (data, status) {
  $("#Messages").text(status);
  if (status == "success") {
    $("#City").val(data.City);
  }
})
```

This code doesn't, of course, update the ASP.NET ViewState. If the page does later post back in the normal ASP.NET fashion, then the City TextBox will fire a TextChanged event. If there's code in the Web Form's server-side code tied to that event, you might end up with unintended consequences. If you're either doing a staged migration or using a hybrid approach, you'll need to test for this. In a fully implemented version of the paradigm, where the Web Form isn't posted back to the server after the initial display, this isn't a problem.

Figure 2 **Using the jQuery AJAX Functionality to Issue a Controller Request**

```
function DeleteCustomer() {
  $.ajax({
    url: 'CustomerManagement/' + $("#CustomerID").val(),
    type: 'delete',
    success: function (data, status) {
      $("#Messages").text(status);
  },
  error: function (data, msg, detail) {
    alert(data + '\n' + msg + '\n' + detail)
    }
  });
  return false;
}
```

Replacing Events

As I noted earlier, you're going to have to live without the ASP.NET server-side events. However, you can instead capture the equivalent JavaScript event that triggers the postback to the server and invoke a method on your service that does what the code in the server-side event would've done. A staged refactoring process leverages this, letting you migrate these events when you have time (or feel the need).

> Most ASP.NET pages weren't designed with the HTTP verbs in mind.

For example, if the page has a delete button for deleting the currently displayed Customer, you can leave the functionality in the page's code file as part of your initial migration—just let the delete button post the page back to the server. When you're ready to migrate the delete function, begin by adding a function to intercept the delete button's client-side onclick event. In this example, I've chosen to wire up the event in JavaScript—a tactic that will work with any client-side event:

```
<asp:Button ID="DeleteButton" runat="server" Text="Delete"  />

<script type="text/javascript">
  $(function () {
    $("#DeleteButton").click(function () { return DeleteCustomer() });
  })
```

In the DeleteCustomer function, rather than serialize the whole page, I'll send only the data required by the server-side delete method: the CustomerID. Because I can embed that single parameter in the URL used to request the service, this lets me use another one of the standard HTTP verbs to select the correct controller method: DELETE (for more on HTTP verbs, see bit.ly/92iEnV).

> Refactoring the code file for a traditional ASPX page into a Web API controller isn't a trivial task.

Using the jQuery ajax function, I can issue a request to my controller, building the URL with data from the page and specifying that the HTTP delete verb is to be used as the type of request (see **Figure 2**).

The next step is to create a routing rule that will identify which part of the URL contains the CustomerID and assign that value to a parameter (in this case, a parameter named CustID):

```
RouteTable.Routes.MapHttpRoute(
  "CustomerManagementDelete",
  "CustomerManagement/{CustID}",
  new { Controller = "Customer" },
  new { httpMethod = new HttpMethodConstraint("Delete") }
);
```

As with the post, the Web API will automatically route an HTTP DELETE request to a method in the controller named or beginning with "Delete"—or to a method flagged with the HttpDelete attribute. And, as before, the Web API will automatically map any data extracted from the URL to parameters on the method that match the name in the template:

```
[HttpDelete]
public HttpResponseMessage FlagCustomerAsDeleted(string CustID)
{
  //... Code to update the Customer object ...
  HttpResponseMessage rsp = new HttpResponseMessage();
  rsp.StatusCode = HttpStatusCode.OK;
  return rsp;
}
```

Beyond the HTTP Verbs

Most ASP.NET pages weren't designed with the HTTP verbs in mind; instead, a "transactional" approach was often used in defining the original version of the code. This can make it difficult to tie the page's functionality into one of the HTTP verbs (or it can force you to create a complex post method that handles several different kinds of processing).

To handle any transaction-oriented functionality, you can add a route that specifies a method (called an "action" in routing-speak) on the controller by name rather than by HTTP type. The following example defines a URL that routes a request to a method called AssignCustomerToOrder and extracts a CustID and OrderID from the URL (unlike post methods, methods associated with other HTTP verbs can accept multiple parameters):

```
RouteTable.Routes.MapHttpRoute(
  "CustomerManagementAssign",
  "CustomerManagement/Assign/{CustID}/{OrderID}",
  new { Controller = "Customer", Action="AssignCustomerToOrder" },
  new { httpMethod = new HttpMethodConstraint("Get") }
  );
```

This declaration for the method picks up the parameters extracted from the URL:

```
[HttpGet]
public HttpResponseMessage AssignCustomerToOrder(
  string CustID, string OrderID)
{
```

The intercept function wired to the appropriate client-side event uses the jQuery get function to pass a URL with the correct components:

```
function AssignOrder() {
  $.get('CustomerManagement/Assign/' + $("#CustomerID").val() + "/" + "A123",
    function (data, status) {
      $("#Messages").text(status);
      });
  return false;
}
```

To recap, refactoring the code file for a traditional ASPX page into a Web API controller isn't a trivial task. However, the flexibility of the ASP.NET Web API, the power it provides for binding HTTP data to .NET objects, and the ability to leverage HTTP standards provide a potential way to move existing applications to an MVC/TDD model—and improve scalability by AJAX-enabling the page along the way. It also provides a paradigm for creating new ASP.NET applications that exploit both the productivity of Web Forms and the functionality of the ASP.NET Web API. ■

Peter Vogel *is a principal at PH&V Information Services, specializing in ASP.NET development with expertise in service-oriented architecture, XML, database and UI design.*

Thanks to the following technical experts for reviewing this article: *Christopher Bennage and Daniel Roth*

Extreme Performance & Linear Scalability

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

Enterprise Distributed Cache

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

ASP.NET Optimization in Web Farms

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

Runtime Data Sharing

- Powerful event notifications for pub/sub data sharing

Download a 60-day FREE trial today!

FREE EXPRESS EDITION

NCache™

Distributed Cache for .NET & Java

www.alachisoft.com

1-800-253-8195

Moving Your Applications to Windows Azure

Alex Homer

Lifestyle experts will tell you that moving to a new home is one of the most stressful events people undertake during their lifetime yet, given a choice between that or moving applications to a new platform, many of us would unhesitatingly start packing the china. Thankfully, however, moving your applications to Windows Azure is a breeze.

For many years, Microsoft has been building highly scalable applications in datacenters around the world—applications that have global reach and high availability, and offer great functionality to users. Windows Azure allows you to take advantage of the same infrastructure to deploy your own applications, with the corresponding capabilities to reduce your maintenance requirements, maximize performance and minimize costs.

Of course, people have been outsourcing their applications to third-party hosting companies for many years. This might be renting rack space or a server in a remote datacenter to install and run their applications, or it might just mean renting space on a Web server and database from a hosting company. In either case, however, the range of features available is usually limited. Typically, there's no authentication mechanism, message queuing, traffic management, data synchronization or other peripheral services that are a standard part of Windows Azure.

It might seem like all of these capabilities make moving applications to Windows Azure fairly complex, but as long as you take the time to consider your requirements and explore the available features, moving to Windows Azure can be a quick and relatively easy process. To help you understand the options and make the correct decisions, the patterns & practices group at Microsoft has recently published an updated version of the Windows Azure migration guide: "Moving Applications to the Cloud on Windows Azure" (msdn.microsoft.com/library/ff728592).

The guide covers a wide range of scenarios for migrating applications to Windows Azure. In the remainder of this article I'll explore these scenarios, look at the decisions you'll need to make, and see how the guide provides practical and useful advice to help you make the appropriate choices. While the guide follows a

This article discusses:

- Infrastructure versus platform hosting
- Storing your data
- Worker roles and background tasks
- Determining the cost of deploying to Windows Azure

Technologies discussed:

Windows Azure, SQL Server, MySQL

somewhat-contrived multi-step migration process—which most people are unlikely to follow in its entirety—this approach demonstrates most of the options and shows the capabilities of Windows Azure that might be useful in your own applications. **Figure 1**, taken from the guide, shows a conceptual map of the migration paths you might follow when moving an application from an on-premises datacenter to Windows Azure.

Infrastructure or Platform Hosting?

As you can see from the map, the initial decision when migrating to Windows Azure is to choose which route to follow—Infrastructure as a Service (IaaS) or Platform as a Service (PaaS).

As its name suggests, the IaaS approach provides the runtime infrastructure—such as a virtual server and network connectivity—for you to install the OS, services and applications of your choice. Effectively, you just pick up your server and run it in Microsoft datacenters. Windows Azure offers a range of preinstalled OSes you can choose from (such as Windows Server and Linux), and you can still take advantage of peripheral services, such as authentication though Windows Azure Active Directory, messaging with Windows Azure storage queues or Service Bus, global routing to your application through Traffic Manager, and more.

Alternatively, you can have Microsoft manage the OS and runtime platform for you by adopting the PaaS approach. Windows Azure Web Sites provides an easy-to-use hosting platform for Web applications and Web sites with simple management and deployment capabilities that can integrate directly with many source control systems, and supports a range of programming languages. If you want more control over the platform, including the capability to run a mixture of different types of roles and directly integrate caching, you can choose the Cloud Services approach. This allows you to deploy separate Web and worker roles; provides a wider range of configuration options; and is well-suited to a versioned and staged deployment environment.

As you'll see later in this article, the choice between IaaS and PaaS isn't limited to application code. When you follow the IaaS route, you can choose to deploy a Windows Azure Virtual Machine (VM) that has a database such as SQL Server or MySQL preinstalled. Meanwhile, on the PaaS route, Windows Azure also offers a hosted database mechanism called Windows Azure SQL Database that's effectively a hosted SQL Server instance you can use in almost exactly the same way as an on-premises SQL Server.

Getting to the Cloud with IaaS

There are several distinct advantages to using the IaaS approach for hosting your applications. For example, you might be able to move to a VM in the cloud without needing to make any changes to your application code, and connect it to your internal network and domain through a virtual network, and all of your testing, deployment, management and monitoring systems will continue to work much as they did before.

Effectively, all you've done is replace the Ethernet cable in your datacenter with an Internet connection to Windows Azure. A Windows Azure virtual network can connect your on-premises network to a network you configure in the cloud using a standard virtual private network (VPN) router, allowing you to use internal network IP addresses just as you would on-premises. Yes, there is some additional latency over the connection, and you'll need to consider how to handle occasional transient connection failures, but you are freed from the need to manage and update the hardware, infrastructure and OS. (To handle transient connection failures for many different kinds of operations, consider using the Microsoft Transient Fault Handling Application Block. For more details, see msdn.microsoft.com/library/hh680934(PandP.50).)

The IaaS approach using Windows Azure VMs is ideal when you need to run software or services where you don't have access to the source code or you can't modify the application (such as when you depend on a third-party application). It also works well if you need a non-standard configuration of the OS or the usual services, or if you want to be able to set specific permissions on files and resources.

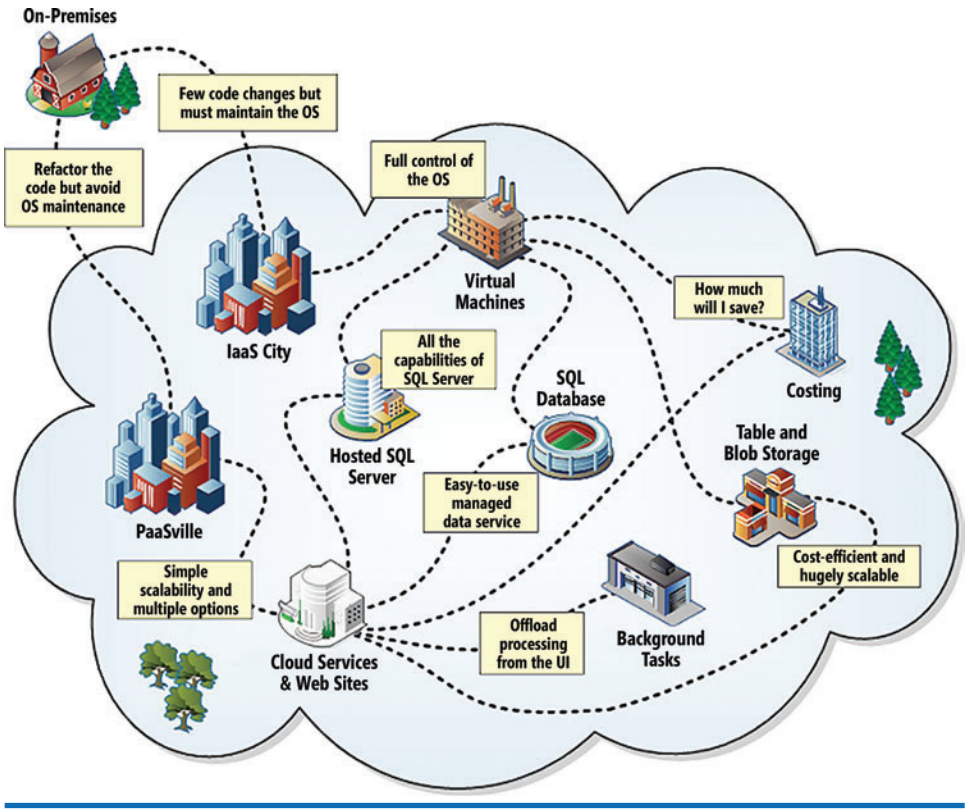


Figure 1 **A Conceptual Map of Some of the Possible Migration Paths in Windows Azure**

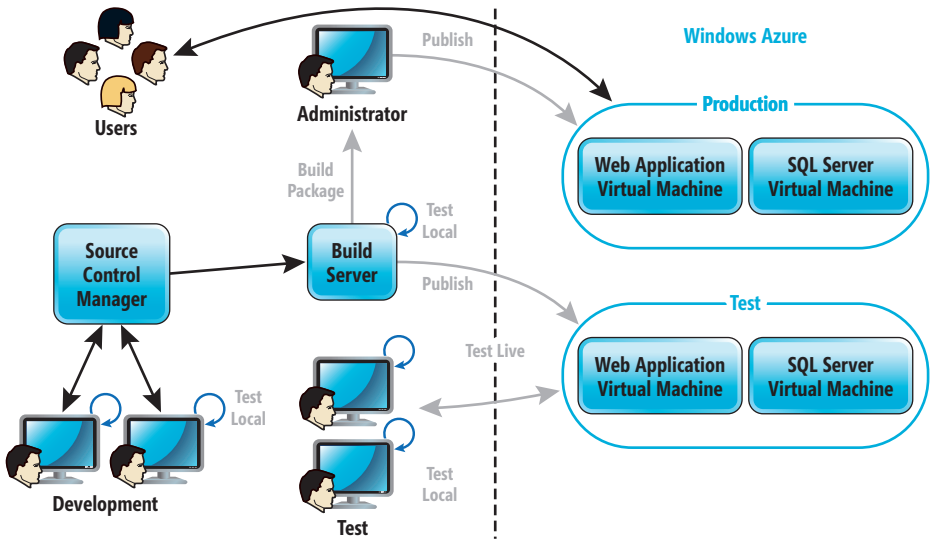


Figure 2 **Overview of a Possible Development, Test and Deployment Mechanism**

In terms of testing and deployment, your development teams will see no difference from existing processes. The on-premises development computers and the build server can deploy to the test and production environments in Windows Azure, or you can locate the test and build servers in the cloud. **Figure 2**, based on a figure from the guide, shows an example of a test and deployment configuration that encompasses both on-premises and cloud-hosted testing environments by using two separate Windows Azure subscriptions—one for testing and one for the live application.

So the only real difference when choosing the Windows Azure IaaS approach is that the application is no longer running in your own expensive, air-conditioned server room, consuming resources and demanding bandwidth from your Internet connection. Instead, it's running in a Microsoft datacenter of your choice where changes to the VM are persisted in the backup storage of the original image, reliable connectivity is provided at all times and the runtime platform will ensure that it's continuously available.

In addition, you can choose from a range of sizes for your VM; update the running instances when required; configure the OS and its services to suit the application's specific demands; deploy additional instances to meet changes in load; and even set up automatic routing to deployments in different datacenters to maximize availability and minimize response times for users around the world.

Simplifying Management with PaaS

If you want to avoid managing the OS yourself, you might choose the PaaS approach. While this does mean that you give up some opportunities to configure your runtime platform, it reduces administrative tasks and management costs because Microsoft is responsible for maintaining the servers, updating the OS and applying patches. You simply concentrate on the application code and its interaction with peripheral services.

The easiest way to move a Web site or Web application to Windows Azure is to deploy it to Windows Azure Web Sites; very few, if any, changes are required to the application. You can deploy from Microsoft Team Foundation Server (TFS) or other source code repository systems such as GitHub. Depending on your needs and your hosting budget, you can choose to host on a shared Web server or on a reserved instance where you can guarantee the performance and manage the number of instances to meet demand.

Alternatively, if you need a built-in mechanism for versioning deployments and staging applications, as well as the freedom to scale parts of the application separately, you may decide to use Windows Azure Cloud Services Web and worker roles to host your application. By moving the background tasks to worker roles and placing the UI in Web roles, you can balance the load on the application, perform asynchronous background processing, and scale each type of role separately by running the appropriate number of instances of each one.

(To implement autoscaling for roles in a Cloud Services deployment on a predefined schedule, or in response to runtime events such as changes in server load, consider using the Microsoft Autoscaling Application Block. For more details, see msdn.microsoft.com/library/hh680892(PandP.50).)

To connect Web and worker roles, you typically pass data between them as messages using Windows Azure storage queues or Windows Azure Service Bus queues. (Service Bus queues support a larger message size and have built-in facilities for authentication and access control.) Using messaging also opens up the design to allow the use of standard messaging and storage patterns such as Request/Response, Fire and Forget, Delayed Write, and more. If your application is built as components following a service-oriented architecture (SOA) design, moving it to Windows Azure Cloud Services will be relatively easy.

Of course, using Web and worker roles can mean that some refactoring of the application is required. However, in many cases this isn't onerous and doesn't affect the core business logic or presentation code. For example, ASP.NET

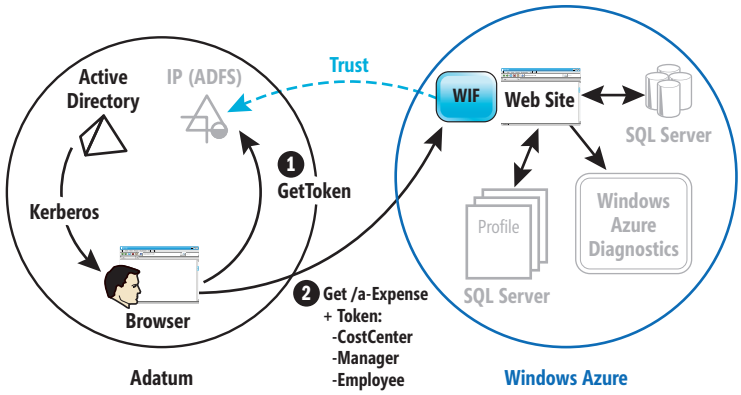


Figure 3 **Adopting a Claims-Based Authentication System**

MVC applications work fine when migrated to Windows Azure, and they can access data stores such as SQL Server in exactly the same way as when running on-premises or when deployed to VMs using the IaaS approach.

Moving to Windows Azure Cloud Services can also present an opportunity to update your authentication and authorization mechanism, especially if you find you need to perform some refactoring of the code. Modern applications increasingly use a claims-based authentication mechanism, including federated identity and single sign-on (SSO).

This type of mechanism allows users to sign on using a range of existing credentials rather than requiring specific credentials just for your application, and to sign on only once when accessing more than one application or Web site. The access-control feature of Windows Azure Active Directory, along with Windows Identity Framework (WIF), makes implementing claims-based authentication and federated identity easy. **Figure 3**, based on a figure from the guide, shows an example for the fictional company Adatum's a-Expense application, where users are authenticated by their own Active Directory and are issued a token that they present to the application in order to gain access.

For more information about claims-based authentication, check out the related patterns & practices publication, "A Guide to Claims-Based Identity and Access Control," at msdn.microsoft.com/library/ff423674.

Where Will My Data Live?

Almost all business applications use data, and often this is stored in a database such as SQL Server or a relational database management system (RDBMS) from another supplier. A typical scenario when migrating an application that uses a relational database is to deploy a Windows Azure VM that hosts the database server (Windows Azure provides preconfigured VMs containing either SQL Server or MySQL). Alternatively, you can install almost any other type of data store that runs on Windows or Linux in a VM running in Windows Azure.

You connect to a cloud-hosted database from your cloud-hosted application just as you would if the database and application were located in your own datacenter. However, this is only one of the options available in Windows Azure. Typically you'll need to decide whether you deploy the data for your applications in the cloud, or keep it on-premises and communicate with the database server over a virtual network or through messaging. If you choose the cloud, you must also decide whether to follow the IaaS or PaaS (SQL Database) path.

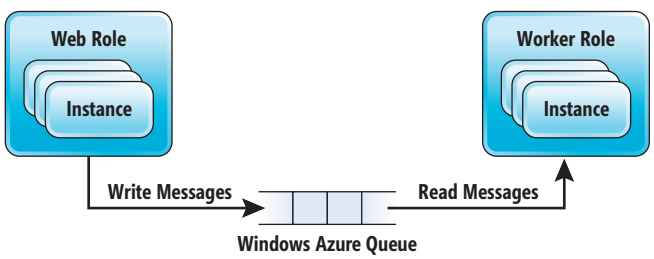


Figure 4 **Passing Messages from a Web Role to a Worker Role**

The Command Query Responsibility Segregation (CQRS) pattern typically uses messaging to communicate data updates and query results between application components, and Windows Azure provides two queuing mechanisms your application can use for this. However, for standard data access, introducing a network such as the Internet between the application and the database can result in delays and subsequent poor performance. If circumstances or regulatory requirements demand an on-premises database, Windows Azure Caching can help to reduce these delays. (For more information about the CQRS pattern, check out the related patterns & practices guide, "CQRS Journey," at msdn.microsoft.com/library/jj554200.)

Windows Azure SQL Database is an ideal solution for general data storage when migrating an existing application that uses SQL Server. Unless the code requires some of the more esoteric features of SQL Server—such as free text search, XML handling capabilities, procedures that require CLR programmability, or distributed queries that rely on the SQL Server Service Broker—the existing application will work with no changes.

Windows Azure SQL Database is also very cost-effective when you need to store only the usual volumes of data. However, for very large volumes of data or when you need to deploy many databases, using a database server hosted in a VM becomes an attractive solution. You can choose the appropriate size of VM, and even use multiple VMs to implement database failover or a shared data store.

Sometimes data storage and querying needs go beyond the capabilities of relational database systems. For example, corporations often have petabytes of data in Web server logs, financial transaction files, social media information, medical data, or other types of data that aren't regularly processed but might be of use for occasional or future querying. Windows Azure offers the HDInsight service (based on open source Hadoop technologies) for just this scenario. For more information, see hadooponazure.com.

Tables, Blobs, Queues and Drives

Windows Azure also offers a different type of storage that's not based on the relational SQL paradigm. You can use Windows Azure storage tables and blobs to store application data, storage queues to pass messages between application components and storage drives that act rather like a traditional disk-based filing system.

Windows Azure storage tables are schema-less, meaning each row is an entity containing a property bag of values, and a table can have different types of entities in each row. This is ideal for structured (columns and rows) and semi-structured data. Windows Azure storage blobs, on the other hand, are better suited to storing unstructured data such as documents, binary data, XML files and images.

Windows Azure storage is also very cheap compared to using a VM-hosted database server or SQL Database, but it does mean that existing data-access code must be rewritten. Typically, Windows Azure tables and blobs are more useful when you design your applications from scratch to run in Windows Azure. However, if your application has a clearly delineated data-access layer that you can replace with one designed to use Windows Azure tables and blobs, it's reasonably easy to change over to using Windows Azure storage instead of a relational database.

As with all cloud-based services, there's a chance that transient network issues or intermittent and temporary throttling of resources could cause an initial connection request to fail. The Transient Fault Handling Application Block (mentioned earlier) can be used to automatically detect failures and retry all storage operations, including relational databases and Windows Azure storage. This is a good practice for all cloud-based applications, and the Transient Fault Handling Application Block makes it easy to implement.

Worker Roles and Background Tasks

One of the strengths of Windows Azure Cloud Services is the provision of a special type of role designed to perform background processing. The worker role isn't limited to use only as an ancillary part of an application; in reality it's a Web role without the Web server (IIS) installed. However, the typical scenario is to run continuous tasks in the worker role that listen on Windows Azure queues (or Service Bus queues) for messages that instruct the role to perform some action in the background, as illustrated in **Figure 4**.

Separating out asynchronous and background tasks in this way helps you to implement many basic application design principles such as separation of concerns and single responsibility. By passing information and commands as messages between roles, you aid encapsulation of each role and reduce dependencies, in much the same way as when applying SOA principles.

For example, the guide explores how Adatum uses worker roles to compress and store images of receipts uploaded by users so as to conserve storage space, and to export the expenses data to the Adatum payroll application. Both tasks are initiated by a message placed in a queue. The message contains data relevant to the individual task.

As you refactor your application for deployment to Windows Azure Cloud Services roles, tasks that are suitable for running in a worker role will usually be obvious. Background tasks may also be initiated on a schedule, and you can minimize hosting costs by running more than one task in a worker role as long as you include code to restart any that might fail or stall. Windows Azure does detect when a role fails and will attempt to restart it, but it can't detect when an individual task in a role fails.

You must also consider how you will handle any feedback or output from the background task. For example, you might decide to use the delayed write pattern to store details of the orders that users submit by packaging each one in a message that the UI sends to a worker role task. The worker role will read the message from the queue and store the details in the database. However, if the order number is generated only at the point of storage, it will be more complex for the worker role to return this order number to the Web role for display in the UI.

Can I Afford Windows Azure?

As you've seen, Windows Azure contains a broad range of features and services that should fulfill all your applications' requirements, even if you aren't exactly sure what all of them are at the moment. You can deploy to any of the datacenters located around the world and take advantage of huge storage and scalability capabilities. But can you actually afford to have your applications live in Windows Azure?

In Chapter 6, "Evaluating Cloud Hosting Costs," of the guide, Adatum carries out several costing exercises to discover approximately how much it will be billed for running its a-Expense application in Windows Azure. The application will use a single Web role and a single worker role, and will need to store about 20GB of data in a relational database and 120GB of images in Windows Azure storage. At current prices, Adatum expects this to cost around $3,000 per year. Although Adatum can't accurately identify the cost of running this application in its own datacenter, because many of the resources it uses are shared with other on-premises applications, the expected cost on Windows Azure compares very favorably.

Even better, by introducing an autoscaling solution such as the Enterprise Library Autoscaling Application Block, Adatum calculates that it can run the application only during office hours, but double the capacity by adding extra role instances at the end of each month when most expenses are filed, and reduce the overall cost to something around $2,000 per year. Moreover, by adapting the application to use Windows Azure tables and blobs instead of SQL Database or SQL Server, Adatum calculates that it could save an additional $750 per year.

Of course, your own application capacity requirements and operating schedules will be different, and will depend on the load the applications must handle and the storage they require. However, it seems clear that you can afford to live in Windows Azure, and that moving in will be a great deal less stressful than moving your family to a new home! ■

More Information

The patterns & practices guide, "Moving Applications to the Cloud on Windows Azure," is available at **msdn.microsoft.com/library/ff728592**. You can download a PDF copy of the guide from **microsoft.com/download/details.aspx?id=29252**.

The associated guides and frameworks from patterns & practices are:

- Developing Multi-tenant Applications for the Cloud: **msdn.microsoft.com/library/ff966499**
- Building Hybrid Applications in the Cloud on Windows Azure: **msdn.microsoft.com/library/hh871440**
- A Guide to Claims-Based Identity and Access Control: **msdn.microsoft.com/library/ff423674**
- The Transient Fault Handling Application Block: **msdn.microsoft.com/library/hh680934(PandP.50)**
- The Autoscaling Application Block: **msdn.microsoft.com/library/hh680892(PandP.50)**

You can find the Windows Azure homepage at **windowsazure.com**, and the Windows Azure Developer Center is at **windowsazure.com/en-us/develop/overview**.

Alex Homer *is a technical writer assigned to the Microsoft patterns & practices division in Redmond, Wash. However, he has so far resisted the dubious attractions of Seattle weather in favor of working from home in the idyllic rural surroundings of the Derbyshire Dales in England. His semi-coherent ramblings on the IT industry and life in general can be found at blogs.msdn.com/alexhomer.*

Thanks to the following technical expert for reviewing this article: *Masashi Narumoto*

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...

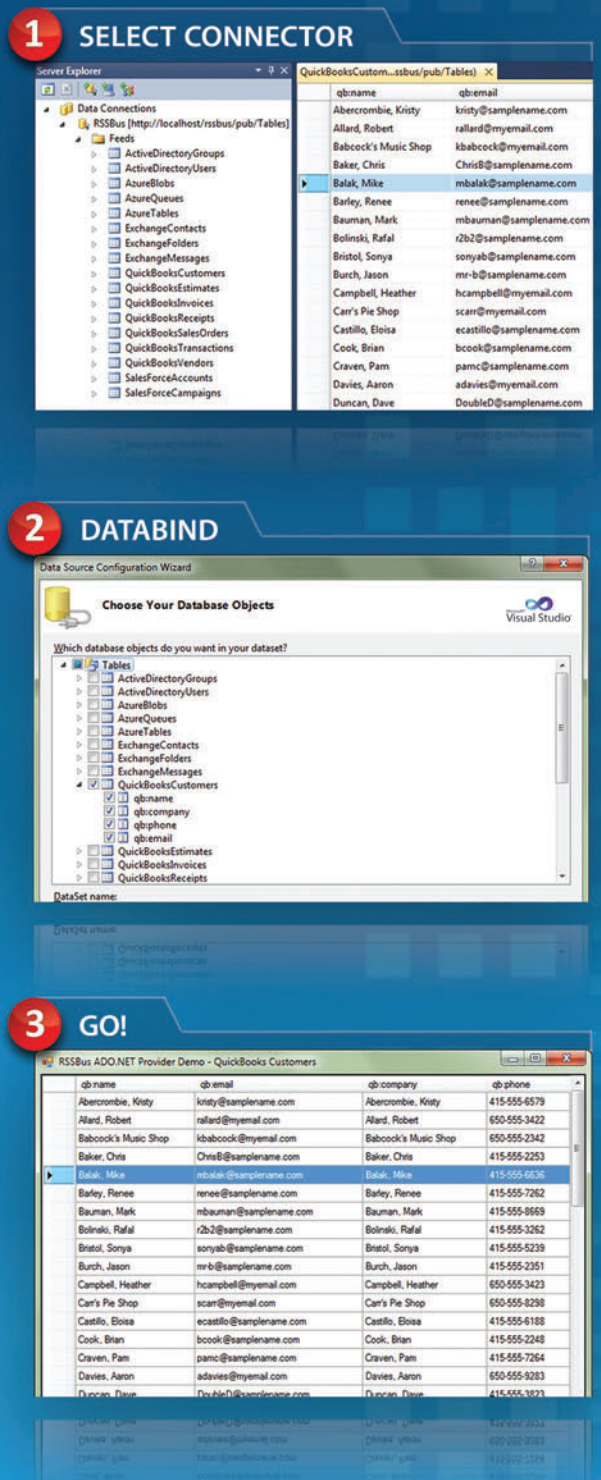
The RSSBus Data Providers give your .NET applications the power to databind (just like SQL) to Amazon, PayPal, eBay, QuickBooks, FedEx, Salesforce, MS-CRM, Twitter, SharePoint, Windows Azure, and much more! Leverage your existing knowledge to deliver cutting-edge WinForms, ASP.NET, and Windows Mobile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

Data Clustering Using Naive Bayes Inference

James McCaffrey

Data clustering is a machine-learning technique that has many important practical applications, such as grouping sales data to reveal consumer-buying behavior, or grouping network data to give insights into communication patterns. Data clustering is also useful for identifying anomalous data points. In this article I present a complete C# program you can use as the basis for adding clustering features to a software system or for creating a powerful standalone clustering utility.

There are many different clustering algorithms, in part because the effectiveness of a clustering algorithm depends to some extent on the characteristics of the data being clustered. The most common algorithm is called k-means clustering. Unfortunately, this algorithm is applicable only for numeric data items. In contrast, the clustering algorithm I'll present in this article is based on a technique called Naive Bayes inference, which works with either categorical or numeric data. Although all the ideas used in the clustering algorithm presented here are well-known, the overall algorithm and specific implementation have not, to the best of my knowledge, been published before. I call this algorithm and its implementation Iterative Naive Bayesian Inference Agglomerative Clustering (INBIAC) to distinguish it from other clustering techniques. Naive Bayes inference is a very common technique for performing data classification, but it's not generally known that Naive Bayes can also be used for data clustering.

This article discusses:

- Data clustering algorithms
- A demo program based on Naive Bayes inference
- Understanding Naive Bayes inference
- The INBIAC algorithm

Technologies discussed:

Visual Studio 2010, C#

Code download available at:

archive.msdn.microsoft.com/mag201303INBIAC

The best way to understand where I'm headed in this article is to take a look at **Figure 1**. The demo program begins by generating eight random data tuples that describe the location (urban, suburban or rural), income (low, medium, high or very high) and politics (liberal or conservative) of eight hypothetical people. The program then loads the raw string data into memory, and converts the data into int values to enable efficient processing. For example, the last tuple of ("Rural," "Low," "Conservative") is stored as (2, 0, 1).

Many clustering algorithms, including INBIAC, require the number of clusters to be specified. Here, variable numClusters is set to 3. The demo program clusters the data and then displays the final clustering of [2, 0, 2, 1, 1, 2, 1, 0]. Behind the scenes, the algorithm seeds clusters 0, 1 and 2 with tuples 1, 6 and 0, respectively. Then each of the remaining five tuples is assigned, one at a time, to a cluster. This type of algorithm is called agglomerative.

Because no training data or user intervention is required, clustering is sometimes termed "unsupervised learning." Each index in

the clustering array represents a tuple and the value in the array is a cluster ID. So tuple 2 ("Suburban," "VeryHigh," "Conservative") is assigned to cluster 0, tuple 1 ("Rural," "Medium," "Conservative") is assigned to cluster 2 and so on.

> The effectiveness of a clustering algorithm depends to some extent on the characteristics of the data being clustered.

The demo program finishes up by displaying the original raw data in clustered form. As you can see, the final clustering seems reasonable. And if you look at the original data, I think you'll agree that trying to cluster data manually, even for very small data sets, would be extremely difficult.

This article assumes you have advanced programming skills with a C-family language, but does not assume you have experience with Naive Bayes inference or clustering algorithms. The demo program shown in **Figure 1** is a single C# console application. I coded it without using OOP techniques so you can more easily refactor to languages that don't fully support OOP.

I removed all error checking to keep the ideas as clear as possible. The code for the demo program is too long to present in its entirety in this article, so I'll focus on explaining the key parts of the algorithm. The complete source code for the demo program is available at archive.msdn.microsoft.com/mag201303INBIAC.

Program Structure

The overall program structure, with some comments and WriteLine statements removed, is listed in **Figure 2**.

I used Visual Studio 2010 to create a C# console app named ClusteringBayesian. In the Solution Explorer window I renamed file Program.cs to the more descriptive ClusteringBayesianProgram.cs, which automatically renamed the single class. I removed unneeded template-generated references to the .NET namespaces; notice the program has few dependencies and needs only the Systems.Collections.Generic namespace.

I declare a class-scope Random object named random. This object is used to generate semi-random raw data, and to determine which tuples to use to seed each cluster. In the Main method, I instantiate random using a seed value of 6, only because this generates a nice-looking demo.

The next few lines of the demo program use helper methods MakeData and ShowData to generate and display eight lines of dummy data. MakeData calls sub-helper methods MakeParty, MakeLocation, MakeIncome and MakePolitics. If you want to experiment with specific hardcoded data tuples, you can replace this code with, for example:

```
int numTuples = 4;
string[] rawData = new string[] { "Urban,VeryHigh,Liberal",
  "Rural,Medium,Conservative", "Urban,Low,Liberal",
  "Suburban,Medium,Liberal" };
```

Next, the demo program hardcodes the attribute names Location, Income and Politics. In some scenarios, for example if your raw data is in a text file with a header or in a SQL table with column names, you may want to scan your data and programmatically determine the attribute names.

The demo program also hardcodes the attribute values Urban, Suburban and Rural for Location; Low, Medium, High and VeryHigh for Income; and Liberal and Conservative for Politics. Again, in some scenarios you may want to scan your raw data to programmatically determine the attribute values.

The demo program calls helper method LoadData to load the raw string data into an array of arrays in memory. For very large data sets, this might not be possible. In such situations you'll

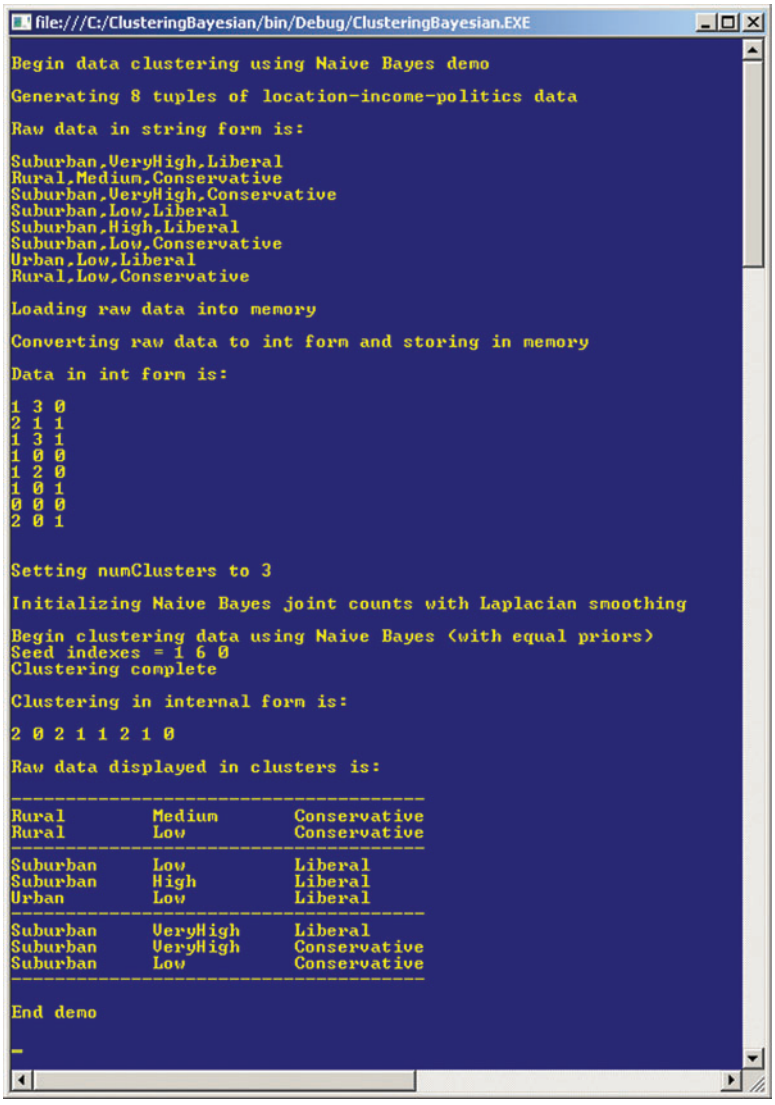


Figure 1 **Data Clustering Using Naive Bayes Inference**

need to either load blocks of data at a time or iterate through the externally stored raw data one line or row at a time.

Although it's possible to work with raw string data directly, it's much more efficient to work with the data encoded as type int. So, next, Helper method TuplesToInts accepts the array of arrays of strings, scans through that array, converts each string to a zero-based index, and stores all the data in an array of arrays of type int. Again, for very large data sets you may need to read raw data a line at a time and convert attribute values to ints on the fly.

The demo program prepares the clustering routine by setting two parameter values. Parameter numClusters is the number of clusters to generate. In general, data clustering is an exploratory process and you must experiment with different values of numClusters (although there are some fascinating techniques for programmatically determining an optimal number of clusters). Parameter numTrials is used by the clustering routine when generating the initial clustering, as I'll explain shortly.

> Because no training data or user intervention is required, clustering is sometimes termed "unsupervised learning."

The clustering method requires two arrays to hold the counts of assigned tuples at any given time in the INBIAC algorithm. Array jointCounts holds the number of clustered tuples that have a particular attribute value and a particular cluster. The jointCounts array is a bit tricky, and I'll explain it in more detail shortly. But each value in jointCounts is initialized to 1 as part of an important step in the Bayesian technique called Laplacian smoothing. The second array, clusterCounts, holds the number of tuples assigned to each cluster at any given time. The index of clusterCounts represents a cluster, and the value is the associated count.

The clustering is performed by method Cluster. Method Cluster returns an array that encodes which cluster is assigned to each tuple. The demo program concludes by displaying the clustering array, and by displaying the raw data grouped by cluster using helper method DisplayClustered.

Naive Bayes Inference

The key to understanding the INBIAC algorithm so that you'll be able to modify the demo code to meet your own needs is understanding Naive Bayes inference. Naive Bayes is best explained by example. Suppose you have the eight tuples shown in **Figure 1** and you want to place each tuple into one of three clusters:

```
[0] Suburban VeryHigh  Liberal
[1] Rural    Medium    Conservative
[2] Suburban VeryHigh  Conservative
[3] Suburban Low       Liberal
[4] Suburban High      Liberal
[5] Suburban Low       Conservative
[6] Urban    Low       Liberal
[7] Rural    Low       Conservative
```

First, assume that each cluster receives a single seed tuple (in a way that will be explained later) so that tuple 1 is assigned to cluster 0, tuple 6 is assigned to cluster 1, and tuple 0 is assigned to cluster 2. There are several ways to represent this clustering, but the INBIAC algorithm would store the information as an array:

```
[  2,  0, -1, -1, -1, -1,  1, -1 ]
```

In this array, the array indexes are tuples, the array values are clusters, and a value of -1 indicates the associated tuple has not yet been assigned to a cluster. So, conceptually, the clustering at this point is:

```
c0 : Rural    Medium   Conservative  (tuple 1)
c1 : Urban    Low      Liberal       (tuple 6)
c2 : Suburban VeryHigh Liberal       (tuple 0)
```

Now the goal is to assign the first unassigned tuple, tuple 2 (Suburban, VeryHigh, Conservative), to one of the three clusters. Naive Bayes computes the probabilities that tuple 2 belongs to clusters 0, 1 and 2, and then assigns the tuple to the cluster that has the greatest probability. Expressed symbolically, if X = (Suburban, VeryHigh, Conservative) and c0 stands for cluster 0, we want:

```
P(c0 | X)
P(c1 | X)
P(c2 | X)
```

The first probability can be read as, "the probability of cluster 0 given the X values." Now, bear with me for a moment. To compute these conditional probabilities, you have to compute terms I call partial probabilities (PP). After the partials for each of the conditionals are computed, the probability for each cluster is equal to the partial for the cluster divided by the sum of all the partials. Symbolically:

```
P(c0 | X) = PP(c0 | X) / [PP(c0 | X) + PP(c1 | X) + PP(c2 | X)]
P(c1 | X) = PP(c1 | X) / [PP(c0 | X) + PP(c1 | X) + PP(c2 | X)]
P(c2 | X) = PP(c2 | X) / [PP(c0 | X) + PP(c1 | X) + PP(c2 | X)]
```

The partial probability for P(c0 | X) is:

```
PP(c0 | X) = P(Suburban | c0) *
             P(VeryHigh | c0) *
             P(Conservative | c0) *
             P(c0)

           = count(Suburban     & c0) / count c0 *
             count(VeryHigh     & c0) / count c0 *
             count(Conservative & c0) / count c0 *
             P(c0)
```

These equations come from Bayes theory and aren't at all obvious. The term "naive" in Naive Bayes means that each of the probability terms in the partial probabilities are assumed to be mathematically independent, which leads to much simpler calculations than if the probability terms were mathematically dependent.

The last term, P(c0), is the "probability of cluster 0." This term is sometimes called a prior and can be computed in one of two ways. One way is to assume equal priors, in which case the P(c0) = P(c1) = P(c2) = 1/3. The other way is to not assume equal priors and use the current counts of assigned tuples, in which case P(c0) = count(c0) / (count(c0) + count(c1) + count(c2)). For the INBIAC algorithm, it's preferable to assume equal priors.

Notice that the equation needs what are called joint counts. For example, count(Suburban & c0) is the count of assigned tuples, where the cluster is c0 and the tuple location is Suburban.

If you look back at the current clustering, you'll see there's a problem: at this point, with only the first three seed tuples assigned to clusters, the count(Suburban & c0) is 0, which makes its term 0, which zeros out the entire partial probability. To avoid this, the joint counts are all initialized with a value of 1. This is called

Figure 2 **Clustering Program Structure**

```
using System;
using System.Collections.Generic;

namespace ClusteringBayesian
{
  class ClusteringBayesianProgram
  {
    static Random random = null;

    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin data clustering using Naive Bayes demo\n");
        random = new Random(6); // Seed of 6 gives a nice demo

        int numTuples = 8;
        Console.WriteLine("Generating " + numTuples +
          "tuples of location-income-politics data");
        string[] rawData = MakeData(numTuples);

        Console.WriteLine("\nRaw data in string form is:\n");
        ShowData(rawData, numTuples);

        string[] attNames = new string[] { "Location", "Income", "Politics" };

        string[][] attValues = new string[attNames.Length][];
        attValues[0] = new string[] { "Urban", "Suburban", "Rural" };
        // Location
        attValues[1] = new string[] { "Low", "Medium", "High", "VeryHigh" };
        // Income
        attValues[2] = new string[] { "Liberal", "Conservative" };
        // Politics

        Console.WriteLine("Loading raw data into memory\n");
        string[][] tuples = LoadData(rawData, attValues);

        Console.WriteLine("Converting raw data to int form and" +
          "storing in memory\n");
        int[][] tuplesAsInt = TuplesToInts(tuples, attValues);

        Console.WriteLine("Data in int form is:\n");
        ShowData(tuplesAsInt, numTuples);

        int numClusters = 3;
        int numTrials = 10;  // Times to get good seed indexes (different tuples)
        Console.WriteLine("\nSetting numClusters to " + numClusters);

        Console.WriteLine("\nInitializing Naive Bayes joint counts with " +
          "Laplacian smoothing");
        int[][][] jointCounts = InitJointCounts(tuplesAsInt, attValues,
          numClusters);
        int[] clusterCounts = new int[numClusters];

        Console.WriteLine("\nBegin clustering data using Naive Bayes " +
          "(with equal priors) ");
        int[] clustering = Cluster(tuplesAsInt, numClusters, numTrials,
          jointCounts, clusterCounts, true);
        Console.WriteLine("Clustering complete");

        Console.WriteLine("\nClustering in internal form is:\n");
        for (int i = 0; i < clustering.Length; ++i)
          Console.Write(clustering[i] + " ");

        Console.WriteLine("Raw data displayed in clusters is:\n");
        DisplayClustered(tuplesAsInt, numClusters, clustering, attValues);

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

    // Other methods go here

  } // class
} // ns
```

Laplacian smoothing. Laplacian smoothing also adds 3, the number of clusters, to the denominators of the conditional probabilities (but not the unconditional probability term). So the modified computation for the partial for c0 is:

```
PP(c0 | X) = P(Suburban     | c0) *
             P(VeryHigh     | c0) *
             P(Conservative | c0) *
             P(c0)

           = count(Suburban     & co) / (count c0 + 3) *
             count(VeryHigh     & co) / (count c0 + 3) *
             count(Conservative & c0) / (count c0 + 3) *
             1 / numClusters

           = (1 / 4) * (1 / 4) * (2 / 4) * (1 / 3)

           = 0.0104
```

Similarly, the partial probabilities for c1 and c2 are:

```
PP(c1 | X) = count(Suburban     & c1) / (count c1 + 3) *
             count(VeryHigh     & c1) / (count c1 + 3) *
             count(Conservative & c1) / (count c1 + 3) *
             1 / numClusters

           = (1 / 4) * (1 / 4) * (1 / 4) * (1 / 3)

           = 0.0052

PP(c2 | X) = count(Suburban     & c2) / (count c2 + 3) *
             count(VeryHigh     & c2) / (count c2 + 3) *
             count(Conservative & c2) / (count c2 + 3) *
             1 / numClusters

           = (2 / 4) * (2 / 4) * (1 / 4) * (1 / 3)

           = 0.0208
```

After the partials for each cluster have been calculated, it's easy to compute the probabilities for each cluster that are needed to assign tuple 1 to a cluster. Here are the computations:

```
P(c0 | X) = PP(X | c0) / [PP(X | c0) + PP(X | c1) + PP(X | c2)]
          = 0.0104 / (0.0104 + 0.0052 + 0.0208)
          = 0.2857

P(c1 | X) = PP(X | c1) / [PP(X | c0) + PP(X | c1) + PP(X | c2)]
          = 0.0052 / (0.0104 + 0.0052 + 0.0208)
          = 0.1429

P(c2 | X) = PP(X | c0) / [PP(X | c0) + PP(X | c1) + PP(X | c2)]
          = 0.0208 / (0.0104 + 0.0052 + 0.0208)
          = 0.5714
```

The tuple is assigned to the cluster with the greatest probability, which in this case is cluster c2.

There are several ways to store the various data used by the INBIAC clustering algorithm.

To summarize, to assign a tuple to a cluster, the partial probabilities for each cluster are computed using the joint counts of tuples that have already been assigned. The partials are used to compute the probability that the tuple belongs to each cluster. The tuple is assigned to the cluster that has the greatest probability.

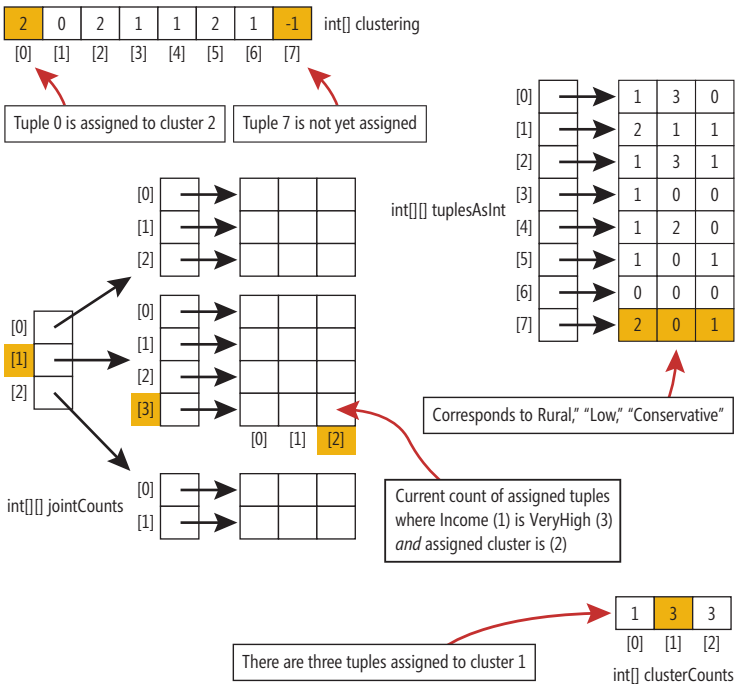


Figure 3 **Key Data Structures**

Key Data Structures

There are several ways to store the various data used by the INBIAC clustering algorithm. **Figure 3** shows most of the data structures used by the demo program. Array jointCounts is used to compute the partial probabilities that in turn are used to compute cluster probabilities that in turn are used to assign a tuple to a cluster. There's one joint count for every combination of attribute value and cluster. So, for the demo, because there are nine attribute values (Urban, Suburban, ... Conservative) and three clusters, there are 9 * 3 = 27 joint counts. The first index in jointCounts indicates the attribute, the second index indicates the attribute value and the third index indicates the cluster. For example, jointCounts[1][3][2] holds the count of assigned tuples where the Income (1) is VeryHigh (3) and cluster is (2).

> The INBIAC algorithm seeds each cluster with a single tuple. It's important that these seed tuples be as different from each other as possible.

Array clustering encodes how tuples are assigned to clusters. The index of array clustering represents a tuple, and the cell value represents a cluster. For example, if clustering[0] = 2, then tuple 0 is assigned to cluster 2. Cell values of -1 indicate the associated tuple has not yet been assigned to a cluster.

Implementing the Clustering Method

Method Cluster is listed in **Figure 4**. The method accepts as inputs the tuplesAsInt array (the data to be clustered), numClusters (the number of clusters to use), numTrials (which assigns initial tuples to clusters), jointCounts (as explained earlier), clusterCounts (the number of tuples assigned to each cluster, needed if computing with non-equal priors) and equalPriors (a Boolean that indicates how to compute probabilities of each cluster when computing partial probabilities).

Method Cluster begins by allocating memory for the clustering array and assigning values of -1 in each cell, to indicate no cluster has been assigned to the associated tuple. Next, helper method GetGoodIndexes is called to get the indexes of the tuples that are maximally different from each other. I'll explain method GetGoodIndexes shortly. Method Cluster next uses the good indexes to initialize the clustering array and then updates the jointCounts and clusterCounts arrays.

The main processing loop iterates through each data tuple in order and computes the probabilities that the current tuple belongs to each cluster using method AllProbabilities. Then the index of the greatest probability is determined using helper IndexOfLargestProbability. Because clusters are zero-based, that index also represents the best cluster, and it's used to assign the current tuple to a cluster (in the clustering array), and update the jointCounts and clusterCounts arrays.

Because the processing loop always starts at tuple [0], this effectively gives more weight to tuples with lower-numbered indexes. An alternative is to walk through the tuples in random order. Notice that the INBIAC algorithm assigns tuples based on the greatest probability of cluster membership. You could also compute and return the average of these greatest probabilities. This would be a measure of confidence in the cluster assignments. Then you could call method Cluster several times and return the clustering that was produced by the call that yielded the highest confidence.

Another option I often use is to post-process the clustering result produced by method Cluster to try and generate a better clustering. The idea in pseudo-code is:

```
loop n times
  select a random tuple index
  unassign the tuple from its curr cluster
  assign the tuple using non-equal priors computation
end loop
```

Recall that INBIAC builds up cluster assignment one tuple at a time by finding the cluster the current tuple belongs to with greatest probability. The probabilities are computed using equal priors, meaning the probabilities of each cluster are assumed to be equal. But after clustering, there's now more information available about how likely each cluster is, and that information can be used to possibly improve the clustering result. The code download implements this option using a method named Refine.

Getting Good Initial Seed Tuples

The INBIAC algorithm seeds each cluster with a single tuple. It's important that these seed tuples be as different from each other as

possible. There are many measures of dissimilarity used by clustering algorithms. Method GetGoodIndexes generates a set of random candidate indexes, then computes the total number of tuple attributes that are different, a metric called the Hamming distance. This process is repeated numTrials times, and the indexes of the tuples that have the greatest dissimilarity are returned.

There are many measures of dissimilarity used by clustering algorithms.

For example, consider the data in **Figure 1**. Suppose the candidate indexes are 0, 1, 2. The corresponding data tuples are:

```
[0] Suburban VeryHigh Liberal
[1] Rural    Medium   Conservative
[2] Suburban VeryHigh Conservative
```

Tuples [0] and [1] differ in three positions; tuples [0] and [2] differ in one position; and tuples [1] and [2] differ in two positions, for a total delta of 3 + 1 + 2 = 6. In pseudo-code, method GetGoodIndexes is:

```
init a result array
loop numTrials times
  generate numClusters distinct random tuple indexes
  compute their dissimilarity
  if curr dissimilarity > greatest dissimilarity
    greatest dissimilarity = curr dissimilarity
    store curr indexes into result array
  end if
end loop
return result array
```

You may wish to consider alternative approaches. One advanced option is to observe that attribute Income—with values Low, Medium, High and VeryHigh—is inherently numeric. So you could modify method GetGoodIndexes so that the difference between Low and VeryHigh is greater than the difference between Low and Medium.

Generating the distinct candidate seed tuple indexes is an interesting little sub-problem. This is performed by helper method GetRandomIndexes. In pseudo-code, the method works like this:

```
init a dictionary to store result indexes
init a result array
set count = 0
while count < numClusters
  generate a random tuple index
  if index not in dictionary
    add index to result set
    add index to dictionary
    increment count
  end if
end while
return result
```

This technique is a fairly brute-force approach, but it has worked well for me in practice.

Wrapping Up

This article, along with the code for the demo program, should give you a solid basis for experimenting with Naive Bayesian clustering and adding clustering functionality to a software system. I developed the INBIAC clustering algorithm while working on a project that had an extremely large data set that contained both numeric and categorical data. I found that existing clustering tools were either too slow, didn't work at all or gave poor results. The algorithm presented here can deal with both categorical and numeric data (if numeric data is binned into categories), can handle huge data sets and is very fast.

As I mentioned in the introduction, research suggests that there's no single best clustering algorithm, but rather that you must experiment with different algorithms to get the best results. The ability to explore data sets with clustering based on Naive Bayes inference can be a valuable addition to your technical skill set. ■

Figure 4 **Method Cluster**

```
static int[] Cluster(int[][] tuplesAsInt, int numClusters,
  int numTrials, int[][][] jointCounts, int[] clusterCounts,
  bool equalPriors)
{
  int numRows = tuplesAsInt.Length;
  int[] clustering = new int[numRows];
  for (int i = 0; i < clustering.Length; ++i)
    clustering[i] = -1;

  int[] goodIndexes = GetGoodIndexes(tuplesAsInt, numClusters, numTrials);

  for (int i = 0; i < goodIndexes.Length; ++i)
  {
    int idx = goodIndexes[i];
    clustering[idx] = i;
  }

  for (int i = 0; i < goodIndexes.Length; ++i)
  {
    int idx= goodIndexes[i];
    for (int j = 0; j < tuplesAsInt[idx].Length; ++j)
    {
      int v = tuplesAsInt[idx][j];
      ++jointCounts[j][v][i];     // Very tricky indexing
    }
  }

  for (int i = 0; i < clusterCounts.Length; ++i)
    ++clusterCounts[i];

  for (int i = 0; i < tuplesAsInt.Length; ++i)
  {
    if (clustering[i] != -1) continue; // Tuple already clustered

    double[] allProbabilities = AllProbabilities(tuplesAsInt[i],
      jointCounts, clusterCounts, equalPriors);
    int c = IndexOfLargestProbability(allProbabilities);
    clustering[i] = c;  // Assign tuple i to cluster c

    for (int j = 0; j < tuplesAsInt[i].Length; ++j)
    {
      int v = tuplesAsInt[i][j];
      ++jointCounts[j][v][c];
    }

    ++clusterCounts[c];
  } // Main loop

  return clustering;
}
```

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: *Dan Liebling*

TED NEWARD

Noda Time

Ever spent much time thinking about time?

Very early in my career, I was working on a system that ended up deploying to several different call centers. Keeping track of "when" an event occurred was particularly important (it was a medical-related system for a call center of nurses), and so, without thinking about it too much, we dutifully wrote the time of the event into a database row and left it at that. Except, as we discovered later, when the system was deployed to four different call centers, each in a different U.S. time zone, the time logs were all a little "off," thanks to the fact that we hadn't thought to include time-zone offsets.

Time in a software system is like that—it all seems pretty straightforward and simple, until it suddenly doesn't anymore.

> My computer doesn't really understand time zones, per se.

In keeping with the theme of my past two columns (all my columns can be found at bit.ly/ghMsco), once again the .NET community benefits from work done by the Java community; in this case, it's a package called "Noda Time," a Microsoft .NET Framework port of the Java "Joda Time" project, which itself was designed as a replacement for the Java "Date" class (a horribly broken piece of software dating back to the days of Java 1.0). Jon Skeet, the author of Noda Time, based it on the algorithms and concepts in Joda Time, but built it from the ground up as a .NET library.

Enough preamble: Do an "Install-Package NodaTime" (notice no space between "Noda" and "Time"), and let's look at some code.

'Me' Time

The first thing to realize is that, with all due respect to Einstein's theories, you don't have to be approaching light speed to realize that time is relative. If it's 7 p.m. (that's 1900 to you European folks) here in Seattle, then it's 7 p.m. for all of us in Seattle, but it's 8 p.m. for my folks in Salt Lake City, 9 p.m. for my travel agent in Dallas and 10 p.m. for my drinking buddy in Boston. We all get that—that's the magic of time zones. But my computer doesn't really understand time zones, per se—it reports the time it's been set to, which in this case is 7 p.m., despite the fact that it's the exact same moment in time for all of us around the world. In other words, it's not that time itself is relative, it's that our representation of time is relative. In Noda Time, this representation is reflected as "global" time, meaning a moment on a universal timeline with which everyone agrees. What we consider to be "local" time—that is, that time with an associated time zone—Noda Time calls "zoned time," as opposed to what Noda Time considers "local" time, which is without any time zone attached (more on this later).

So, for example, the Noda Time "Instant" refers to a point on the global timeline, with an origin point of midnight on Jan. 1, 1970, Coordinated Universal Time (UTC). (There's nothing magical about that date, except that this is by convention the "start of the epoch" for Unix systems counting "ticks" since that date, and thus serves as good a reference origin point as any other.) So, doing this gives us the current Instant (assuming that the Noda Time namespace is referenced—"using NodaTime"—of course):

```
var now = SystemClock.Instance.Now;
```

To get a Seattle-relative time, we want a ZonedDateTime. This is essentially an Instant, but with the time zone information included, so that it identifies itself as being relative to "Seattle, on Jan. 9, 2013" (the date being important because we need to know whether we're in daylight saving time [DST] or not). A ZonedDateTime is obtained through a constructor, passing in an Instant and a DateTime zone. We have the Instant, but we need the DateTime zone for Seattle in DST. To obtain the DateTime zone, we need an IDateTime zoneProvider. (The reason for this indirection is subtle, but has to do with the fact that the .NET Framework uses a representation for time zones different from any other programming platform; the Internet Assigned Names Authority, or IANA, uses a format like "America/Los_Angeles.") Noda Time offers two built-in providers, one being the IANA version and the other the standard .NET base class library (BCL) version, through static properties on the DateTime zoneProviders class:

```
var seattleTZ = dtzi["America/Vancouver"];
var dtzi = DateTime zoneProviders.Tzdb;
```

The time zone database (TZDB) version is the IANA version, and so obtaining the time zone that represents Seattle is a matter of selecting it (which, according to IANA, is "America/Los_Angeles," or if you want something closer, "America/Vancouver"):

```
var seattleNow = new ZonedDateTime(now, seattleTZ);
```

And if we print this out, we get a representation of "Local: 1/9/2013 7:54:16 PM Offset: -08 Zone: America/Vancouver." Notice that "Offset" portion in the representation? It's important, because remember that based on what day of the year it is (and what country you're in, and what calendar you're operating under, and ...), the offset from UTC will change. For those of us in Seattle, DST means gaining or losing an hour off the local clock, so it's

important to note what the offset from UTC is. In fact, Noda Time keeps track of that separately, because when parsing a date such as "2012-06-26T20:41:00+01:00," for example, we know that it was one hour ahead of UTC, but we don't know if that was because of DST in that particular time zone or not.

Still think time is easy?

'Us' Time

Now let's assume that I want to know how long until an important date in my life—such as my 25th wedding anniversary, which will be on Jan. 16, 2018. (Keeping track of such things is somewhat important, as I think any spouse would tell you, and I need to know how long I have before I have to buy a really expensive gift.) This is where Noda Time is really going to shine, because it's going to keep track of all the niggling little details for you.

> It's impossible to write tests based on time, particularly because time has this annoying habit of continuing on.

First, I need to construct a LocalDateTime (or, if I didn't care about the time, a LocalDate; or, if I didn't care about the date, a LocalTime). A LocalDateTime (or LocalDate or LocalTime) is a relative position on the timeline that doesn't know exactly what it's relative to—in other words, it's a point in time without knowing its time zone.

(Despite not knowing the time zone, this is still a useful bit of information. Think of it like this: If you and I are working together in the same office, and we want to meet later today, I'll say, "Let's meet at 4 p.m." Because we're both in the same time zone, no additional information is necessary to qualify this time unambiguously to each other.)

So, because I know the point in time that I care about already, it's easy to construct:

```
var twentyFifth = new LocalDate(2018, 1, 16);
```

And because I'm just asking about the difference of two dates without concern for the time zones, I only need the LocalDate part of the LocalDateTime part of the ZonedDateTime from my earlier calculation:

```
var today = seattleNow.LocalDateTime.Date;
```

But what we're asking here is for a new kind of time unit: a "period" between two times. (In the BCL, this is a Duration.) Noda Time uses a different construct to represent this—a Period—and like all properly represented units of measure, it requires a unit to go with it. For example, the answer "47" is useless without an accompanying unit, such as "47 days," "47 hours" or "47 years." Period provides a handy method, Between, to calculate the number of some particular time unit between two LocalDates:

```
var period = Period.Between(today, twentyFifth, PeriodUnits.Days);
testContextInstance.WriteLine("Only {0} more days to shop!", period.Days);
```

This tells me exactly how many days, but we don't usually count days at that large an amount (1,833, at the time I wrote this article) like that. We prefer time to be in more manageable chunks, such as "years, months, days," which we can again ask Noda Time to manage. We can ask it to give us a Period that contains a years/months/days breakdown, by OR-ing the PeriodUnits flags together:

```
Period.Between(today, twentyFifth,
  PeriodUnits.Years | PeriodUnits.Months | PeriodUnits.Days)
```

Or, because this is a pretty common request, we can ask it to give us a Period that contains a years/months/days breakdown by using the pre-constructed flag of the same name:

```
Period.Between(today, twentyFifth, PeriodUnits.YearMonthDay)
```

(Apparently, I still have a little time yet, which is good, because I have no idea what to get her.)

Testing Time

Frequent readers of this column will know that I like to write exploration tests when investigating a new library, and this column is no exception. However, it's impossible to write tests based on time, particularly because time has this annoying habit of continuing on. Each millisecond that passes throws off whatever the expected result is, and that makes it hard, if not impossible, to write tests that will produce predictable results that we can assert and report violations.

For this reason, anything that provides time (such as, you know, a clock) implements the IClock interface, including the SystemClock I used earlier to obtain the Instant for "right now" (the Now static property). If we, for example, create an implementation that implements the IClock interface and provides a constant value back for the Now property (the only member required by the IClock

Figure 1 **Creating Exploration Tests**

```
[TestClass]
public class UnitTest1
{
  // SystemClock.Instance.Now was at 13578106905161124 when I
  // ran it, so let's mock up a clock that returns that moment
  // in time as "Now"
  public class MockClock : IClock
  {
    public Instant Now
    {
      get { return new Instant(13578106905161124); }
    }
  }

  [TestMethod]
  public void TestMethod1()
  {
    IClock clock = new MockClock(); // was SystemClock.Instance;
    var now = clock.Now;
    Assert.AreEqual(13578106905161124, now.Ticks);

    var dtzi = DateTime zoneProviders.Tzdb;
    var seattleTZ = dtzi["America/Vancouver"];
    Assert.AreEqual("America/Vancouver", seattleTZ.Id);

    var seattleNow = new ZonedDateTime(now, seattleTZ);
    Assert.AreEqual(1, seattleNow.Hour);
    Assert.AreEqual(38, seattleNow.Minute);

    var today = seattleNow.LocalDateTime.Date;
    var twentyFifth = new LocalDate(2018, 1, 16);
    var period = Period.Between(today, twentyFifth, PeriodUnits.Days);
    Assert.AreEqual(1832, period.Days);
  }
}
```

interface, in fact), because the rest of the Noda Time library basically uses Instants to recognize that moment in time, we have essentially created an entirely testable environment, which allows the entire thing to be tested, asserted and verified. Thus, I can change my earlier code slightly and create a set of exploration tests, as shown in **Figure 1**.

By going through all of your time-related code and using Noda Time instead of the built-in .NET time types, code becomes much more testable simply by replacing the IClock used to obtain the Instant for "right now" to something controllable and known.

But Wait ...

There's a lot more to Noda Time than just what I've shown here. For example, it's relatively easy to add time units (days, months and so on) to a given time by using the "Plus" and "Minus" methods (for which there are also operator overloads, if those make more sense to use), as well as a FakeClock class designed specifically for testing time-related code, including the ability to programmatically "advance" time in some discrete fashion, making it easier to test elapsed-time-sensitive code (such as Windows Workflow instances, for example, that are supposed to act after a period of time has elapsed without activity).

At a deeper, more conceptual level, Noda Time also demonstrates how a type system in a programming language can help differentiate between subtly different kinds of values within the problem domain: By separating out the different kinds of time (instants, local time, local dates, local dates and times, and zoned dates and times, for example) into discrete and interrelated types, it helps the programmer be clear and explicit about exactly what this code is supposed to be doing or working with. It can be particularly important, for example, to differentiate a "birth date" from a "birth day" in some code: One reflects the moment in the universe's timeline when a person was born, the other is a recurring date on which we celebrate that moment in the universe's timeline. (Practically speaking, one has a year attached to it, the other doesn't.)

Skeet has made it clear that he doesn't consider the library "finished" in any way, and he has plans to enhance and extend it further. Fortunately, Noda Time is available for use today, and developers owe it to themselves to NuGet Noda Time, have a look, and start figuring out how and where to use it in the problem domain. After all, time is precious.

Happy coding! ■

Ted Neward *is a principal with Neward & Associates LLC. He has written more than 100 articles and authored and coauthored a dozen books, including "Professional F# 2.0" (Wrox, 2010). He is an F# MVP and noted Java expert, and speaks at both Java and .NET conferences around the world. He consults and mentors regularly—reach him at ted@tedneward.com 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:
Jon Skeet

RACHEL APPEL

Data Access and Storage Options in Windows Store Apps

Managing data is a critical part of app development. Whether it's a game, news, travel or fashion app, it's always all about the data. Modern apps often need to manage data scattered throughout multiple, disparate locations and in countless formats. I'll discuss the various data storage options and data access APIs available for building Windows Store apps, in all languages, as well as data management strategies for both content and configuration.

Data Management and Storage Considerations

As an app developer, you need to determine your app's data requirements prior to starting your project, because changing the underlying architecture causes a lot of rework. You might have an existing data source, in which case the decision is made for you, but with a greenfield project, you must think about where to store the data. Your two options are on the device or at a remote location:

- **Local:** Usually this data is in a file or local database, but in Windows 8, you can now treat other apps as sources for data by using the built-in File Picker or contracts. In JavaScript apps, Web Storage and the Indexed Database (IndexedDB) API are also available as local data sources.
- **Remote:** This data could be in the cloud using Windows Azure SkyDrive or any remote HTTP endpoint that can serve JSON or XML data, including public APIs such as Facebook or Flickr.

The size of the data often determines whether the data is local or remote; however, most modern apps will use data from both sources. This is because smaller, more mobile devices such as slates, tablets and phones are the norm, and they don't usually have much storage space. Despite that, they still need data to function correctly when offline. For example, the Surface, as with many portable devices, comes in 32GB and 64GB models. Simple text-based data such as JSON isn't usually large, but relational databases and media data (such as images, audio and video) can fill up a device quickly.

Let's take a look at the various local and remote options for storing app content data.

Web Storage

It might sound like Web Storage (bit.ly/lml0UI) is simply storage on the Web, but it isn't. Instead, Web Storage, an HTML5 standard, is a great way to keep app data on the client, locally. Both Windows Store apps as well as plain old HTML pages support Web Storage. There's no database setup required and no files to copy, as Web Storage is an in-memory database.

Web Storage is accessible via JavaScript through one of the two following properties of the window object:

1. **localStorage:** Local data that's persistent after the app terminates and is available to future app instances.
2. **sessionStorage:** Also local data; however, sessionStorage is destroyed when a Windows Store app terminates execution.

You can store data from simple types to complex objects in Web Storage by attaching dynamic properties to either the sessionStorage or localStorage variables. The dynamic properties are a key/value pair with syntax similar to this:

```
sessionStorage.lastPage = 5;
WinJS.xhr({ url: "data/data.json" }).then(function (xhr) {
  localStorage.data = xhr.responseText;
};
```

The lastPage property exists until the app terminates because it's part of sessionStorage, while the data property of localStorage persists past the lifetime of the app.

Being able to persist data locally between app sessions makes Web Storage an excellent choice for supporting offline scenarios. Small data is also more suited for offline support. Because JSON data is compact, it's easy to stuff entire JSON datasets into the 5MB of space provided by Web Storage and have plenty of space left over for some media data.

Because Web Storage is an HTML5 standard, it's only available in Windows Store projects built with JavaScript.

IndexedDB

Another standard in the HTML5 family is IndexedDB (bit.ly/TT3btM), which is a local data store for large, searchable and persistent data sets. As a component of HTML5, you can use IndexedDB in client Web apps for browsers as well as Windows Store apps. IndexedDB stores items in an object database and is extremely flexible because you can store any kind of data from text to Binary Large Objects (BLOBs). For example, multimedia files tend to be quite large, so storing audio and video in IndexedDB is a good choice.

Modern Apps LIVE!

MODERN APPS FROM START TO FINISH

Presented in Partnership with Magenic

MARCH 26-28, 2013
MGM GRAND HOTEL
LAS VEGAS, NV

YOUR BACKSTAGE PASS TO THE MICROSOFT PLATFORM

Development Managers, **Software Architects** and **Development Leads** are gathering in Las Vegas, March 26-28, to learn the latest and greatest techniques in low-cost, high-value application development at **Modern Apps Live!**.

What makes **Modern Apps Live!** so unique is the singular topic focus; sessions build on each other as the conference progresses, leaving you with a holistic understanding of modern applications.

MODERN APPS FROM START TO FINISH

BUY 1, GET 1 FREE!

CO-LOCATED WITH:

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ATTENDEES OF MODERN APPS LIVE! WILL HAVE FULL ACCESS TO ALL VISUAL STUDIO LIVE! LAS VEGAS SESSIONS (AND PRIORITY SEATING IN ALL MODERN APPS LIVE! SESSIONS).

Modern Apps Live!
Conference Chair
Rockford Lhotka
CTO, Magenic

Session topics include (in order):

- **TFS Setup & Visual Studio Project Setup**
- **Business Layer Implementation**
- **Database Design/ Implementation**
- **Data Access Layer Implementation**
- **App Server Configuration/ Deployment: Win2012, Azure**
- **UX design**
- **Using Azure Mobile Services**
- **UX Win8 Implementation**
- **UX WP8 Implementation**
- **UX iOS Implementation**
- **BI Design/ Implementation**
- **QAT and Automated Testing**
- **Modern App Deployment**

REGISTER TODAY!

Use promo code MALM1

MODERNAPPSLIVE.COM

PRODUCED BY 1105 MEDIA

Because IndexedDB is an object database, it doesn't use SQL statements, so you must access data through an object-oriented-style syntax. Interaction with an IndexedDB data store is through transactions and cursors, as shown here:

```
var dataStore = "Datastore";
var trn = db.transaction(dataStore, IDBTransaction.READ_ONLY);
var store = trn.objectStore(dataStore);
trans.oncomplete = function(evt) { // transaction complete };
var request = store.openCursor();
request.onsuccess = function(evt) {
  var cursor = evt.openCursor();
};
request.onerror = function(error) { // error handling };
```

Because IndexedDB specializes in really big data, using it for small item storage causes it to behave inefficiently, making Web Storage a much better choice for bite- (or byte-) sized local data. IndexedDB is also well-matched for content data, but ill-suited for app configuration data.

SQLite

SQLite (bit.ly/65FUBZ) is a self-contained, transactional, relational and file-based database, which requires no configuration and doesn't need a database administrator to maintain it. You can use SQLite with any Windows Runtime (WinRT) language, and it's available as a Visual Studio extension. While SQLite works well in JavaScript apps, you need to obtain the SQLite3 WinRT wrapper (bit.ly/J4zzPN) from GitHub before using it.

Developers with backgrounds in ASP.NET or Windows Forms gravitate to relational databases, but a relational database management system (RDBMS) isn't always the best choice when writing modern apps, due to space issues on mobile devices as well as the varied types and formats of data, especially multimedia. Because SQLite is a relational database, it makes sense for those apps that need relational and transactional behaviors. This means SQLite is great for line-of-business (LOB) apps or data-entry apps, and can also be a repository for local, offline data originally obtained from an online source.

If the SQLite database becomes too large for portable devices, you can move it to a server or cloud location. The code won't change much, because the SQLite3 library uses a traditional connection and Create/Read/Update/Delete (CRUD) objects similar to the following code:

```
// C# code that opens an async connection.
var path =
  Windows.Storage.ApplicationData.Current.LocalFolder.Path + @"\data.db";
var db = new SQLiteAsyncConnection(path);
// JavaScript code that opens an async connection.
var dbPath =
  Windows.Storage.ApplicationData.current.localFolder.path + '\\data.db;
SQLite3JS.openAsync(dbPath).then(function (db) {
  // Code to process data.
});
```

As you can see, using SQLite is just like using other SQL databases. Limits on SQLite databases go as high as 140TB. Keep in mind that very large data often warrants professional database administration for the best possible data integrity, performance and security. Most DBAs and developers who work with relational databases prefer a GUI tool to create and manage database objects or run ad hoc queries on the data, and the Sqliteman (bit.ly/9LrB1o) admin utility is ideal for all basic SQLite operations.

If you're porting existing Windows desktop apps written in Windows Forms, Windows Presentation Foundation (WPF) or Silverlight, you might already be using SQL Server Compact (SQL CE). If this is the case, you can migrate SQL CE databases to SQLite with the ExportSqlCE (bit.ly/dwVaR3) utility and then use Sqliteman to administer them.

Files as Data and the File API

Why bother with a database at all, especially if users of your app want to stick with the files they already have? This is especially true of photos and documents. The File API goes beyond simply providing a navigator to directories and files; it gives the user the ability to choose an app as a file location. This means apps can talk to each other and exchange data. A File Open Picker can interact with Bing, camera, or photo apps as well as regular directories and named locations such as My Documents, Videos, or Music. Sharing data so easily between apps, services and personal files is a first-rate feature of Windows.

Many OSes have a mechanism for registering types of files that apps intend to use alongside the ability to launch those apps when a user interacts with an icon in the OS. In Windows, this is called a file association. Windows 8 takes this concept further by allowing apps to talk to each other through a system-wide feature called contracts. One way to implement a contract is through a File Picker. Notice that the following code is similar to the File Dialog APIs from desktop apps or earlier Windows versions (note: some code has been left out of this example for brevity; for a more thorough examination of the FileOpenPicker class, see bit.ly/UztmDv):

```
fileOpen: function () {
  var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
  openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
  openPicker.suggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
  openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);
  openPicker.pickSingleFileAsync().done(function (file) {
  // ...
  });
}
```

Choosing an app from a Windows 8 picker launches that app. For example, if the user selects Bing, the picker will launch the Bing app and then return the user's image selection to your app.

Now that you've seen the local options, let's look at the options for remote data.

Web Services and the ASP.NET Web API

Most developers are familiar with consuming and modifying data with XML Web services because they date back to the days of the Microsoft .NET Framework 1.x. The main advantage in using Web services is that the data lives at a central remote location and the apps from multiple devices can access the data any time while connected. Access to the underlying database is piped through a set of HTTP endpoints that exchanges JSON or XML data. Many public APIs such as Twitter or Flickr expose JSON or XML data that you can consume in Windows Store apps.

Web services come in a variety of flavors:

- ASMX services: Use traditional ASP.NET to deliver data across HTTP.
- Windows Communication Foundation (WCF) and Rich Internet Application (RIA) services: A message-based way to send data between HTTP endpoints.

- OData: Open Data protocol, another API for transporting data over HTTP.
- ASP.NET Web API: A new ASP.NET MVC 4 framework that makes it easy to build RESTful HTTP services that deliver JSON or XML data to apps or Web sites.

If you're building back-end services from the ground up, the ASP.NET Web API streamlines the development process because it makes it easy to build services in a consistent RESTful fashion, so they can be readily consumed by apps.

Regardless of the back-end service, if it's over HTTP, you can use the HttpWebRequest and HttpWebResponse objects to communicate with a Web service via C#. In JavaScript apps, a WinJS XMLHttpRequest wrapper is fitted for asynchronous operations, which looks like the following code:

```
WinJS.xhr({ url: "data.json" }).then(function (xhr) {
  var items = JSON.parse(xhr.responseText);
  items.forEach(function (item) {
    list.push(item);
  })
});
```

Storage space isn't usually an issue with Web services because a Web service is just the software that transports the data between endpoints. This means its underlying database can live anywhere, such as on a remote server, Web host or Windows Azure instance.

You can host any of the previously noted Web services such as an ASP.NET Web API instance or a WCF service on Windows Azure, along with the data itself.

SkyDrive

Don't forget that SkyDrive (bit.ly/HYB7iw) is not only an option for data storage, but an excellent option. Think of SkyDrive as a non-storage storage option. Users can choose SkyDrive as a location in the cloud and access documents through File Open or Save pickers. Allowing users to save files to SkyDrive means zero worries about database management, and with 7GB of space per user, there's plenty of room. SkyDrive users can also purchase more storage space.

The Live API (bit.ly/mPNb03) contains a fully featured set of RESTful SkyDrive APIs for reading and writing to SkyDrive. A call to SkyDrive to retrieve the list of shared items looks like this:

```
GET https://apis.live.netv5.0/me/skydrive/shared?access_token=ACCESS_TOKEN
```

A Microsoft.Live namespace allows C# developers to access the Live and SkyDrive APIs, while JavaScript developers can make RESTful calls with HTTP verbs POST or PUT. SkyDrive isn't recommended for app configuration data.

Windows Azure Mobile Services

Windows Azure Mobile Services is a great option for those building cross- and multi-platform apps. Powered by Windows Azure, this option offers more than just scalable storage; it offers push notifications, business logic management, an authentication API and a complete SDK. In addition to these features, an easy-to-use, Web-based administrative tool is provided.

The Mobile Services SDK integrates with Windows Store, Windows Phone 8, iOS and Android apps. All of the major platforms are supported, and with the Mobile Services SDK, you can build out a working prototype in just minutes and be on your way to delivering data to multi-platform apps in no time.

Among the many items in the SDK, you can use the query object to query data from tables, similar to this:

```
var query = table.orderBy('column').read({ success: function(results) { ... }});
```

As you can see, the code is quite the same as any other API, so the learning curve is the same as the other options discussed here. You can access the SDK and Mobile Services using any Windows Store app language.

Application Data Management and Storage Options

All of the previously noted data storage and access options are for storing content, but as a developer, you also need to deal with configuration data for the app. This is the metadata that describes your app or the capabilities of its device, not the user's data. Modern apps make use of both persistent application data (such as user preferences) as well as temporary metadata (such as the user's last scroll position or trending search terms). These small yet effective conveniences ensure the best-possible experience for the user, so it's important to build them into your app.

While some of this data belongs on the device, consider the fact that many apps work across multiple platforms and devices, so centralizing and synchronizing application data in Windows Azure with the content data often makes sense.

A specific set of APIs for managing application data exists in an aptly named ApplicationData class in the Windows.Storage namespace, and the code looks similar to this:

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;
var roamingSettings = Windows.Storage.ApplicationData.current.roamingSettings;
```

You can store simple or complex objects in either the localSettings or roamingSettings properties.

Using either local or roaming settings is the preferred way to work with configuration data. Technologies such as IndexedDB, files or SkyDrive ordinarily aren't good options for app configuration data, and SQLite makes sense here only if the app is already using it to store content. You must also consider what to do when the app is offline or disconnected from the Internet. In other words, some of your data needs to be cached, but without consuming too much disk space.

Content and Configuration

In summary, to implement a proper data architecture, you shouldn't depend on the limited space on portable devices, so hosting data in the cloud is usually a good bet. But, if you have a legacy database, reusing it might be a requirement. Windows Store apps support a variety of structured and BLOB storage needs for both content and configuration.

If you're building a Window Store app and would like assistance with choosing a data access strategy, I highly recommend that you check out the Generation App (bit.ly/W8GenAppDev) program. Generation App guides you through the process of building a Windows Store (or Windows Phone) app in 30 days by 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 experts for reviewing this article: *Scott Klein and Miriam Wallace*

DAVID S. PLATT

Grieving

My last two columns talked about how our Internet software and content have drastically reshaped some industries (higher education) and will soon reshape others (medicine). This month I want to discuss how our software is reshaping a universal human experience. And I don't mean taxes.

A good friend of mine, a longtime reader named Lloyd, called me for the first time in months. I inquired about his 20-year-old daughter, Lauren, who had been ill when we last talked. "She passed away about three weeks ago," he said, pain etching his voice.

If you've read this column regularly, you know how crazy I am about my own daughters, now 12 and 10. My job was to change their diapers. Their job will be to close my eyes the final time and to weep at my funeral. Neither job should happen the other way around. If there's a worse piece of news than the one Lloyd got, I don't want to know what it is.

Lloyd's family did something unthinkable even a decade ago. They left Lauren's Facebook account open, and allowed anyone to post on it. The outpouring of love, of others being saddened and sharing the grief, has been a godsend to the family and to everyone else who loved her. The kids at the special-needs camp where she worked. The classmates from the nursing school she had to drop out of. "There was a tribute from a clinic that she'd supported in Peru," said Lloyd. "They wrote in Spanish, of course, which she spoke fluently. I'd never known she was helping them."

This simultaneous grieving together, without regard for geographic location, is new to the human experience. It does not change the sad facts. I wouldn't say that it makes the grief easier to bear. I think it's more that all these people coming together give each other the strength to bear the unbearable.

I doubt the creators of Facebook were thinking of communal grieving when they designed their site. They were probably thinking of the usual I'm-at-the-beach-eat-your-heart-out photos that we see way too many of. Or viewing real-time pictures of a bar's patrons so you can see who's there, and decide if it's worth leaving your pre-gaming (getting drunk on cheaper booze before heading out to the expensive bar). You know, the stuff that the Facebook creators do themselves. But users with different priorities and desires have repurposed Facebook to suit themselves, in a way the original creators never imagined.

We see this all the time in software. We build something for one reason and it gets used for others we never thought of. DLLs were originally built to share one copy of Windows code (CreateWindow and so on) among multiple apps. They then got repurposed to support language localization. Now they're used for dependency injection of mock or stub objects in test-driven development. I've often said that your software isn't successful until it's been used in a way that you never imagined.

So the next time you're brainstorming user stories, certainly take the most obvious and likely ones. But also take 15 minutes and brainstorm the less-likely ones and work through those ideas, too. Though I guarantee you won't think of things an imaginative or pain-wracked user will.

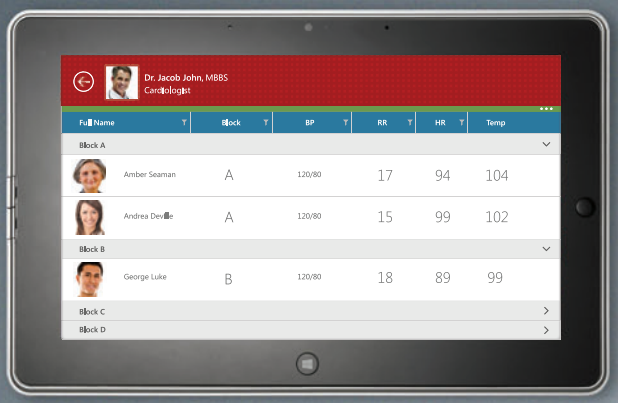
The next time you're brainstorming user stories, certainly take the most obvious and likely ones. But also take 15 minutes and brainstorm the less-likely ones and work through those ideas, too.

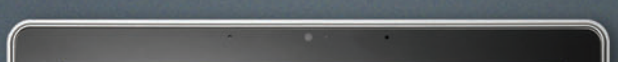
They're still not quite sure what Lauren died of, except that it took a long time and it sucked. So the usual link to contribute to the fight against whatever it was doesn't fit here. Instead, Lloyd asks, "Just help someone who needs it, wherever you are." And I'll add: If this column—or anything else I've ever written—has ever meant anything to you, then do it right now. I don't care what (a hot meal, a warm coat, a prescription refill someone can't afford, a stiff drink and a shoulder to cry on—anything) or whom. But don't dump it onto your endlessly ignored to-do list on your smartphone. Before the sun sets, if it's up now, or before it rises again if it's down, go *do* it. Thank you.

At the end of our call, I could hear Lloyd squaring his shoulders and preparing to put one foot in front of the other. "Hey, Dave, you know what?" he said. "You ought to write about this." Consider it done, friend. ■

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.





Largest Enterprise Toolkit for WinRT and Windows Phone 8 Development

The best toolkit just got better.

- All-new WinRT DataGrid control
- 10+ new controls for WinRT
- 20+ new controls for Windows Phone 8

WinRT + Windows Phone 8 = $199

Limited-time offer

Syncfusion

Download Now

www.syncfusion.com/winrtmsdn