THE MICROSOFT JOURNAL FOR DEVELOPERS

MARCH 2014 VOL 29 NO 3

msdn magazine

THE MICROSOFT JOURNAL FOR DEVELOPERS

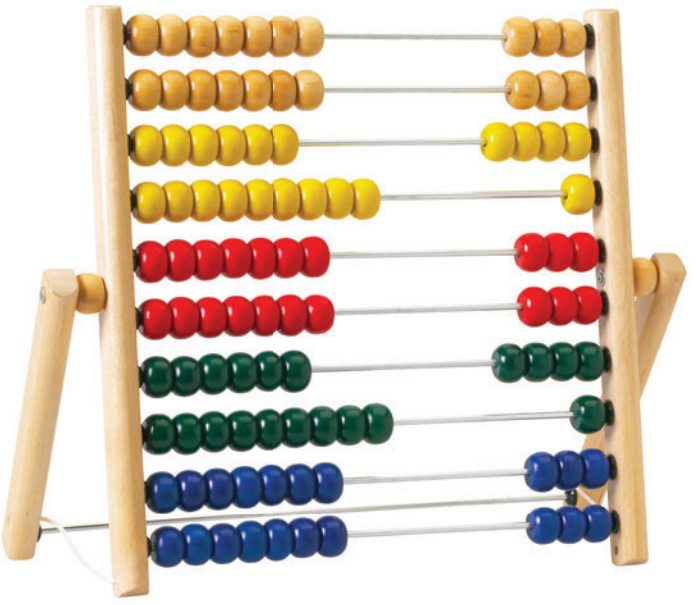
MARCH 2014 VOL 29 NO 3

msdn magazine

Asynchronous Programming.....18, 26

COLUMNS

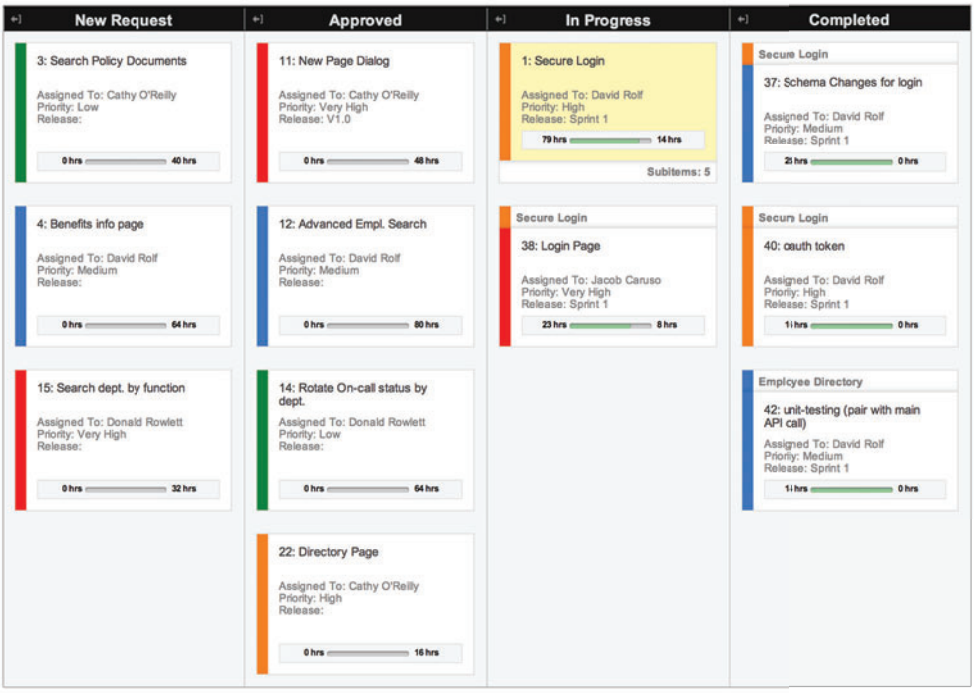
Microsoft



VS.

VS.

OnTime THE #1 SELLING SCRUM SOFTWARE



Boosting your Scrum team's efficiency is easy. Make the switch from sticky notes and upgrade to OnTime at **OnTimeNow.com/MSDN**.

msdn magazine

MARCH 2014 VOLUME 29 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.

DOCUMENT

- *OCR, Barcode & Forms Recognition*
- *PDF Read, Write & Edit*
- *Cleanup and Preprocessing*
- *Annotation and Markup*

MEDICAL

- *DICOM*
- *PACS*
- *Medical Workstation*
- *Image Processing*

MULTIMEDIA

- *Playback, Capture & Conversion*
- *MPEG-2 Transport Stream*
- *Distributed Transcoding*
- *DVR*

IMAGING

- *Raster and Vector Imaging*
- *Viewers*
- *Imaging Processing*
- *150+ Formats*

LEADTOOLS comprehensive line of imaging SDKs have all the document, medical, imaging and multimedia technology you need to develop powerful applications on any platform including .NET, Windows API, WinRT, iOS, OS X, Android, HTML5 and more.

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEADTOOLS®
THE WORLD LEADER IN IMAGING SDKs

SALES@LEADTOOLS.COM
800.637.1840

MICHAEL DESMOND

Everything I Need to Know I Learned from Calvin and Hobbes

Don't Get Me Started columnist David Platt this month dives into the revolt at Avon over the company's attempt to deploy an SAP-based order entry and customer management system. Our back-page columnist takes his cues from many a muse, be they Nobel-winning physicists or cartoon characters from the funny pages. And in that last regard, he and I share common inspiration.

When Bill Watterson's brilliant Calvin and Hobbes comic strip exploded onto newspaper pages in 1985, it was an unexpected well spring of insight and wisdom. As a parent, I've marveled at Watterson's ability to capture the simple genius of a boy at play. And as editor in chief of *MSDN Magazin*e, I've found that Watterson's incorrigible 6-year-old, Calvin, and his loyal tiger, Hobbes, offer real lessons for working developers. Here are just a few.

Test, Test, Test! The Duplicator story arc is one of my favorites in the 10-year run of the comic, but it's a cautionary tale for developers. Calvin invented a box that creates copies of himself, who he hoped would do all his chores and school work. But Calvin never tested his Duplicator, and he quickly faced a squad of ill-behaved dupes. If Calvin had designed a test to determine the actual behavior of the dupes his invention created, he might have saved himself a lot of work.

Remediate: Calvin later developed an add-on for his Duplicator, called the Ethicator, which let the operator set each dupe's personality to either Good or Evil. A simple patch saved what would otherwise have been a costly project failure, as Calvin created a compliant, good-aligned dupe to do his chores.

Fail Gracefully: Alas, the good Calvin dupe tried to befriend Calvin's nemesis Susie Derkins. "I don't mind if he cleans my room and gets me good grades," Calvin griped, "but when he starts talking to girls that's going too darn far." The unpredicted behavior led to an angry confrontation between Calvin and his dupe, who suddenly cried "Oops! I've had an evil thought!" and vanished in a puff of smoke. An exception-handling routine could have preserved the investment in the duplicate Calvin.

Value Extensibility Then there was the Transmogrifier, which could turn anyone into one of four target animals: eel, baboon, giant bug or dinosaur. Calvin showed great awareness allowing support for additional targets, including an extensible UI to handle them. The Transmogrifier would later support worms, elephants, tigers and giant slugs. I wonder if he used XML?

Leverage the platform Both the Duplicator and Transmogrifier—as well as later Calvin inventions the Cerebral Enhance-O-Tron and the Time Machine—were built on a common, corrugated cardboard box platform and permanent marker UI. Simple geometries, familiar materials and streamlined interfaces defined all four inventions.

Don't Skimp on Security When Calvin and Hobbes created their exclusive club, "Get Rid Of Slimy girlS (G.R.O.S.S)," they secured entry to the club treehouse with a long, multi-verse password about tigers, which ended with the line "Tigers are great! They're the toast of the town. Life's always better when a tiger's around!" That final stanza alone is a 308-bit password, and I haven't even described the dancing component. But Calvin struggled to remember the verse, illuminating the deep challenge of balancing usability and security.

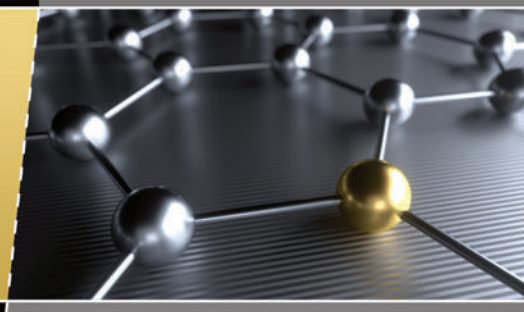
Mind the org chart: G.R.O.S.S. offered a final, valuable lesson—the danger posed by vague, shifting or tangled lines of authority. Calvin may have been "Dictator for Life" of G.R.O.S.S., but that didn't stop "First Tiger" Hobbes from trying to usurp his authority. Constant management reorgs created a volatile environment that produced hijacked meetings, failed initiatives and constant, internecine bickering. G.R.O.S.S. never did mount a successful attack on Susie Derkins.

Make Space for Creativity If Watterson's protagonists have one message for developers, it's this: Dare to dream. Some of Calvin's greatest insights occur while careening through the woods in a toboggan or wagon. Take risks. Make mistakes. And, remember, life's always better when a tiger's around.

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

Empower Your Customers

New! v5.0



Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents
- Create, fill-out and annotate PDF forms
- Fast and lightweight 32- and 64-bit components for .Net and ActiveX/COM
- New WinRT Component enables publishing C#, C++CX or Javascript apps to Windows Store
- New Postscript/EPS to PDF conversion module

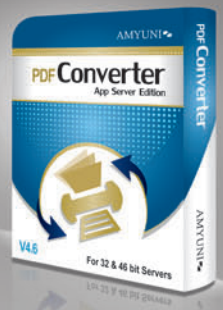
Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment
- Includes our Microsoft certified PDF Converter printer driver
- Export PDF documents into other formats such as JPeg, PNG, XAML or HTML5

Advanced HTML to PDF & XAML

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

High Performance PDF Printer For Desktops and Servers



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2012 and Windows 8
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language
- Easy licensing and deployment to fit system administrator's requirements

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

AMYUNI

All development tools available at

www.amyuni.com

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

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

DINO ESPOSITO

A First Look at ASP.NET Identity

Offspring of the "One ASP.NET" approach to Web development that came with Visual Studio 2013, the new ASP.NET Identity system is the preferred way to handle user authentication in ASP.NET applications, whether based on Web Forms or MVC. In this column, I'll review the basics of ASP.NET authentication and explore the new ASP.NET Identity system from the perspective of ASP.NET MVC 5 developers.

ASP.NET has long supported two basic types of authentication: Windows authentication and forms authentication. Windows authentication is seldom practical for public Web sites because it's based on Windows accounts and access control list (ACL) tokens. Thus, it requires users to have a Windows account in the application's domain, and it also assumes clients are connecting from Windows-equipped machines. The other option is forms authentication, a widely adopted approach. Forms authentication is based on a simple idea. For each access to a protected resource, the application ensures the request includes a valid authentication cookie. If a valid cookie is found, then the request is served as usual; otherwise, the user is redirected to a login page and asked to provide credentials. If these credentials are recognized as valid, then the application issues an authentication cookie with a given expiration policy. It's simple and it just works.

> The simple membership API has become quite a popular way of managing authentication.

Implementation of any forms authentication module can't happen without a distinct module that takes care of collecting user credentials and checking them against a database of known users. Writing this membership subsystem has been one of the key responsibilities of development teams—but also one of the most annoying things ever. Writing a membership system is not hard, per se. It mostly requires running a query against some sort of storage system and checking a user name and password. This code is boilerplate and can grow fairly big as you add new authentication features such as changing and recovering passwords, handling a changing number of online users and so on. In addition, it has to be rewritten nearly from scratch if you change the structure of the storage or add more information to the object that describes the user. Back in 2005, with the release of ASP.NET 2.0, Microsoft addressed this problem by introducing right into the framework a provider-based architecture and the membership provider. Instead of reinventing the wheel every time, you could just derive membership from the built-in system and override only the functions you intended to change.

Figure 1 **Foundation of a Controller Based on ASP.NET Identity**

```
public class AccountController : Controller
{
  public UserManager<ApplicationUser> UserManager { get; private set; }

  public AccountController(UserManager<ApplicationUser> manager)
  {
    UserManager = manager;
  }

  public AccountController() :
    this(new UserManager<ApplicationUser>(
      new UserStore<ApplicationUser>(new ApplicationDbContext())))
  {
  }

  ...
}
```

The ASP.NET native membership provider is a standalone component that exposes a contracted interface. The ASP.NET runtime, which orchestrates the authentication process, is aware of the membership interface and can invoke whatever component is configured as the membership provider of the application. ASP.NET came with a default membership provider based on a new given database schema. However, you could easily write your own membership provider to basically target a different database—typically, an existing database of users.

Does that sound like a great chunk of architecture? In the beginning, nearly everybody thought so. In the long run, though, quite a few people who repeatedly tried to build a custom membership provider started complaining about the verbosity of the interface. Actually, the membership provider comes in the form of an inheritable base class, MembershipProvider, which includes more than 30 members marked as abstract. This means that for any new membership provider you wanted to create, there were at least 30 members to override. Worse yet, you didn't really need many of them most of the time. A simpler membership architecture was needed.

Introducing the Simple Membership Provider

To save you from the burden of creating a custom membership layer completely from scratch, Microsoft introduced with Visual Studio 2010 SP1 another option: the simple membership API. Originally available in WebMatrix and Web Pages, the simple membership

Built for Today... Ready for Tomorrow

Touch-Enabled .NET Controls by DevExpress.

Create solutions your customers expect today and leverage your existing skillsets to build next generation applications for tomorrow. DevExpress Controls are built to emulate the UI experiences at the heart of Microsoft Office AND to enable touch-first experiences for next-generation devices like iPad and Surface.

Your next great app starts here.
DevExpress.com/Touch

DevExpress®

API has become quite a popular way of managing authentication, especially in ASP.NET MVC. In particular, the Internet application template in ASP.NET MVC 4 uses the simple membership API to support user management and authentication.

Looking under the hood of the API, it turns out that it's just a wrapper on top of the classic ASP.NET membership API and its SQL Server-based data stores. Simple membership lets you work with any data store you have and requires only that you indicate which columns in the table serve as the user name and user ID.

> The Internet application template in ASP.NET MVC 4 uses the simple membership API to support user management and authentication.

The major difference from the classic membership API is a significantly shorter list of parameters for any methods. In addition, you get a lot more freedom as far as the schema of the membership storage is concerned. As an example of the simplified API, consider what it takes to create a new user:

```
WebSecurity.CreateUserAndAccount(username, password,
  new { FirstName = fname, LastName = lname, Email = email });
```

You do most of the membership chores via the WebSecurity class. In ASP.NET MVC 4, however, the WebSecurity class expects to work with an extended membership provider, not a classic membership provider. The additional capabilities in an extended membership provider are related to dealing with OAuth accounts. As a result, in ASP.NET MVC 4, you have two parallel routes for membership implementation: classic membership API using the MembershipProvider class and simple membership API using the ExtendedMembershipProvider class. The two APIs are incompatible.

Before the arrival of Visual Studio 2013 and ASP.NET MVC 5, ASP.NET already offered quite a few ways to handle user authentication. With forms authentication, you could rely on classic membership, the simple membership API as defined in Web Pages and a variety of custom membership systems. Consider the common position among ASP.NET experts was that complex real-world applications require their own membership provider. More often than not, the main reason for having a custom membership system was to circumvent structural differences between the required database format and the format of the existing database of user credentials, which might have been in use for years.

Clearly, this wasn't a situation that could last forever. The community of developers demanded with loud voices a unified system for membership that's simple to use, narrowly focused and usable in the same way from within any flavor of ASP.NET. This idea weds together well with the One ASP.NET approach pushed by Visual Studio 2013.

Figure 2 **Definition of the Default User Class in ASP.NET Identity**

```
namespace Microsoft.AspNet.Identity.EntityFramework
{
  public class IdentityUser : IUser
  {
    public string Id { get; }
    public string UserName { get; set; }
    public string PasswordHash { get; set; }
    public string SecurityStamp { get; set; }
    public ICollection<IdentityUserRole> Roles { get; private set; }
    public ICollection<IdentityUserClaim> Claims { get; private set; }
    public ICollection<IdentityUserLogin> Logins { get; private set; }
  }
}
```

One Identity Framework

The purpose of authentication is getting the identity associated with the current user. The identity is retrieved and the provided credentials are compared to records stored in a database. Subsequently, an identity system such as ASP.NET Identity is based on two primary blocks: the authentication manager and the store manager. In the ASP.NET Identity framework, the authentication manager takes the form of the UserManager<TUser> class. This class basically provides a façade for signing users in and out. The store manager is an instance of the UserStore<TUser> class. **Figure 1** shows the skeleton of an ASP.NET MVC account controller class that's based on ASP.NET Identity.

The controller holds a reference to the authentication identity manager, UserManager. This instance of UserManager is injected into the controller. You can use either an Inversion of Control (IoC) framework or the poor man's alternative, the dependency injection (DI) pattern, which uses two controllers, one of which gets a default value (see **Figure 1**).

The identity store, in turn, is injected into the authentication identity manager, where it's used to verify credentials. The identity store takes the form of the UserStore<TUser> class. This class results from the composition of multiple types:

```
public class UserStore<TUser> :
  IUserLoginStore<TUser>,
  IUserClaimStore<TUser>,
  IUserRoleStore<TUser>,
  IUserPasswordStore<TUser>,
  IUserSecurityStampStore<TUser>,
  IUserStore<TUser>,
  IDisposable where TUser : IdentityUser
  {
  }
```

All interfaces implemented by UserStore<TUser> are basic repositories for optional user-related data such as passwords, roles, claims and, of course, user data. The identity store needs to know about the actual data source, though. As shown in **Figure 1**, the data source is injected in the UserStore class through the constructor.

Storage of users' data is managed through the Entity Framework Code First approach. This means you don't strictly need to create a physical database to store your users' credentials; you can, instead, define a User class and have the underlying framework create the most appropriate database to store such records.

The ApplicationDbContext class wraps up the Entity Framework context to save users' data. Here's a possible definition for the ApplicationDbContext class:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
}
```

Basically, the database context of ASP.NET Identity handles the persistence of a given user type. The user type must implement the

Figure 3 **Finalizing the Authentication Process through an External Endpoint**

```
public async Task<ActionResult> ExternalLoginCallback(
  string loginProvider, string returnUrl)
{
  ClaimsIdentity id = await UserManager
  .Authentication
  .GetExternalIdentityAsync(AuthenticationManager);

  var result = await UserManager
    .Authentication
    .SignInExternalIdentityAsync(
      AuthenticationManager, id);
  if (result.Success)
    return RedirectToLocal(returnUrl);
  else if (User.Identity.IsAuthenticated)
  {
    result = await UserManager
      .Authentication
      .LinkExternalIdentityAsync(
        id, User.Identity.GetUserId());
    if (result.Success)
      return RedirectToLocal(returnUrl);
    else
      return View("ExternalLoginFailure");
  }
}
```

IUser interface or just inherit from IdentityUser. **Figure 2** presents the source code of the default IdentityUser class.

Here's an example of a realistic custom user class you might want to use in your applications:

```
public class ApplicationUser : IdentityUser
{
  public DateTime Birthdate { get; set; }
}
```

The use of Entity Framework Code First is a great move here as it makes the structure of the database a secondary point. You still need one, but to create it, you can use code based on classes. In addition, you can use Entity Framework Code First migration tools to modify a previously created database as you make changes to the class behind it. (For more information on this, see the "Code First Migrations" article in the MSDN Data Developer Center at bit.ly/Th92qf.)

Authenticating Users

ASP.NET Identity is based on the newest Open Web Interface for .NET (OWIN) authentication middleware. This means the typical steps of authentication (for example, creating and checking cookies) can be carried out through the abstract OWIN interfaces and not directly via ASP.NET/IIS interfaces. Support for OWIN requires the account controller to have another handy property, like this:

```
private IAuthenticationManager AuthenticationManager
{
  get {
  return HttpContext.GetOwinContext().Authentication;
  }
}
```

The IAuthenticationManager interface is defined in the Microsoft.Owin.Security namespace. This property is important because it needs to be injected into any operation that involves authentication-related steps. Here's a typical login method:

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
  var identity = await UserManager.CreateIdentityAsync(user,
    DefaultAuthenticationTypes.ApplicationCookie);
  AuthenticationManager.SignIn(new AuthenticationProperties() {
    IsPersistent = isPersistent }, identity);
}
```

The method SignInAsync checks the specified user name and password against the store associated with the authentication manager. To register a user and add the user to the membership database, you use code like this:

```
var user = new ApplicationUser() { UserName = model.UserName };
var result = await UserManager.CreateAsync(user, model.Password);
if (result.Succeeded)
{
  await SignInAsync(user, isPersistent: false);
  return RedirectToAction("Index", "Home");
}
```

All in all, ASP.NET Identity provides a unified API for tasks related to authentication. For example, it unifies the code required to authenticate against a proprietary database or a social network OAuth-based endpoint. **Figure 3** shows a fragment of the code you need to authenticate users against an external login engine. The code in **Figure 3** gets called once the OAuth authentication (for example, against Facebook) has been completed successfully.

> ASP.NET Identity is bound to Visual Studio 2013, but it's also expected to have an autonomous life of its own when it comes to future builds and releases.

The Bottom Line

As I see things, ASP.NET Identity is an overdue solution that should have come years ago. The key issue concerning ASP.NET Identity right now is the development team is trying to come up with a programming interface that's generic and testable enough to last for a long time—or at least until something newer and better shows up in the industry.

For the foreseeable future, ASP.NET Identity promises to be as good as old-fashioned membership was perceived to be a decade ago. Personally, I like the expressiveness of the API and the attempt to fuse together different forms of authentication—built-in and OAuth-based, for example. Another great plus is the integration with OWIN, which makes it somewhat independent from a specific runtime such as IIS/ASP.NET.

ASP.NET Identity is bound to Visual Studio 2013, but it's also expected to have an autonomous life of its own when it comes to future builds and releases. I've just scratched the surface of the new identity API here. Stay tuned for newer builds and releases! ■

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

Thanks to the following technical expert for reviewing this article: *Pranav Rastogi (Microsoft)*

ASPOSE.TOTAL

Every Aspose component combined in *ONE* powerful suite!

Powerful File Format APIs

- **Aspose.Words**
  DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.
- **Aspose.Cells**
  XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.
- **Aspose.BarCode**
  JPG, PNG, BMP, GIF, TIF, WMF, ICON & other image formats.
- **Aspose.Pdf**
  PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.
- **Aspose.Email**
  MSG, EML, PST, EMLX & other formats.
- **Aspose.Slides**
  PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.
- **Aspose.Diagram**
  VSD, VSDX, VSS, VST, VSX & other formats.

... and many others!

Aspose.Total for .NET
Aspose.Total for Java
Aspose.Total for Cloud
Aspose.Total for Android

Get your FREE evaluation copy at www.aspose.com

.NET Java Cloud Android

BRUNO TERKALY AND
RICARDO VILLALOBOS

The Windows Azure Service Bus and the Internet of Things, Part 2

In our last column (msdn.microsoft.com/magazine/dn574801), we discussed the current technology landscape for machine-to-machine (M2M) computing, which refers to technologies that interconnect devices, usually for industrial instrumentation, in the form of sensors or meters. The proliferation of affordable and easy-to-program tiny computers has expanded this concept into what's called the Internet-of-Things (IoT), opening the door to scenarios where even ordinary home appliances can be controlled or used as sources of information to generate events. This way, it isn't difficult to send alerts when it's time to replenish the fridge, automatically close the window blinds as night falls or set the thermostat based on the family habits.

We also made the case for using the Windows Azure Service Bus for device connectivity, as an alternative to using a VPN, when trying to solve the addressability, security, and performance concerns associated with deploying a large number of sensors or meters. This is becoming increasingly relevant considering that, according to the latest BI Intelligence report from *Business Insider*, there will be more than 9 billion connections directly related to the IoT by the year 2018 (read.bi/18L5cg8).

Using a designated Service Bus queue or topic for a device provides an elegant way to incorporate resiliency and occasional connectivity for IoT applications. In this article, we'll walk through a hands-on Windows Azure implementation that illustrates these concepts, designing a Service Bus blueprint with device queues, deploying a listening worker role in Cloud Services, and programming an Arduino device that executes commands sent remotely by mobile clients, as shown in **Figure 1**.

If you look at the diagram, the Windows Azure Service Bus component becomes the centerpiece of the design, providing the authentication, message distribution and scalability to support the multiple devices that will be sending data or receiving remote commands. The Service Bus is available in all Microsoft datacenters that offer Windows Azure services, and it's backed up by a highly redundant storage infrastructure. Also, like all other Windows Azure components, it offers an open and easy-to-understand REST interface, along with multiple SDKs (Microsoft .NET Framework, Java, PHP, Ruby, among others) built on top of it.

Code download available at msdn.microsoft.com/magazine/msdnmag0314.

In our proposed architecture, devices "talk" to a .NET application running on Windows Azure Cloud Services, which acts as a gateway to the Service Bus in order to simplify the communication process with its assigned queue. This approach fully enables any of the four IoT communication patterns described in our previous column: Telemetry, Inquiry, Command and Notification. Here, we'll implement a scenario in which a mobile device sends a command to another device in order to execute an action—in this case, turn an LED on or off. One of the benefits of this solution is that if the device is temporarily offline, it can pick up the commands whenever it reconnects to the Internet. You can also set up an expiration time in a message to avoid the execution of a task at an inconvenient moment or schedule messages to be sent at a specific time in the future.

For this example, we'll use the well-known, well-documented Arduino device, as described in our previous column. For the mobile client portion of the proof-of-concept, we'll create a Windows Phone application.

Here's our simple scenario:

1. When the Arduino device is started, it sends an identification signal to the gateway application running on Windows Azure Cloud Services. The gateway creates a Service Bus

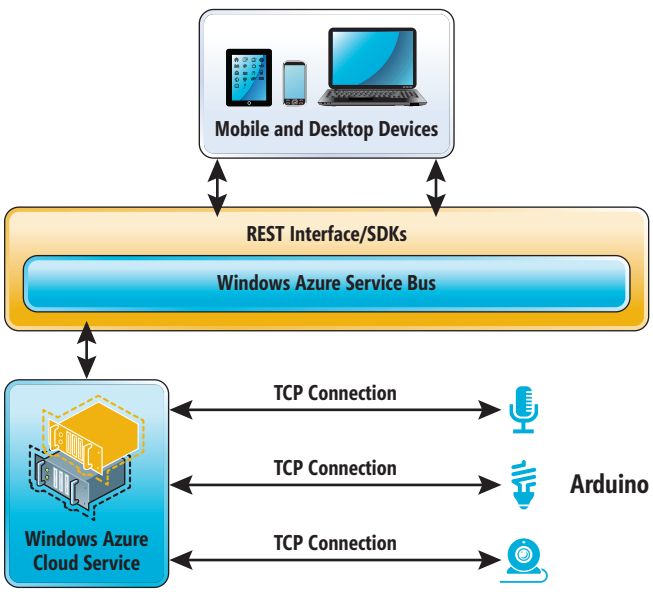


Figure 1 **An Internet-of-Things Architecture Using the Windows Azure Service Bus**

queue for the device in case it doesn't exist, and establishes a TCP connection, ready to send commands.

2. A Windows Phone application sends a command to the Windows Azure Service Bus queue assigned to the device.
3. The message remains in the queue until the gateway application picks it up and sends the command to the Arduino device via the established TCP connection.
4. The Arduino device turns the LED on or off based on the command.

Let's look at the steps to make this happen, one by one.

Step 1: Create the Windows Azure Service Bus Namespace Using your Windows Azure credentials (you can request a trial account at bit.ly/1atsgSa), log in to the Web portal and click on the SERVICE BUS section (see **Figure 2**). Select the CREATE option, and enter a name for your namespace. Then, click on CONNECTION INFORMATION and copy the text in the Connection String box, which you'll need later.

Step 2: Create the Gateway Application and Deploy to Windows Azure Cloud Services Code for the gateway application, which retrieves messages from the Service Bus queue and relays the commands to the Arduino device, is included with the code download (available at msdn.microsoft.com/magazine/msdnmag0314). It's based on the work of Clemens Vaster, who kindly contributed his guidance and expertise to this article. His original project can be found at bit.ly/LOuKOv.

Before we dive into this code, be sure you have Visual Studio 2013 installed, along with version 2.2 of the Windows Azure SDK for .NET (bit.ly/JYXx5n). The solution includes three different projects:

- **ArduinoListener**—contains the main WorkerRole code.
- **ConsoleListener**—the console version of the ArduinoListener, for local testing.
- **MSDNArduinoListener**—the Windows Azure deployment project for ArduinoListener.

If you inspect the ServiceConfiguration.cscfg files (for both cloud and local deployment) for the MSDNArduinoListener project, you'll see a setting that stores the connection string for the Service Bus. Replace its value with the one obtained in Step 1. The rest is already configured for the solution to work, including the definition of port 10100 for receiving connections from the devices. Next, open the WorkerRole.cs file in the ArduinoListener project, where the main code is located.

There are four main sections to analyze.

First, a TcpListener is created, and connections from devices are accepted:

```
var deviceServer = new TcpListener(deviceEP);
deviceServer.Start(10);
try
{
  do
  {
    TcpClient connection = await deviceServer.AcceptTcpClientAsync();
    if (connection != null)
    {
      ...
```

Once a connection with the device has been established, a NetworkStream is defined and set to listening mode. The readBuffer variable will contain the identifier value sent by each Arduino device:

```
NetworkStream deviceConnectionStream = connection.GetStream();
var readBuffer = new byte[64];
if (await deviceConnectionStream.ReadAsync(readBuffer, 0, 4) == 4)
{
  int deviceId = IPAddress.NetworkToHostOrder(BitConverter.ToInt32(readBuffer, 0));
  ...
```

Next, a queue is created based on the deviceId value (in case it doesn't exist), and a message receiver object is defined (see **Figure 3**). Then, the device queue receiver is set to asynchronous mode to pull messages (commands from the queue). This queue will store commands sent by mobile devices, such as a Windows Phone.

When a message is received in the queue, its content is inspected and if it matches the "ON" or "OFF" commands, the information is written to the connection stream established with the device (see **Figure 4**).

Notice that the message isn't removed from the queue (message.CompleteAsync) unless the writing operation to the device connection stream is successful. Also, in order to keep the connection alive, the device is expected to send a ping heartbeat. For this proof of concept, we aren't expecting confirmation from the device when it receives the message. In a production system, however, this would be required to comply with the "command" pattern.

Step 3: Deploy the ArduinoListener Windows Azure Project to Cloud Services Deploying the ArduinoListener to Windows Azure is extremely simple. In Visual Studio 2013, right-click on the MSDNArduinoListener project and select the Publish option. You'll find

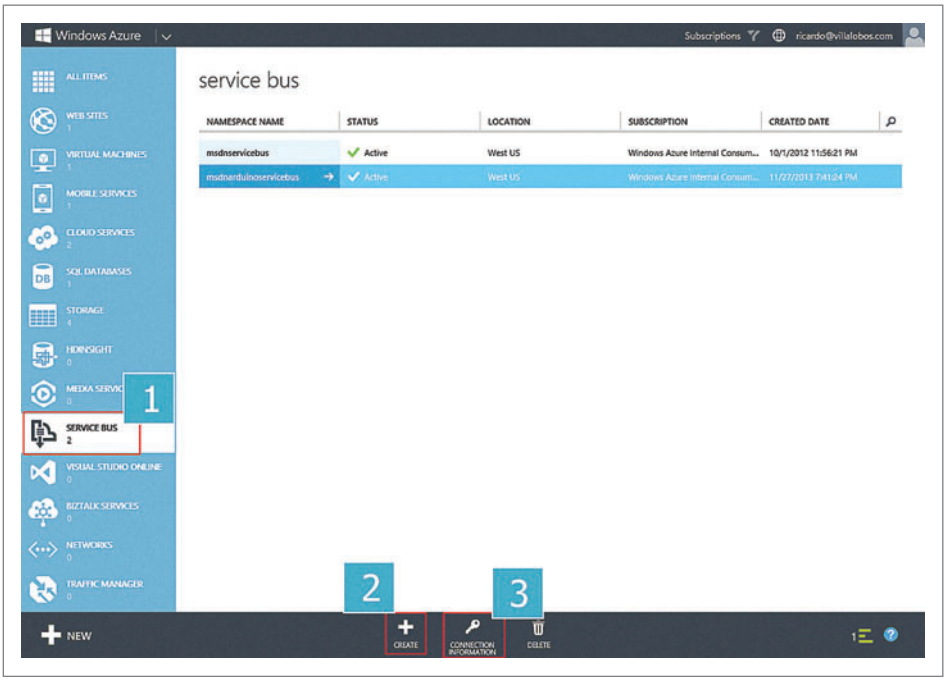


Figure 2 **Creating the Windows Azure Service Bus Namespace**

1&1 NEW eCOM

SELL MORE WITH A PROFESSIONAL DESIGN.

- Whether beginner or professional, create your store online in a few easy steps
- Choose from over a hundred high-quality designs and templates between industries
- Store links easily with an existing domain or your new included domain (free)*
- Whether PC, tablet or smartphone your shop will be displayed optimally on all devices

MEMBER OF united internet

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

* Offer valid for a limited time only. Complete packages come with a 30 day money back guarantee and no minimum contract term. The $7.99 per month price reflects a 12 month pre-payment option for the 1&1 Online Store Starter package. After 12 months, regular price of $9.99 per month applies. Some features listed are only available with package upgrade.

MERCE

1&1 ONLINE STORE COMPLETE PACKAGES
starting at
$7.99 month*

Try now! 30 day money back guarantee.

START SELLING NOW!

MORE POSSIBILITIES. MORE SUCCESS.

- Your shop can grow with your business
- Target customers with special promotions and cross selling
- Product rating capability: Build trust by letting customers share feedback
- Sell internationally: Wide selection of languages, currencies and payment options

FIND CUSTOMERS. KEEP CUSTOMERS.

- Search engine optimization (SEO): rank higher on Google and other search engines
- Easy synchronization with Amazon, Ebay, and more
- Easily create your own Facebook Store
- Create customer loyalty by providing free newsletters and coupons

MAXIMUM RELIABILITY. PROFESSIONAL SUPPORT.

- Convenient shipping processing via UPS, FedEx, etc.
- Reliability through geo-redundant operation in two separate 1&1 Data Centers
- 24/7 expert customer service by experienced eCommerce professionals

1 TEST 30 DAY MONEY BACK GUARANTEE

1 MONTH FLEXIBLE PAYMENT OPTIONS

1 CALL SPEAK WITH AN EXPERT 24/7

1&1®

Call **1 (877) 461-2631**

1and1.com

Figure 3 **Creating a Queue**

```
var namespaceManager = NamespaceManager.
CreateFromConnectionString(RoleEnvironment.
GetConfigurationSettingValue("serviceBusConnectionString"));
if (!namespaceManager.QueueExists(string.Format("dev{0:X8}", deviceId)))
{
  namespaceManager.CreateQueue(string.Format("dev{0:X8}", deviceId));
}
var deviceQueueReceiver = messagingFactory.CreateMessageReceiver(
  string.Format("dev{0:X8}", deviceId), ReceiveMode.PeekLock);
do
{
  BrokeredMessage message = null;
  message = await deviceQueueReceiver.ReceiveAsync();
  ...
```

specific instructions for the Publish Windows Azure Application Wizard at bit.ly/1iP9g2p. After completing the wizard, you end up with a cloud service located at xyz.cloudapp.net. Record this name, as you'll need it when you create the Arduino client in the next step.

Step 4: Program the Arduino Device to Talk to the Gateway (Listener) Arduino devices offer a rich interface for performing network operations using a simple Web client object. For our prototype, we decided to use the Arduino Uno R3 model (bit.ly/18ZlcM8), along with its corresponding Ethernet shield (bit.ly/1do6eRD). To install, interact and program Arduino devices using Windows, follow the guide at bit.ly/1dNBi9R. You'll end up with an easy-to-use IDE (called the Arduino application), where you can write programs (called sketches) using JavaScript, as shown in **Figure 5**.

Figure 6 shows the sketch for interacting with the Arduino Listener created in Step 3, and now deployed in Windows Azure.

Sketches for the Arduino have two main sections: setup and loop. Instructions in the setup section are executed once, and this is where variables are initialized and connections established. In our example, the Ethernet client and related values are defined, a serial connection (for debugging purposes) is established, and the pin where the LED is connected is initialized as an output port.

Code in the loop section is executed constantly, and it includes two main blocks based on the status of the TCP connection between the Arduino device and the listener running in Windows Azure Cloud Services: connected or disconnected. When the connection is established for the first time, a stopWatch object is started to keep track of the time elapsed for the connection. Also, the device identifier is sent to the listener, to be used as the name of the queue where messages and commands will be stored.

The code block that handles the Arduino behavior after the connection has been established keeps track of the time elapsed since the connection was created, pinging the listener every 200,000 ms, to keep the connection alive when no commands are received. This code also tries to read data from the listener, putting the data into the buf array when it arrives. If a value of "1" is detected, the LED is turned on, if the value is "2," the LED is turned off. The stopwatch object is reset after each command.

Figure 4 **Writing to the Connection Stream**

```
if (message != null)
{
  Stream stream = message.GetBody<Stream>();
  StreamReader reader = new StreamReader(stream);
  string command = reader.ReadToEnd();

  if (command != null)
  {
    switch (command.ToUpperInvariant())
    {
      case "ON":
        await deviceConnectionStream.WriteAsync(OnFrame, 0, OnFrame.Length);
        await message.CompleteAsync();
        break;
      case "OFF":
        await deviceConnectionStream.WriteAsync(OffFrame, 0, OffFrame.Length);
        await message.CompleteAsync();
        break;
    }
  }
}
```

Once the sketch has been uploaded to the device, the code runs on the Arduino controller in an infinite loop, trying to connect to a cloud service. When connected, it forwards the device id so the cloud service knows to which device it's talking. Then the code begins to read input from the cloud service, telling the device whether to turn on or off the LED light (in this case, it's connected to digital port 8 of the device).

Step 5: Creating a Windows Phone Client to Send to Device Queue Interacting with the device is as simple as sending messages to the device queue. As we mentioned at the beginning of the article, the Windows Azure Service Bus provides a REST interface that lets you interact with it from multiple programming languages. Because there's no official SDK for Windows Phone developers, we used one of the examples from the Windows Phone community, which shows how to authenticate and interact with the Service Bus using HTTP requests and the WebClient object. The source code is also included with the code download, in the Visual Studio 2013 project called MSDNArduinoClient. **Figure 7** shows the client's main screen, from which you send commands to the Arduino device.

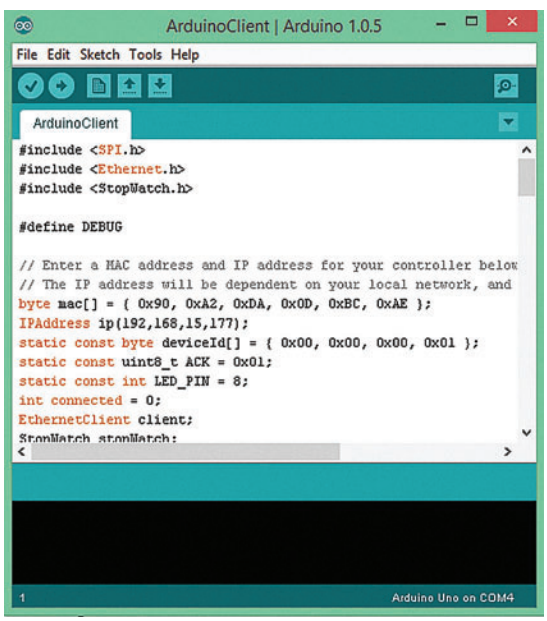


Figure 5 **The Arduino Application**

Creating similar clients for other mobile devices (including iOS and Android) wouldn't be difficult, as most of them provide libraries to generate REST commands using HTTP request clients. Moreover, it's possible to directly interact with the Windows Azure Service Bus using traditional languages such as Java, PHP or Ruby, which simplifies this process. These SDKs are published under an open source license, and can be found at github.com/WindowsAzure.

Figure 6 **The Arduino Device Code**

```
#include <SPI.h>
#include <Ethernet.h>
#include <StopWatch.h>
// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network,
// and it's optional if DHCP is enabled.
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0D, 0xBC, 0xAE };
static const byte deviceId[] = { 0x00, 0x00, 0x00, 0x01 };
static const uint8_t ACK = 0x01;
static const int LED_PIN = 8;
int connected = 0;
EthernetClient client;
StopWatch stopWatch;
long pingInterval = 200000;
void setup()
{
  Serial.begin(9600);
  Serial.println("Initialized");
  Ethernet.begin(mac);
  pinMode(LED_PIN, OUTPUT);
}
void turnLedOn()
{
  digitalWrite(LED_PIN, HIGH);
}
void turnLedOff()
{
  digitalWrite(LED_PIN, LOW);
}
void loop()
{
  if ( connected == 0)
  {
    Serial.println("Trying to connect");
    char* host = "xyz.cloudapp.net";
    client.setTimeout(10000);
    connected = client.connect(host, 10100);
    if (connected)
    {
      Serial.println(
        "Connected to port, writing deviceId and waiting for commands...");
      client.write(deviceId, sizeof(deviceId));
      stopWatch.start();
    }
    else
    {
      Serial.println("Connection unsuccessful");
      client.stop();
      stopWatch.reset();
    }
  }
  if (connected == 1)
  {
    if (stopWatch.elapsed() > pingInterval)
    {
      Serial.println("Pinging Server to keep connection alive...");
      client.write(deviceId, sizeof(deviceId));
      stopWatch.reset();
      stopWatch.start();
    }
    byte buf[16];
    int readResult = client.read(buf, 1);
    if (readResult == 0)
    {
      Serial.println("Can't find listener, disconnecting...");
      connected = 0;
      stopWatch.reset();
    }
    else if (readResult == 1)
    {
      Serial.println("Data acquired, processing...");
      switch ( buf[0] )
      {
        case 1:
          Serial.println("Command to turn led on received...");
          turnLedOn();
          break;
        case 2:
          Serial.println("Command to turn led off received...");
          turnLedOff();
          break;
      }
      stopWatch.reset();
      stopWatch.start();
    }
  }
}
```

Figure 7 **The Windows Phone Client Interface**

Wrapping Up

Building an Internet-of-Things architecture using the Windows Azure Service Bus to manage devices and services connections provides an easy way to secure, scale and address clients individually without incurring costly VPN solutions, with the benefit of efficiently handling occasionally disconnected scenarios. Queues act as dedicated mailboxes where messages between devices and services are exchanged, supporting the different communication use cases and patterns commonly found in the field. Windows Azure provides a reliable, geo-distributed and robust infrastructure for deploying the services required with a high volume of interconnected sensors and meters—a trend that will continue to grow in the years ahead. ■

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

Ricardo Villalobos *is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in multiple industries. 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 DPE Globally Engaged Partners team for Microsoft, helping companies worldwide to implement solutions in Windows Azure. You can read his blog at blog.ricardovillalobos.com.*

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

Thanks to the following Microsoft technical experts for reviewing this article: *Abhishek Lal and Clemens Vasters*

Patterns for Asynchronous MVVM Applications: Data Binding

Stephen Cleary

Asynchronous code using the async and await keywords is transforming the way programs are written, and with good reason. Although async and await can be useful for server software, most of the current focus is on applications that have a UI. For such applications, these keywords can yield a more responsive UI. However, it's not immediately obvious how to use async and await with established patterns such as Model-View-ViewModel (MVVM). This article is the first in a short series that will consider patterns for combining async and await with MVVM.

To be clear, my first article on async, "Best Practices in Asynchronous Programming" (msdn.microsoft.com/magazine/jj991977), was relevant to all applications that use async/await, both client and server. This new series builds on the best practices in that article and introduces patterns specifically for client-side MVVM applications. These patterns are just patterns, however, and may not necessarily be the best solutions for a specific scenario. If you find a better way, let me know!

This article discusses:

- Combining asynchronous programming with the MVVM pattern
- Developing an asynchronous data-bound property
- Common mistakes with ViewModels
- An approach that's data-binding friendly

Technologies discussed:

Asynchronous Programming, MVVM

As of this writing, the async and await keywords are supported on a wide number of MVVM platforms: desktop (Windows Presentation Foundation [WPF] on the Microsoft .NET Framework 4 and higher), iOS/Android (Xamarin), Windows Store (Windows 8 and higher), Windows Phone (version 7.1 and higher), Silverlight (version 4 and higher), as well as Portable Class Libraries (PCLs) targeting any mix of these platforms (such as MvvmCross). The time is now ripe for "async MVVM" patterns to develop.

I'm assuming you're somewhat familiar with async and await and quite familiar with MVVM. If that's not the case, there are a number of helpful introductory materials available online. My blog (bit.ly/19IkogW) includes an async/await intro that lists additional resources at the end, and the MSDN documentation on async is quite good (search for "Task-based Asynchronous Programming"). For more information on MVVM, I recommend pretty much anything written by Josh Smith.

A Simple Application

In this article, I'm going to build an incredibly simple application, as **Figure 1** shows. When the application loads, it starts an HTTP request and counts the number of bytes returned. The HTTP request may complete successfully or with an exception, and the application will update using data binding. The application is fully responsive at all times.

First, though, I want to mention that I follow the MVVM pattern rather loosely in my own projects, sometimes using a proper domain Model, but more often using a set of services and data

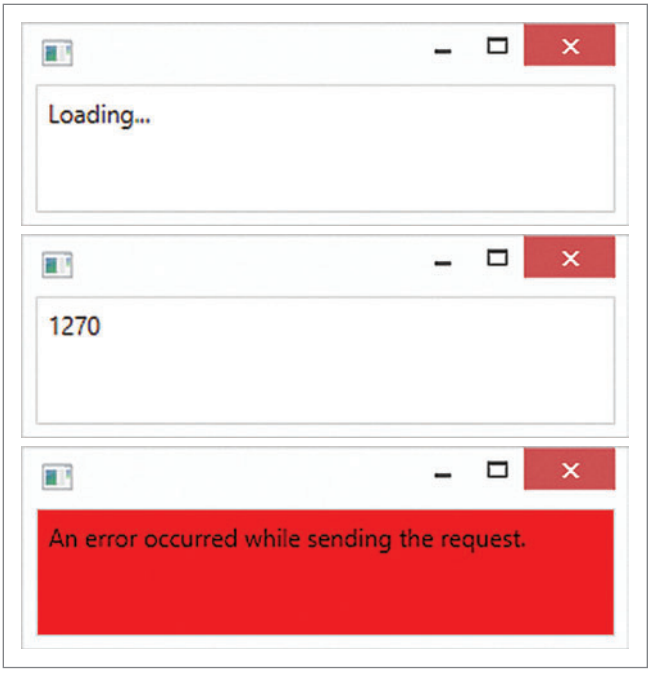


Figure 1 **The Sample Application**

transfer objects (essentially a data access layer) instead of an actual Model. I'm also rather pragmatic when it comes to the View; I don't shy away from a few lines of codebehind if the alternative is dozens of lines of code in supporting classes and XAML. So, when I talk about MVVM, understand that I'm not using any particular strict definition of the term.

One of the first things you have to consider when introducing async and await to the MVVM pattern is identifying which parts of your solution need the UI threading context. Windows platforms are serious about UI components being accessed only from the UI thread that owns them. Obviously, the view is entirely tied to the UI context. I also take the stand in my applications that anything linked to the view via data binding is tied to the UI context. Recent versions of WPF have loosened this restriction, allowing some sharing of data between the UI thread and background threads (for example, BindingOperations.EnableCollectionSynchronization). However, support for cross-thread data binding isn't guaranteed on every MVVM platform (WPF, iOS/Android/Windows Phone, Windows Store), so in my own projects I just treat anything data-bound to the UI as having UI-thread affinity.

As a result, I always treat my ViewModels as though they're tied to the UI context. In my applications, the ViewModel is more closely related to the View than the Model—and the ViewModel layer is essentially an API for the entire application. The View literally provides just the shell of UI elements in which the actual application exists. The ViewModel layer is conceptually a testable UI, complete with a UI thread affinity. If your Model is an actual domain model (not a data access layer) and there's data binding between the Model and ViewModel, then the Model itself also has UI-thread affinity. Once you've identified which layers have UI affinity, you should be able to draw a mental line between the "UI-affine code" (View and ViewModel, and possibly the Model) and the "UI-agnostic code" (probably the Model and definitely all other layers, such as services and data access).

Furthermore, all code outside the View layer (that is, the ViewModel and Model layers, services, and so on) should not depend on any type tied to a specific UI platform. Any direct use of Dispatcher (WPF/Xamarin/Windows Phone/Silverlight), CoreDispatcher (Windows Store), or ISynchronizeInvoke (Windows Forms) is a bad idea. (SynchronizationContext is marginally better, but barely.) For example, there's a lot of code on the Internet that does some asynchronous work and then uses Dispatcher to update the UI; a more portable and less cumbersome solution is to use await for asynchronous work and update the UI without using Dispatcher.

ViewModels are the most interesting layer because they have UI affinity but don't depend on a specific UI context. In this series, I'll combine async and MVVM in ways that avoid specific UI types while also following async best practices; this first article focuses on asynchronous data binding.

Asynchronous Data-Bound Properties

The term "asynchronous property" is actually an oxymoron. Property getters should execute immediately and retrieve current values, not kick off background operations. This is likely one of the reasons the async keyword can't be used on a property getter. If you find your design asking for an asynchronous property, consider some alternatives first. In particular, should the property actually be a method (or a command)? If the property getter needs to kick off a new asynchronous operation each time it's accessed, that's not a property at all. Asynchronous methods are straightforward, and I'll cover asynchronous commands in another article.

In this article, I'm going to develop an asynchronous data-bound property; that is, a data-bound property that I update with the results of an async operation. One common scenario is when a ViewModel needs to retrieve data from some external source.

As I explained earlier, for my sample application, I'm going to define a service that counts the bytes in a Web page. To illustrate the responsiveness aspect of async/await, this service will also delay a few seconds. I'll cover more realistic asynchronous services in a later article; for now, the "service" is just the single method shown in **Figure 2**.

Figure 2 **MyStaticService.cs**

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public static class MyStaticService
{
  public static async Task<int> CountBytesInUrlAsync(string url)
  {
    // Artificial delay to show responsiveness.
    await Task.Delay(TimeSpan.FromSeconds(3)).ConfigureAwait(false);

    // Download the actual data and count it.
    using (var client = new HttpClient())
    {
      var data = await client.GetByteArrayAsync(url).ConfigureAwait(false);
      return data.Length;
    }
  }
}
```

Note that this is considered a service, so it's UI-agnostic. Because the service is UI-agnostic, it uses ConfigureAwait(false) every time it does an await (as discussed in my other article, "Best Practices in Asynchronous Programming").

Let's add a simple View and ViewModel that starts an HTTP request on startup. The example code uses WPF windows with the Views creating their ViewModels on construction. This is just for simplicity; the async principles and patterns discussed in this series of articles apply across all MVVM platforms, frameworks and libraries. The View for now will consist of a single main window with a single label. The XAML for the main View just binds to the UrlByteCount member:

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Label Content="{Binding UrlByteCount}"/>
  </Grid>
</Window>
```

The codebehind for the main window creates the ViewModel:

```
public partial class MainWindow
{
  public MainWindow()
  {
    DataContext = new BadMainViewModelA();
    InitializeComponent();
  }
}
```

Common Mistakes

You might notice the ViewModel type is called BadMainViewModelA. This is because I'm going to first look at a couple of common mistakes relating to ViewModels. One common mistake is to synchronously block on the operation, like so:

```
public class BadMainViewModelA
{
  public BadMainViewModelA()
  {
    // BAD CODE!!!
    UrlByteCount =
      MyStaticService.CountBytesInUrlAsync("http://www.example.com").Result;
  }

  public int UrlByteCount { get; private set; }
}
```

This is a violation of the async guideline "async all the way," but sometimes developers try this if they feel they're out of options. If you execute that code, you'll see it works, to a certain extent. Code that uses Task.Wait or Task<T>.Result instead of await is synchronously blocking on that operation.

There are a few problems with synchronous blocking. The most obvious is the code is now taking an asynchronous operation and blocking on it; by doing so, it loses all the benefits of asynchronicity. If you execute the current code, you'll see the application does nothing for a few seconds, and then the UI window springs fully formed into view with its results already populated. The problem is the application is unresponsive, which is unacceptable for many modern applications. The example code has a deliberate delay to emphasize that unresponsiveness; in a real-world application, this problem might go unnoticed during development and show up only in "unusual" client scenarios (such as loss of network connectivity).

Another problem with synchronous blocking is more subtle: The code is more brittle. My example service uses ConfigureAwait(false) properly, just as a service should. However, this is easy to forget, especially if you (or your coworkers) don't regularly use async. Consider what could happen over time as the service code is maintained. A maintenance developer might forget a ConfigureAwait, and at that point the blocking of the UI thread would become a deadlock of the UI thread. (This is described in more detail in my previous article on async best practices.)

OK, so you should use "async all the way." However, many developers proceed to the second faulty approach, illustrated in **Figure 3**.

Again, if you execute this code, you'll find that it works. The UI now shows immediately, with "0" in the label for a few seconds before it's updated with the correct value. The UI is responsive, and everything seems fine. However, the problem in this case is handling errors. With an async void method, any errors raised by the asynchronous operation will crash the application by default. This is another situation that's easy to miss during development and shows up only in "weird" conditions on client devices. Even changing the code in **Figure 3** from async void to async Task barely improves the application; all errors would be silently ignored, leaving the user wondering what happened. Neither method of handling errors is appropriate. And though it's possible to deal with this by catching exceptions from the asynchronous operation and updating other data-bound properties, that would result in a lot of tedious code.

A Better Approach

Ideally, what I really want is a type just like Task<T> with properties for getting results or error details. Unfortunately, Task<T> is not data-binding friendly for two reasons: it doesn't implement INotifyPropertyChanged and its Result property is blocking. However, you can define a "Task watcher" of sorts, such as the type in **Figure 4**.

Figure 3 **BadMainViewModelB.cs**

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

public sealed class BadMainViewModelB : INotifyPropertyChanged
{
  public BadMainViewModelB()
  {
    Initialize();
  }

  // BAD CODE!!!
  private async void Initialize()
  {
    UrlByteCount = await MyStaticService.CountBytesInUrlAsync(
      "http://www.example.com");
  }

  private int _urlByteCount;
  public int UrlByteCount
  {
    get { return _urlByteCount; }
    private set { _urlByteCount = value; OnPropertyChanged(); }
  }

  public event PropertyChangedEventHandler PropertyChanged;
  private void OnPropertyChanged([CallerMemberName] string propertyName = null)
  {
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

Figure 4 **NotifyTaskCompletion.cs**

```
using System;
using System.ComponentModel;
using System.Threading.Tasks;

public sealed class NotifyTaskCompletion<TResult> : INotifyPropertyChanged
{
  public NotifyTaskCompletion(Task<TResult> task)
  {
    Task = task;
    if (!task.IsCompleted)
    {
      var _ = WatchTaskAsync(task);
    }
  }

  private async Task WatchTaskAsync(Task task)
  {
    try
    {
      await task;
    }
    catch
    {
    }

    var propertyChanged = PropertyChanged;
    if (propertyChanged == null)
        return;

    propertyChanged(this, new PropertyChangedEventArgs("Status"));
    propertyChanged(this, new PropertyChangedEventArgs("IsCompleted"));
    propertyChanged(this, new PropertyChangedEventArgs("IsNotCompleted"));
    if (task.IsCanceled)
    {
      propertyChanged(this, new PropertyChangedEventArgs("IsCanceled"));
    }
    else if (task.IsFaulted)
    {
      propertyChanged(this, new PropertyChangedEventArgs("IsFaulted"));
      propertyChanged(this, new PropertyChangedEventArgs("Exception"));
      propertyChanged(this,
        new PropertyChangedEventArgs("InnerException"));
      propertyChanged(this, new PropertyChangedEventArgs("ErrorMessage"));
    }
    else
    {
      propertyChanged(this,
        new PropertyChangedEventArgs("IsSuccessfullyCompleted"));
      propertyChanged(this, new PropertyChangedEventArgs("Result"));
    }
  }

  public Task<TResult> Task { get; private set; }
  public TResult Result { get { return (Task.Status == TaskStatus.RanToCompletion) ?
    Task.Result : default(TResult); } }
  public TaskStatus Status { get { return Task.Status; } }
  public bool IsCompleted { get { return Task.IsCompleted; } }
  public bool IsNotCompleted { get { return !Task.IsCompleted; } }
  public bool IsSuccessfullyCompleted { get { return Task.Status ==
    TaskStatus.RanToCompletion; } }
  public bool IsCanceled { get { return Task.IsCanceled; } }
  public bool IsFaulted { get { return Task.IsFaulted; } }
  public AggregateException Exception { get { return Task.Exception; } }
  public Exception InnerException { get { return (Exception == null) ?
    null : Exception.InnerException; } }
  public string ErrorMessage { get { return (InnerException == null) ?
    null : InnerException.Message; } }

  public event PropertyChangedEventHandler PropertyChanged;
}
```

Let's walk through the core method NotifyTaskCompletion<T>.WatchTaskAsync. This method takes a task representing the asynchronous operation, and (asynchronously) waits for it to complete. Note that the await does *not* use ConfigureAwait(false); I want to return to the UI context before raising the PropertyChanged notifications. This method violates a common coding guideline here: It has an empty general catch clause. In this case, though, that's exactly what I want. I don't want to propagate exceptions directly back to the main UI loop; I want to capture any exceptions and set properties so that the error handling is done via data binding. When the task completes, the type raises PropertyChanged notifications for all the appropriate properties.

An updated ViewModel using NotifyTaskCompletion<T> would look like this:

```
public class MainViewModel
{
  public MainViewModel()
  {
    UrlByteCount = new NotifyTaskCompletion<int>(
      MyStaticService.CountBytesInUrlAsync("http://www.example.com"));
  }

  public NotifyTaskCompletion<int> UrlByteCount { get; private set; }
}
```

This ViewModel will start the operation immediately and then create a data-bound "watcher" for the resulting task. The View data-binding code needs to be updated to bind explicitly to the result of the operation, like this:

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Label Content="{Binding UrlByteCount.Result}"/>
  </Grid>
</Window>
```

Note that the label content is data-bound to NotifyTaskCompletion<T>.Result, not Task<T>.Result. NotifyTaskCompletion<T>.Result is data-binding friendly: It is *not* blocking, and it will notify the binding when the task completes. If you run the code now, you'll find it behaves just like the previous example: The UI is responsive and loads immediately (displaying the default value of "0") and then updates in a few seconds with the actual results.

The benefit of NotifyTaskCompletion<T> is it has many other properties as well, so you can use data binding to show busy indicators or error details. It isn't difficult to use some of these convenience properties to create a busy indicator and error

Figure 5 **MainWindow.xaml**

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
    <BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
  </Window.Resources>
  <Grid>
    <!-- Busy indicator -->
    <Label Content="Loading..." Visibility="{Binding UrlByteCount.IsNotCompleted,
      Converter={StaticResource BooleanToVisibilityConverter}}"/>

    <!-- Results -->
    <Label Content="{Binding UrlByteCount.Result}" Visibility="{Binding
      UrlByteCount.IsSuccessfullyCompleted,
      Converter={StaticResource BooleanToVisibilityConverter}}"/>

    <!-- Error details -->
    <Label Content="{Binding UrlByteCount.ErrorMessage}" Background="Red"
      Visibility="{Binding UrlByteCount.IsFaulted,
      Converter={StaticResource BooleanToVisibilityConverter}}"/>
  </Grid>
</Window>
```

details completely in the View, such as the updated data-binding code in **Figure 5**.

With this latest update, which changes only the View, the application displays "Loading…" for a few seconds (while remaining responsive) and then updates to either the results of the operation or to an error message displayed on a red background.

NotifyTaskCompletion<T> handles one use case: When you have an asynchronous operation and want to data bind the results. This is a common scenario when doing data lookups or loading during startup. However, it doesn't help much when you have an actual *command* that's asynchronous, for example, "save the current record." (I'll consider asynchronous commands in my next article.)

At first glance, it seems like it's a lot more work to build an asynchronous UI, and that's true to some extent. Proper use of the async and await keywords strongly encourages you to design a better UX. When you move to an asynchronous UI, you find you can no longer block the UI while an asynchronous operation is in progress. You must think about what the UI *should* look like during the loading process, and purposefully design for that state. This is more work, but it is work that should be done for most modern applications. And it's one reason that newer platforms such as the Windows Store support only asynchronous APIs: to encourage developers to design a more responsive UX.

Wrapping Up

When a code base is converted from synchronous to asynchronous, usually the service or data access components change first, and async grows from there toward the UI. Once you've done it a few times, translating a method from synchronous to asynchronous becomes fairly straightforward. I expect (and hope) that this translation will be automated by future tooling. However, when async hits the UI, that's when real changes are necessary.

When the UI becomes asynchronous, you must address situations where your applications are unresponsive by enhancing their UI design. The end result is a more responsive, more modern application. "Fast and fluid," if you will.

This article introduced a simple type that can be summed up as a Task<T> for data binding. Next time, I'll look at asynchronous commands, and explore a concept that's essentially an "ICommand for async." Then, in the final article in the series, I'll wrap up by considering asynchronous services. Keep in mind the community is still developing these patterns; feel free to adjust them for your particular needs. ■

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 homepage, including his blog, is at stephencleary.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *James McCaffrey and Stephen Toub*

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

MAY THE CODE BE WITH YOU!

PLATINUM SPONSOR esri

SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA

LAS VEGAS 2014

March 10 - 14 | Planet Hollywood Resort & Casino

LIVE! 360 DEV

Visual Studio LIVE!

YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE

LIVE!

Visual Studio Live! is your guide to the .NET Development universe, featuring code-filled days, networking nights and independent education. Whether you are a .NET developer, software architect or a designer, Visual Studio Live!'s multi-track events include focused, cutting-edge education on the .NET platform that you'll be ready to implement as soon as you get back to the office.

COMPREHENSIVE TRAINING FOR THE DEVELOPER WORLD.

Visual Studio Live! Las Vegas is part of Live! 360 DEV, which means you'll have access to four (4) other co-located events **at no additional cost:**

SQL Server LIVE!
SQL SERVER FOR MODERN DEVELOPERS

SharePoint LIVE!
TRAINING FOR COLLABORATION

Modern Apps LIVE!
MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

AND INTRODUCING Web Dev Live!

Web Dev LIVE!
HTML5, JAVASCRIPT & ASP.NET TRAINING

Five (5) events means over a hundred sessions to choose from – mix and match sessions to create your own, custom event line-up - it's like no other dev conference available today!

Sessions are filling up quickly – Register Today!

Use promo code VSLMAR2

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

vslive.com/lasvegas

Asynchronous TCP Sockets as an Alternative to WCF

James McCaffrey

In a Microsoft technologies environment, using Windows Communication Foundation (WCF) is a common approach for creating a client-server system. There are many alternatives to WCF, of course, each with its own advantages and disadvantages, including HTTP Web Services, Web API, DCOM, AJAX Web technologies, named pipe programming and raw TCP socket programming. But if you take into account factors such as development effort, manageability, scalability, performance and security, in many situations using WCF is the most efficient approach.

However, WCF can be extremely complicated and might be overkill for some programming situations. Prior to the release of the Microsoft .NET Framework 4.5, asynchronous socket programming was, in my opinion, too difficult in most cases to justify its use. But the ease of using the new C# await and async language features changes the balance, so using socket programming for asynchronous client-server systems is now a more attractive option than it used to be. This article explains how to use these new asynchronous features of the .NET Framework 4.5 to create low-level, high-performance asynchronous client-server software systems.

This article discusses:

- Setting up a TCP socket-based service
- Creating a Windows Forms application demo client
- Creating a Web application demo client

Technologies discussed:

Visual Studio 2012, Microsoft .NET Framework 4.5, C#

Code download available at:

msdn.microsoft.com/magazine/msdnmag0314

The best way to see where I'm headed is to take a look at the demo client-server system shown in **Figure 1**. At the top of the image a command shell is running an asynchronous TCP socket-based service that accepts requests to compute the average or minimum of a set of numeric values. In the middle part of the image is a Windows Forms (WinForm) application that has sent a request to compute the average of (3, 1, 8). Notice the client is asynchronous—after the request is sent, while waiting for the service to respond, the user is able to click on the button labeled Say Hello three times, and the application is responsive.

The bottom part of **Figure 1** shows a Web application client in action. The client has sent an asynchronous request to find the minimum value of (5, 2, 7, 4). Although it's not apparent from the screenshot, while the Web application is waiting for the service response, the application is responsive to user input.

In the sections that follow, I'll show how to code the service, the WinForm client and the Web application client. Along the way I'll discuss the pros and cons of using sockets. This article assumes you have at least intermediate-level C# programming skill, but does not assume you have deep understanding or significant experience with asynchronous programming. The code download that

accompanies this article has the complete source code for the three programs shown in **Figure 1**. I have removed most normal error checking to keep the main ideas as clear as possible.

Creating the Service

The overall structure of the demo service, with a few minor edits to save space, is presented in **Figure 2**. To create the service, I launched Visual Studio 2012, which has the required .NET Framework 4.5, and created a new C# console application named DemoService. Because socket-based services tend to have specific, limited functionality, using a more descriptive name would be preferable in a real-life scenario.

After the template code loaded into the editor, I modified the using statements at the top of the source code to include System.Net and System.Net.Sockets. In the Solution Explorer window, I renamed file Program.cs to ServiceProgram.cs and Visual Studio automatically renamed class Program for me. Starting the service is simple:

```
int port = 50000;
AsyncService service = new AsyncService(port);
service.Run();
```

Each custom socket-based service on a server must use a unique port. Port numbers between 49152 and 65535 are generally used for custom services. Avoiding port number collisions can be tricky. It's possible to reserve port numbers on a server using the system registry ReservedPorts entry. The service uses an object-oriented programming (OOP) design and is instantiated via a constructor that accepts the port number. Because service port numbers are fixed, the port number can be hardcoded rather than passed as a parameter. The Run method contains a while loop that will accept and process client requests until the console shell receives an <enter> key press.

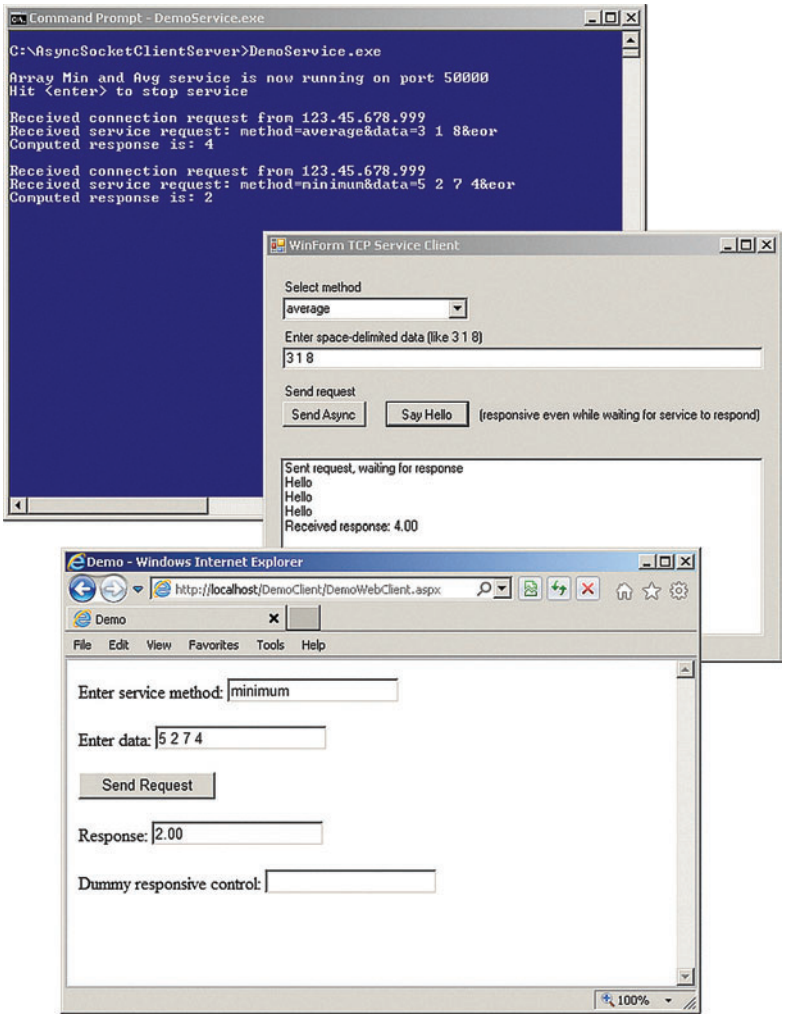


Figure 1 **Demo TCP-Based Service with Two Clients**

> Each custom socket-based service on a server must use a unique port.

The AsyncService class has two private members, ipAddress and port. These two values essentially define a socket. The constructor accepts a port number and programmatically determines the IP address of the server. Public method Run does all the work of accepting requests, then computing and sending responses. The Run method calls helper method Process, which in turn calls helper Response. Method Response calls helpers Average and Minimum.

There are many ways to organize a socket-based server. The structure used in the demo tries to strike a balance between modularity and simplicity, and has worked well for me in practice.

The Service Constructor and Run Methods

The two public methods of the socket-based demo service are presented in **Figure 3**. After storing the port name, the constructor uses method GetHostName to determine the name of the server, and to then fetch a structure that contains information about the server. The AddressList collection holds different machine addresses, including IPv4 and IPv6 addresses. The InterNetwork enum value means an IPv4 address.

This approach restricts the server to listen to requests using only the server's first assigned IPv4 address. A simpler alternative could allow the server to accept requests sent to any of its addresses by just assigning the member field as this.ipAddress = IPAddress.Any.

Notice the service's Run method signature uses the async modifier, indicating that in the body of the method some asynchronous method will be called in conjunction with the await keyword. The method returns void rather than the more usual Task because Run is called by the Main method, which, as a special case, does not allow the async modifier. An alternative is to define method Run to return type Task and then call the method as service.Run().Wait.

The service's Run method instantiates a TcpListener object using the server's IP address and port number. The listener's Start method begins monitoring the specified port, waiting for a connection request.

Figure 2 **The Demo Service Program Structure**

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading.Tasks;
namespace DemoService
{
  class ServiceProgram
  {
    static void Main(string[] args)
    {
      try
      {
        int port = 50000;
        AsyncService service = new AsyncService(port);
        service.Run();
        Console.ReadLine();
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
        Console.ReadLine();
      }
    }
  }

  public class AsyncService
  {
    private IPAddress ipAddress;
    private int port;

    public AsyncService(int port) { . . }
    public async void Run() { . . }
    private async Task Process(TcpClient tcpClient) { . . }
    private static string Response(string request)
    private static double Average(double[] vals) { . . }
    private static double Minimum(double[] vals) { . . }
  }
}
```

Figure 3 **Service Constructor and Run Methods**

```
public AsyncService(int port)
{
  this.port = port;
  string hostName = Dns.GetHostName();
  IPHostEntry ipHostInfo = Dns.GetHostEntry(hostName);
  this.ipAddress = null;
  for (int i = 0; i < ipHostInfo.AddressList.Length; ++i) {
    if (ipHostInfo.AddressList[i].AddressFamily ==
      AddressFamily.InterNetwork)
    {
      this.ipAddress = ipHostInfo.AddressList[i];
      break;
    }
  }
  if (this.ipAddress == null)
    throw new Exception("No IPv4 address for server");
}

public async void Run()
{
  TcpListener listener = new TcpListener(this.ipAddress, this.port);
  listener.Start();
  Console.Write("Array Min and Avg service is now running"
  Console.WriteLine(" on port " + this.port);
  Console.WriteLine("Hit <enter> to stop service\n");

  while (true) {
    try {
      TcpClient tcpClient = await listener.AcceptTcpClientAsync();
      Task t = Process(tcpClient);
      await t;
    }
    catch (Exception ex) {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Inside the main processing while loop, a TcpClient object, which you can think of as an intelligent socket, is created and waits for a connection via the AcceptTcpClientAsync method. Prior to the .NET Framework 4.5, you'd have to use BeginAcceptTcpClient and then write custom asynchronous coordination code, which, believe me, is not simple. The .NET Framework 4.5 adds many new methods that, by convention, end with "Async." These new methods, combined with the async and await keywords, make asynchronous programming much, much easier.

Method Run calls method Process using two statements. An alternative is to use shortcut syntax and call method Process in a single statement: await Process(tcpClient).

One of the advantages of using low-level sockets instead of WCF is that you can easily insert diagnostic WriteLine statements anywhere you choose.

To summarize, the service uses TcpListener and TcpClient objects to hide the complexity of raw socket programming, and uses the new AcceptTcpClientAsync method in conjunction with the new async and await keywords to hide the complexity of asynchronous programming. Method Run sets up and coordinates connection activities, and calls method Process to process requests and then a second statement to await on the return Task.

The Service Process and Response Methods

The Process and Response methods of the service object are presented in **Figure 4**. The Process method's signature uses the async modifier and returns type Task.

One of the advantages of using low-level sockets instead of Windows Communication Foundation (WCF) is that you can easily insert diagnostic WriteLine statements anywhere you choose. In the demo, I replaced clientEndPoint with the dummy IP address value 123.45.678.999 for security reasons.

The three key lines in method Process are:

```
string request = await reader.ReadLineAsync();
...
string response = Response(request);
...
await writer.WriteLineAsync(response);
```

You can interpret the first statement to mean, "read a line of the request asynchronously, allowing other statements to execute if necessary." Once the request string is obtained, it's passed to the Response helper. Then the response is sent back to the requesting client asynchronously.

The server is using a read-request, write-response cycle. It's simple, but there are several caveats of which you should be aware. If the server reads without writing, it can't detect a half-open situation. If the server writes without reading (for example, responding with a large amount of data), it could deadlock with the client. A

BEST SELLER

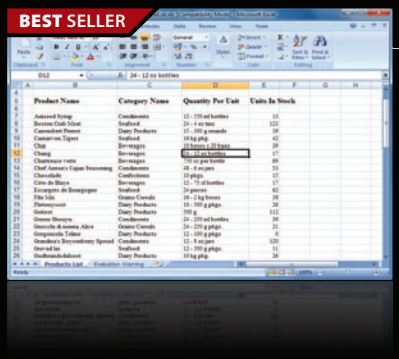
Help & Manual Professional | from $583.10

ec software

Easily create documentation for Windows, the Web and iPad.

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

BEST SELLER



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

ASPOSE
Your File Format Experts

Every Aspose .NET component in one package.

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

BEST SELLER

GdPicture.NET Ultimate | from $4,127.59

GdPicture
imaging technologies

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

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

BEST SELLER

ComponentOne ActiveReports 8 | from $1,567.02

ComponentOne
a division of GrapeCity

The award-winning .NET reporting tool for HTML5, WPF, WinForms, ASP.NET & Windows Azure.

- Create sophisticated, fast and powerful reports
- Generate flexible layouts using Section, Region and Fixed page designers
- Experience a new scalable, distributed and load balanced Enterprise-grade report server
- Utilize the just released HTML5 and touch-optimized report viewers for mobile devices
- Explore an array of data visualization options including charts, maps and more

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

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

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

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

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

GSA Schedule Contract GS-35F-0188R

read-write design is acceptable for simple in-house services but shouldn't be used for services that are critical or public-facing.

The Response method accepts the request string, parses the request and computes a response string. A simultaneous strength and weakness of a socket-based service is that you must craft some sort of custom protocol. In this case, requests are assumed to look like:

```
method=average&data=1.1 2.2 3.3&eor
```

In other words, the service expects the literal "method=" followed by the string "average" or "minimum," then an ampersand character ("&") followed by the literal "data=". The actual input data must be in space-delimited form. The request is terminated by an "&" followed by the literal "eor," which stands for end-of-request. A disadvantage of socket-based services compared to WCF is that serializing complex parameter types can be a bit tricky sometimes.

In this demo example, the service response is simple, just a string representation of the average or minimum of an array of numeric values. In many custom client-server situations, you'll have to design some protocol for the service response. For example, instead of sending a response just as "4.00," you might want to send the response as "average=4.00."

Method Process uses a relatively crude approach to close a connection if an Exception occurs. An alternative is to use the C# using statement (which will automatically close any connection) and remove the explicit call to method Close.

Figure 4 **The Demo Service Process and Response Methods**

```
private async Task Process(TcpClient tcpClient)
{
  string clientEndPoint =
    tcpClient.Client.RemoteEndPoint.ToString();
  Console.WriteLine("Received connection request from "
    + clientEndPoint);
  try {
    NetworkStream networkStream = tcpClient.GetStream();
    StreamReader reader = new StreamReader(networkStream);
    StreamWriter writer = new StreamWriter(networkStream);
    writer.AutoFlush = true;
    while (true) {
      string request = await reader.ReadLineAsync();
      if (request != null) {
        Console.WriteLine("Received service request: " + request);
        string response = Response(request);
        Console.WriteLine("Computed response is: " + response + "\n");
        await writer.WriteLineAsync(response);
      }
      else
        break; // Client closed connection
    }
    tcpClient.Close();
  }
  catch (Exception ex) {
    Console.WriteLine(ex.Message);
    if (tcpClient.Connected)
      tcpClient.Close();
  }
}

private static string Response(string request)
{
  string[] pairs = request.Split('&');
  string methodName = pairs[0].Split('=')[1];
  string valueString = pairs[1].Split('=')[1];

  string[] values = valueString.Split(' ');
  double[] vals = new double[values.Length];
  for (int i = 0; i < values.Length; ++i)
    vals[i] = double.Parse(values[i]);

  string response = "";
  if (methodName == "average") response += Average(vals);
  else if (methodName == "minimum") response += Minimum(vals);
  else response += "BAD methodName: " + methodName;

  int delay = ((int)vals[0]) * 1000; // Dummy delay
  System.Threading.Thread.Sleep(delay);

  return response;
}
```

> An advantage of low-level services is that you have greater control over your data-access approach.

Helper methods Average and Minimum are defined as:

```
private static double Average(double[] vals)
{
  double sum = 0.0;
  for (int i = 0; i < vals.Length; ++i)
    sum += vals[i];
  return sum / vals.Length;
}
private static double Minimum(double[] vals)
{
  double min = vals[0]; ;
  for (int i = 0; i < vals.Length; ++i)
    if (vals[i] < min) min = vals[i];
  return min;
}
```

In most situations, if you're using a program structure similar to the demo service, your helper methods at this point would connect to some data source and fetch some data. An advantage of low-level services is that you have greater control over your data-access approach. For example, if you're getting data from SQL, you can use classic ADO.NET, the Entity Framework or any other data access method.

A disadvantage of a low-level approach is you must explicitly determine how to handle errors in your system. Here, if the demo service is unable to satisfactorily parse the request string, instead of returning a valid response (as a string), the service returns an error message. Based on my experience, there are very few general principles on which to rely. Each service requires custom error handling.

Notice the Response method has a dummy delay:

```
int delay = ((int)vals[0]) * 1000;
System.Threading.Thread.Sleep(delay);
```

This response delay, arbitrarily based on the first numeric value of the request, was inserted to slow the service down so that the WinForm and Web application clients could demonstrate UI responsiveness while waiting for a response.

The WinForm Application Demo Client

To create the WinForm client shown in **Figure 1**, I launched Visual Studio 2012 and created a new C# WinForm application named DemoFormClient. Note that, by default, Visual Studio modularizes a WinForm application into several files that separate the UI code from the logic code. For the code download that accompanies this article, I refactored the modularized Visual Studio code into a single source code file. You can compile the application by launching a Visual Studio

Figure 5 **WinForm Demo Client Button Click Handlers**

```
private async void button1_Click(object sender, EventArgs e)
{
  try {
    string server = "mymachine.network.microsoft.com";
    int port = 50000;
    string method = (string)comboBox1.SelectedItem;
    string data = textBox1.Text;
    Task<string> tsResponse = SendRequest(server, port, method, data);
    listBox1.Items.Add("Sent request, waiting for response");
    await tsResponse;
    double dResponse = double.Parse(tsResponse.Result);
     istBox1.Items.Add("Received response: " + dResponse.ToString("F2"));
  }
  catch (Exception ex) {
    listBox1.Items.Add(ex.Message);
  }
}

private void button2_Click(object sender, EventArgs e)
{
  listBox1.Items.Add("Hello");
}
```

Figure 6 **WinForm Demo Client SendRequest Method**

```
private static async Task<string> SendRequest(string server,
  int port, string method, string data)
{
  try {
    IPAddress ipAddress = null;
    IPHostEntry ipHostInfo = Dns.GetHostEntry(server);
    for (int i = 0; i < ipHostInfo.AddressList.Length; ++i) {
      if (ipHostInfo.AddressList[i].AddressFamily ==
        AddressFamily.InterNetwork)
      {
        ipAddress = ipHostInfo.AddressList[i];
        break;
      }
    }
    if (ipAddress == null)
      throw new Exception("No IPv4 address for server");

    TcpClient client = new TcpClient();
    await client.ConnectAsync(ipAddress, port); // Connect

    NetworkStream networkStream = client.GetStream();
    StreamWriter writer = new StreamWriter(networkStream);
    StreamReader reader = new StreamReader(networkStream);

    writer.AutoFlush = true;
    string requestData = "method=" + method + "&" + "data=" +
      data + "&eor"; // 'End-of-request'
    await writer.WriteLineAsync(requestData);
    string response = await reader.ReadLineAsync();

    client.Close();
    return response;
  }
  catch (Exception ex) {
    return ex.Message;
  }
}
```

command shell (which knows where the C# compiler is), and executing the command: csc.exe /target:winexe DemoFormClient.cs.

Using the Visual Studio design tools, I added a ComboBox control, a TextBox control, two Button controls and a ListBox control, along with four Label controls. For the ComboBox control, I added strings "average" and "minimum" to the control's Items collection property. I changed the Text properties of button1 and button2 to Send Async and Say Hello, respectively. Then, in design view, I double-clicked on the button1 and button2 controls to register their event handlers. I edited the click handlers as shown in **Figure 5**.

Notice the signature of the button1 control's click handler was changed to include the async modifier. The handler sets up a hardcoded server machine name as a string and port number. When using low-level socket-based services, there's no automatic discovery mechanism and so clients must have access to the server name or IP address and port information.

The key lines of code are:

```
Task<string> tsResponse = SendRequest(server, port, method, data);
// Perform some actions here if necessary
await tsResponse;
double dResponse = double.Parse(tsResponse.Result);
```

SendRequest is a program-defined asynchronous method. The call can be loosely interpreted as "send an asynchronous request that will return a string, and when finished continue execution at the statement 'await tsResponse,' which occurs later." This allows the application to perform other actions while waiting for the response. Because the response is encapsulated in a Task, the actual string result must be extracted using the Result property. That string result is converted to type double so that it can be formatted nicely to two decimal places.

An alternative calling approach is:

```
string sResponse = await SendRequest(server, port, method, data);
double dResponse = double.Parse(sResponse);
listBox1.Items.Add("Received response: " + dResponse.ToString("F2"));
```

> A disadvantage of a low-level approach is that you must explicitly determine how to handle errors in your system.

Here, the await keyword is placed in-line with the asynchronous call to SendRequest. This simplifies the calling code a bit and also allows the return string to be fetched without a call to Task.Result. The choice of using an inline await call or using a separate-statement await call will vary from situation to situation, but as a general rule of thumb, it's better to avoid the explicit use of a Task object's Result property.

Most of the asynchronous work is performed in the SendRequest method, which is listed in **Figure 6**. Because SendRequest is asynchronous, it might better be named SendRequestAsync or MySendRequestAsync.

SendRequest accepts a string representing the server name and begins by resolving that name to an IP address using the same code logic that was used in the service class constructor. A simpler alternative is to just pass the name of the server: await client.ConnectAsync(server, port).

After the server's IP address is determined, a TcpClient intelligent-socket object is instantiated and the object's ConnectAsync method is used to send a connection request to the server. After setting up a network StreamWriter object to send data to the server and a StreamReader object to receive data from the server, a request string is created using the formatting expected by the server. The request is sent and received asynchronously and returned by the method as a string.

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

CHICAGO 2014

May 5 - 8 | Chicago Hilton

vslive.com/chicago

LIVE! Visual Studio YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE LIVE! CHICAGO

LIVE LONG AND CODE

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

Topics include:

- Visual Studio/.NET
- Windows Client (Windows 8.1/ WinRT/WPF)
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint/Office
- SQL Server

REGISTER BY APRIL 2 AND SAVE $200!

Use promo code CHTIP1

vslive.com/chicago

The Web Application Demo Client

I created the demo Web application client shown in **Figure 1** in two steps. First, I used Visual Studio to create a Web site to host the application, and then I coded the Web application using Notepad. I launched Visual Studio 2012 and created a new C# Empty Web Site named DemoClient at http://localhost/. This set up all the necessary IIS plumbing to host an application and created the physical location associated with the Web site at C:\inetpub\wwwroot\DemoClient\. The process also created a basic configuration file, Web.config, which contains information to allow applications in the site to access async functionality in the .NET Framework 4.5:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="false" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

Next, I launched Notepad with administrative privileges. When creating simple ASP.NET applications, I sometimes prefer using Notepad instead of Visual Studio so I can keep all application code in a single .aspx file, rather than generating multiple files and unwanted example code. I saved the empty file as DemoWebClient.aspx at C:\inetpub\wwwroot\DemoClient.

The overall structure of the Web application is shown in **Figure 7**.

At the top of the page I added Import statements to bring the relevant .NET namespaces into scope, and a Page directive that includes the Async=true attribute.

The C# script region contains two methods, SendRequest and Button1_Click. The application page body has two TextBox controls and one Button control for input, an output TextBox control to hold the service response, and a dummy, unused TextBox control to demonstrate UI responsiveness while the application waits for the service to respond to a request.

Figure 7 **Web Application Demo Client Structure**

```
<%@ Page Language="C#" Async="true" AutoEventWireup="true"%>
<%@ Import Namespace="System.Threading.Tasks" %>
<%@ Import Namespace="System.Net" %>
<%@ Import Namespace="System.Net.Sockets" %>
<%@ Import Namespace="System.IO" %>

<script runat="server" language="C#">
  private static async Task<string> SendRequest(string server,
  private async void Button1_Click(object sender, System.EventArgs e) { . . }
</script>

<head>
  <title>Demo</title>
</head>

<body>
  <form id="form1" runat="server">
  <div>

  <p>Enter service method:
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox></p>
  <p>Enter data:
    <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox></p>
  <p><asp:Button Text="Send Request" id="Button1"
    runat="server" OnClick="Button1_Click"> </asp:Button> </p>
  <p>Response:
    <asp:TextBox ID="TextBox3" runat="server"></asp:TextBox></p>
  <p>Dummy responsive control:
    <asp:TextBox ID="TextBox4" runat="server"></asp:TextBox></p>

  </div>
  </form>
</body>
</html>
```

The code for the Web application's SendRequest method is exactly the same as the code in the WinForm application's SendRequest. The code for the Web application's Button1_Click handler differs only slightly from the WinForm's button1_Click handler to accommodate the different UI:

```
try {
  string server = "mymachine.network.microsoft.com";
  int port = 50000;
  string method = TextBox1.Text;
  string data = TextBox2.Text;
  string sResponse = await SendRequest(server, port, method, data);
  double dResponse = double.Parse(sResponse);
  TextBox3.Text = dResponse.ToString("F2");
}
catch (Exception ex) {
  TextBox3.Text = ex.Message;
}
```

Even though the code for the Web application is essentially the same as the code for the WinForm application, the calling mechanism is quite a bit different. When a user makes a request using the WinForm, the WinForm issues the call directly to the service and the service responds directly to the WinForm. When a user makes a request from the Web application, the Web application sends the request information to the Web server that's hosting the application, the Web server makes the call to the service, the service responds to the Web server, the Web server constructs a response page that includes the response and the response page is sent back to the client browser.

Wrapping Up

So, when should you consider using asynchronous TCP sockets instead of WCF? Roughly 10 years ago, before the creation of WCF and its predecessor technology ASP.NET Web Services, if you wanted to create a client-server system, using sockets was often the most logical option. The introduction of WCF was a big advance, but because of the huge number of scenarios WCF is designed to handle, using it for simple client-server systems might be overkill in some situations. Although the latest version of WCF is easier to configure than previous versions, it can still be tricky to work with WCF.

For situations where the client and server are on different networks, making security a major consideration, I always use WCF. But for many client-server systems where client and server are located on a single secure enterprise network, I often prefer using TCP sockets.

A relatively new approach for implementing client-server systems is to use the ASP.NET Web API framework for HTTP-based services combined with the ASP.NET SignalR library for asynchronous methods. This approach, in many cases, is simpler to implement than using WCF and avoids many of the low-level details involved with a socket approach. ■

Dr. James McCaffrey *works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Internet Explorer and Bing. He can be reached at jammc@microsoft.com.*

Thanks to the following technical experts for their advice and for reviewing this article: *Piali Choudhury (MS Research), Stephen Cleary (consultant), Adam Eversole (MS Research) Lynn Powers (MS Research) and Stephen Toub (Microsoft)*

A .NET Developer Primer for Single-Page Applications

Long Le

A majority of Microsoft .NET Framework developers have spent most of their professional lives on the server side, coding with C# or Visual Basic .NET when building Web applications. Of course, JavaScript has been used for simple things such as modal windows, validation, AJAX calls and so on. However, JavaScript (client-side code for the most part) has been leveraged as a utility language, and applications were largely driven from the server side.

Lately there's been a huge trend of Web application code migrating from the server side to the client side (browser) to meet users' expectations for fluid and responsive UX. With this being the case, a lot of .NET developers (especially in the enterprise) are dealing with an extreme amount of anxiety about JavaScript best practices, architecture, unit testing, maintainability and the recent explosion of different kinds of JavaScript libraries. Part of the trend of moving to the client side is the increasing use of single-page applications (SPAs). To say that SPA development is the future is an extreme understatement. SPAs are how some of the best applications on the Web offer fluid UX and responsiveness, while minimizing payloads (traffic) and round-trips to the server.

> To say that SPA development is the future is an extreme understatement.

In this article, I'll address the anxieties you might experience when making the transition from the server side into the SPA realm. The best way to deal with these anxieties is to embrace JavaScript as a first-class language just like any .NET language, such as C#, Visual Basic .NET, Python and so on.

Following are some fundamental principles of .NET development that are sometimes ignored or forgotten when developing apps in JavaScript:

- Your code base is manageable in .NET because you're decisive with class boundaries and where classes actually live within your projects.

This article discusses:

- Steps to convert an ASP.NET MVC 5 application to a single-page application (SPA)
- Setting up an SPA infrastructure
- Adding create, read, update and delete functionality

Technologies discussed:

ASP.NET MVC 5, JavaScript, Single-Page Applications, Kendo UI, RequireJS, Entity Framework, Web API, OData

Code download available at:

easyspa.codeplex.com

- You separate concerns, so you don't have classes that are responsible for hundreds of different things with overlapping responsibilities.
- You have reusable repositories, queries, entities (models) and data sources.
- You put some thought into naming your classes and files so they're more meaningful.
- You practice good use of design patterns, coding conventions and organization.

Because this article is for .NET developers who are being introduced to the SPA world, I'll incorporate the least number of frameworks possible to build a manageable SPA with sound architecture.

Creating an SPA in Seven Key Steps

Following are seven key steps to convert a new ASP.NET Web Application that was created with the out-of-the-box Visual Studio 2013 ASP.NET MVC template into an SPA (with references to the appropriate project files that can be found in the accompanying code download).

1. Download and install the NuGet packages RequireJS, RequireJS text plug-in and Kendo UI Web.
2. Add a configuration module (Northwind.Web/Scripts/app/main.js).
3. Add an app module (Northwind.Web/Scripts/app/app.js).
4. Add a router module (Northwind.Web/Scripts/app/router.js).
5. Add an action and view both named Spa (Northwind.Web/Controllers/HomeController.cs and Northwind.Web/Views/Home/Spa.cshtml).
6. Modify the _ViewStart.cshtml file so MVC will load views without using the _Layout.cshtml file by default (Northwind.Web/Views/_ViewStart.cshtml).
7. Update the layout navigation (menu) links to match the new SPA-friendly URLs (Northwind.Web/Views/Shared/_Layout.cshtml).

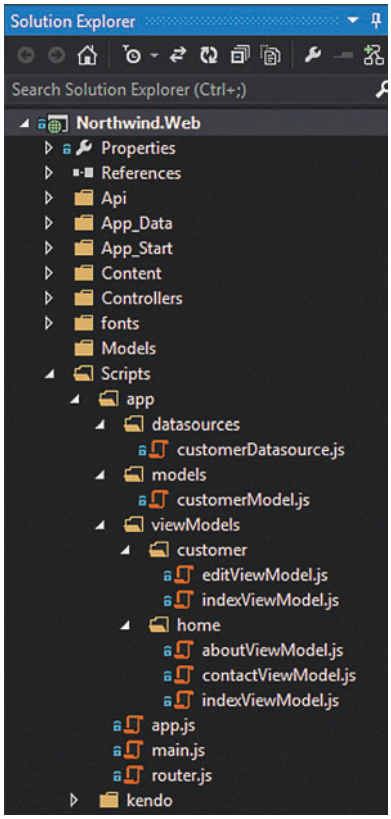


Figure 1 **ASP.NET MVC Project Structure**

After these seven steps have been carried out, your Web application project structure should look something like **Figure 1**.

I'll show how to build an awesome SPA in ASP.NET MVC with the following JavaScript libraries, available via NuGet:

- **RequireJS (requirejs.org):** This is a JavaScript file and module loader. RequireJS will provide #include/import/require APIs and the ability to load nested dependencies with dependency injection (DI). The RequireJS design approach uses the Asynchronous Module Definition (AMD) API for JavaScript modules, which helps to encapsulate pieces of code into useful units. It also provides an intuitive way to refer to other units of code (modules). RequireJS modules also follow the module pattern (bit.ly/18byc2Q). A simplified implementation of this pattern uses JavaScript functions for encapsulation. You'll see this pattern in action later as all JavaScript modules will be wrapped within a "define" or "require" function. Those familiar with DI and Inversion of Control (IoC) concepts can think of this as a client-side DI framework. If that's as clear as mud at the moment, no worries—I'll soon get into some coded illustrations where all this will make sense.
- **Text plug-in for RequireJS (bit.ly/1cd8lTZ):** This will be used to remotely load chunks of HTML (views) into the SPA.
- **Entity Framework (bit.ly/1bKiZ9l):** This is pretty self-explanatory, and because the focus of this article is on SPA, I won't get too much into Entity Framework. However, if you're new to this, there's plenty of documentation available.
- **Kendo UI Web (bit.ly/t4VkVp):** This is a comprehensive JavaScript/HTML5 framework that encompasses Web UI Widgets, DataSources, templates, the Model-View-ViewModel (MVVM) pattern, SPAs, styling, and so on to help deliver a responsive and adaptive application that will look great.

Figure 2 **RequireJS Configuration**

```
require.config({
  paths: {
    // Packages
    'jquery': '/scripts/jquery-2.0.3.min',
    'kendo': '/scripts/kendo/2013.3.1119/kendo.web.min',
    'text': '/scripts/text',
    'router': '/scripts/app/router'
  },
  shim : {
    'kendo' : ['jquery']
  },
  priority: ['text', 'router', 'app'],
  jquery: '2.0.3',
  waitSeconds: 30
});

require([
  'app'
], function (app) {
  app.initialize();
});
```

Figure 3 **Registered Route Definitions and Corresponding URLs**

Registered Route (Definition)	Actual Full (Bookmarkable) URL
/	localhost:25061/home/spa/home/index
/home/index	localhost:25061/home/spa/#/home/index/home/about
/home/about	localhost:25061/home/spa/#/home/about/home/contact
/home/contact	localhost:25061/home/spa/#/home/contact/customer/index
/customer/index	localhost:25061/home/spa/#/customer/index

Setting up the SPA Infrastructure

To show how to set up the SPA infrastructure, first I'll explain how to create the RequireJS (config) module (Northwind.Web/Scripts/app/main.js). This module will be the app start-up entry point. If you've created a console app, you can think of this as the Main entry point in Program.cs. It basically contains the first class and the method that's called when the SPA starts up. The main.js file basically serves as the SPA's manifest and is where you'll define where all things in the SPA are and their dependencies, if any. The code for RequireJS configuration is shown in **Figure 2**.

> You have two options for SPA views that will be loaded into the SPA: standard HTML (*.html) or ASP.NET MVC Razor (*.cshtml) pages.

In **Figure 2**, the paths property contains a list of where all the modules are located and their names. Shim is the name of a module defined previously. The shim property includes any dependencies the module may have. In this case, you're loading a module named kendo and it has a dependency on a module named jquery, so if a module requires the kendo module, go ahead and load jQuery first, because jQuery has been defined as a dependency for the kendo module.

In **Figure 2**, the code "require([], function(){})" will load in the next module, which is the module I named app. Note that I've deliberately given meaningful names to modules.

So, how does your SPA know to invoke this module first? You configure this on the first landing page in the SPA with the data-main attribute in the script reference tag for RequireJS. I've specified that it run the main module (main.js). RequireJS will handle all the heavy lifting involved in loading this module; you just have to tell it which module to load first.

You have two options for SPA views that will be loaded into the SPA: standard HTML (*.html) or ASP.NET MVC Razor (*.cshtml) pages. Because this article is intended for .NET developers—and a lot of enterprises have server-side libraries and frameworks they'd like to continue using in their views—I'll go with the latter option of creating Razor views.

I'll start off by adding a view and name it Spa.cshtml, as mentioned previously. This view will basically load up the shell or all the HTML for the layout of the SPA. From this view, I'll load in the other views (for example, About.cshtml, Contact.cshtml, Index.cshtml and so on) as the user navigates through the SPA, by swapping the views that replace all the HTML in the "content" div.

Creating the SPA Landing Page (Layout) (Northwind.Web/Views/Spa.cshtml) Because the Spa.cshtml view is the SPA's landing page where you'll load in all your other views, there won't be much markup here, other than referencing the required style sheets and RequireJS. Note the data-main attribute in the following code, which tells RequireJS which module to load first:

```
@{
  ViewBag.Title = "Spa";
  Layout = "~/Views/Shared/_Layout.cshtml";
}

<link href=
  "~/Content/kendo/2013.3.1119/kendo.common.min.css" rel="stylesheet" />
<link href=
  "~/Content/kendo/2013.3.1119/kendo.bootstrap.min.css" rel="stylesheet" />
<script src=
  "@Url.Content("~/scripts/require.js")"
  data-main="/scripts/app/main"></script>
<div id="app"></div>
```

Adding an Action for the SPA Layout (Northwind.Web/Controllers/HomeController.cs) To create and load the Spa.cshtml view, add an action and view:

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

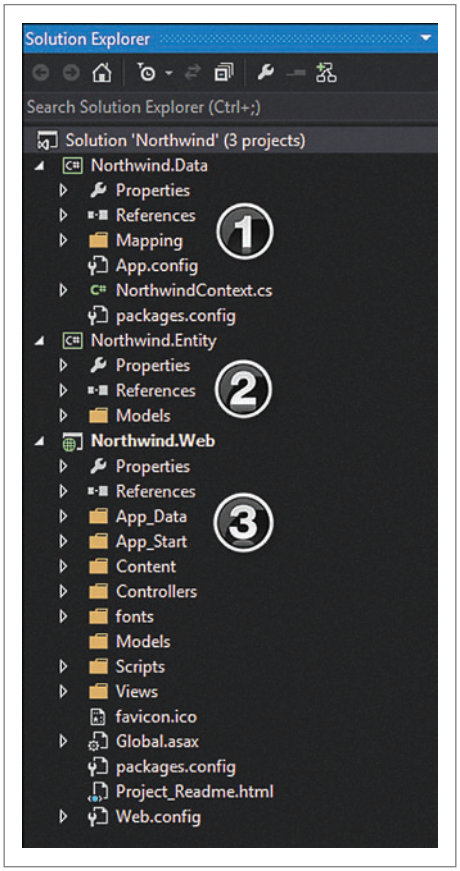


Figure 4 **A Best-Practice Solution Structure**

Create the Application Module (Northwind.Web/Scripts/app/app.js) Here's the Application module, responsible for initializing and starting the Kendo UI Router:

```
define([
  'router'
], function (router) {
  var initialize = function() {
    router.start();
  };

  return {
    initialize: initialize
  };
});
```

Create the Router Module (Northwind.Web/Scripts/app/router.js) This is called by app.js. If you're already familiar with ASP.NET MVC routes, it's the same notion here. These are the SPA routes for your views. I'll define all the routes for all the SPA views so when the user navigates through the SPA, the Kendo UI router will know what views to load into the SPA. See **Listing 1** in the accompanying download.

The Kendo UI Router class is responsible for tracking the application state and navigating between the application states. The router integrates into the browser history using the fragment part of the URL (#page), making the application states bookmarkable and linkable. When a routable URL

ALEXSYS TEAM®

WORKFLOW APPLICATIONS | HELP DESK | BUG TRACKING

MADE EASY!

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

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

Alexsys Team® Features Include:

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

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

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

FIND OUT MORE!

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

Figure 5 **Customer Grid View Markup with an MVVM Widget and Event Bindings**

```
<div class="demo-section">
  <div class="k-content" style="width: 100%">
    <div id="grid"
      data-role="grid"
      data-sortable="true"
      data-pageable="true"
      data-filterable="true"
      data-editable="inline"
      data-selectable="true"
      data-toolbar='[ { template: kendo.template($("#toolbar").html()) } ]'
      data-columns='[
        { field: "CustomerID", title: "ID", width: "75px" },
        { field: "CompanyName", title: "Company"},
        { field: "ContactName", title: "Contact" },
        { field: "ContactTitle", title: "Title" },
        { field: "Address" },
        { field: "City" },
        { field: "PostalCode" },
        { field: "Country" },
        { field: "Phone" },
        { field: "Fax" } ]'
      data-bind="source: dataSource, events:
        { change: onChange, dataBound: onDataBound }">
    </div>
    <style scoped>
    #grid .k-toolbar {
      padding: 15px;
    }

    .toolbar {
      float: right;
    }
    </style>
  </div>
</div>

<script type="text/x-kendo-template" id="toolbar">
  <div>
    <div class="toolbar">
      <span data-role="button" data-bind="click: edit">
        <span class="k-icon k-i-tick"></span>Edit</span>
      <span data-role="button" data-bind="click: destroy">
        <span class="k-icon k-i-tick"></span>Delete</span>
      <span data-role="button" data-bind="click: details">
        <span class="k-icon k-i-tick"></span>Edit Details</span>
    </div>
    <div class="toolbar" style="display:none">
      <span data-role="button" data-bind="click: save">
        <span class="k-icon k-i-tick"></span>Save</span>
      <span data-role="button" data-bind="click: cancel">
        <span class="k-icon k-i-tick"></span>Cancel</span>
    </div>
  </div>
</script>
```

is clicked, the router kicks in and tells the application to put itself back into the state that was encoded into the route. The route definition is a string representing a path used to identify the state of the application the user wants to see. When a route definition is matched from the browser's URL hash fragment, the route handler is called (see **Figure 3**).

As for the Kendo UI layout widget, its name speaks for itself. You're probably familiar with the ASP.NET Web Forms MasterPage or MVC layout included in the project when you create a new ASP.NET MVC Web Application. In this SPA project, it's located at the path Northwind.Web/Views/Shared/_Layout.cshtml. There's little difference between the Kendo UI layout and MVC layout, except the Kendo UI layout runs on the client side. Just as the layout worked on the server side, where the MVC runtime would swap out the content of the layout with other views, the Kendo UI layout works the same exact way. You swap out the view (content) of the Kendo UI layout using the showIn method. View contents (HTML) will be placed in the div with the ID "content," which was passed into the Kendo UI layout when it was initialized. After initializing the layout, you then render it inside the div with the ID "app," which is a div in the landing page (Northwind.Web/Views/Home/Spa.cshtml). I'll review that shortly.

> The Kendo UI Router class is responsible for tracking the application state and navigating between the application states.

The loadView helper method takes in a view model, a view and—if needed—a callback to invoke once the view and view model binding takes place. Within the loadView method, you leverage the Kendo UI FX library to aesthetically enhance the UX by adding some simple animation to the view swapping process. This is done by sliding the current loaded view to the left, remotely loading in the new view and then sliding the new loaded view back to the center. Obviously, you can easily change this to a variety of different animations using the Kendo UI FX library. One of the key benefits of using the Kendo UI layout is shown when you invoke the showIn method to swap out views. It will ensure the view is unloaded, destroyed properly and removed from the browser's DOM, thus ensuring the SPA can scale and is performant.

Figure 6 **Customer Web API OData Controller**

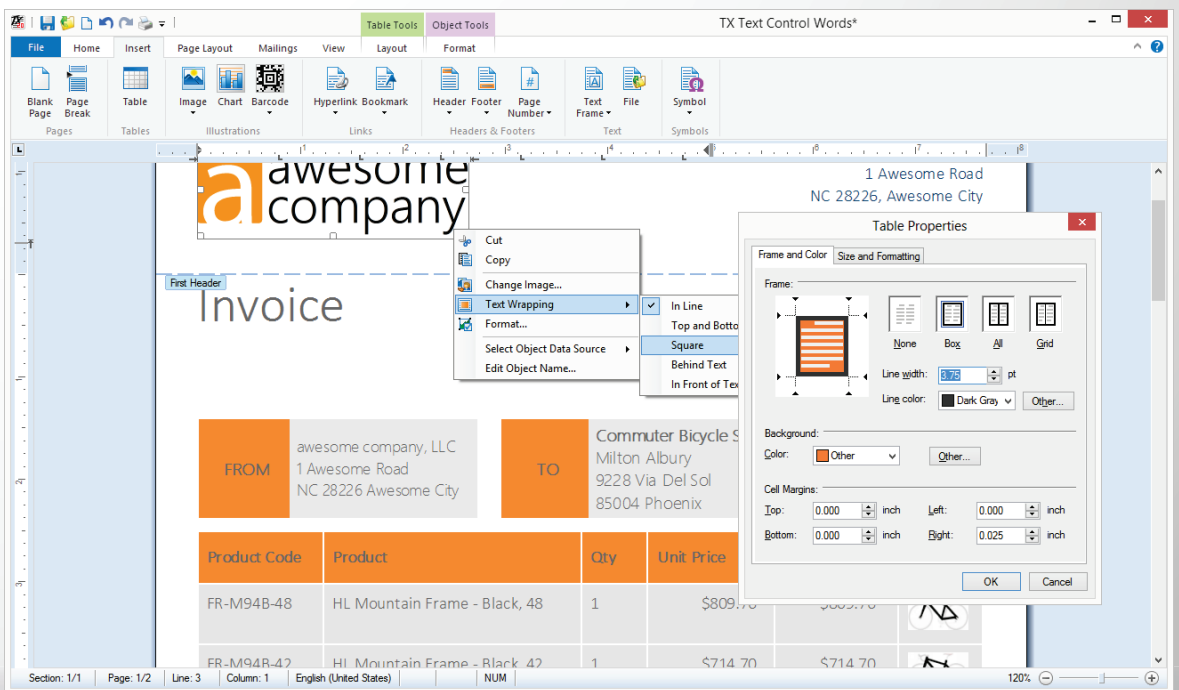
```
public class CustomerController : EntitySetController<Customer, string>
{
  private readonly NorthwindContext _northwindContext;

  public CustomerController()
  {
    _northwindContext = new NorthwindContext();
  }

  public override IQueryable<Customer> Get()
  {
    return _northwindContext.Customers;
  }

  protected override Customer GetEntityByKey(string key)
  {
    return _northwindContext.Customers.Find(key);
  }

  protected override Customer UpdateEntity(string key, Customer update)
  {
    _northwindContext.Customers.AddOrUpdate(update);
    _northwindContext.SaveChanges();
    return update;
  }

  public override void Delete(string key)
  {
    var customer = _northwindContext.Customers.Find(key);
    _northwindContext.Customers.Remove(customer);
    _northwindContext.SaveChanges();
  }
}
```

TX TEXT CONTROL®
word processing components

There are many .NET 'Rich Text Editors' out there...

Meh, sorry!

There is only one



Reporting ▪ Rich Text Editing ▪ Spell Checking ▪ Barcodes

www.textcontrol.com

/txtextcontrol

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

Visual Studio Partner

Figure 7 **Configuring ASP.NET MVC Web API Routes for OData**

```
public static void Register(HttpConfiguration config)
{
  // Web API configuration and services

  ODataModelBuilder modelBuilder = new ODataConventionModelBuilder();

  var customerEntitySetConfiguration =
    modelBuilder.EntitySet<Customer>("Customer");

  customerEntitySetConfiguration.EntityType.Ignore(t => t.Orders);
  customerEntitySetConfiguration.EntityType.Ignore(t =>
    t.CustomerDemographics);

  var model = modelBuilder.GetEdmModel();
  config.Routes.MapODataRoute("ODataRoute", "odata", model);

  config.EnableQuerySupport();

  // Web API routes
  config.MapHttpAttributeRoutes();

  config.Routes.MapHttpRoute(
    "DefaultApi", "api/{controller}/{id}",
    new {id = RouteParameter.Optional});
}
```

Edit the _ViewStart.cshtml View (Northwind.Web/Views/_ViewStart.cshtml) Here's how to configure all views to not use the ASP.NET MVC layout by default:

```
@{
  Layout = null;
}
```

At this point, the SPA should be working. When clicking on any of the navigation links on the menu, you see the current content is being swapped out via AJAX thanks to the Kendo UI router and RequireJS.

These seven steps needed to convert a fresh ASP.NET Web Application into an SPA aren't too bad, are they?

Now that the SPA is up and running, I'll go ahead and do what most developers will end up doing with an SPA, which is adding some create, read, update and delete (CRUD) functionality.

Adding CRUD Functionality to the SPA

Here are the key steps needed to add a Customer grid view to the SPA (and the related project code files):

- Add a CustomerController MVC controller (Northwind.Web/Controllers/CustomerController.cs).
- Add a REST OData Customer Web API controller (Northwind.Web/Api/CustomerController.cs).
- Add a Customer grid view (Northwind.Web/Views/Customer/Index.cshtml).

> There's little difference between the Kendo UI layout and MVC layout, except the Kendo UI layout runs on the client side.

- Add a CustomerModel module (Northwind.Web/Scripts/app/models/CustomerModel).
- Add a customerDatasource module for the Customer grid (Northwind.Web/Scripts/app/datasources/customerDatasource.js).
- Add an indexViewModel module for the Customer grid view (Northwind.Web/Scripts/app/viewModels/indexViewModel.js).

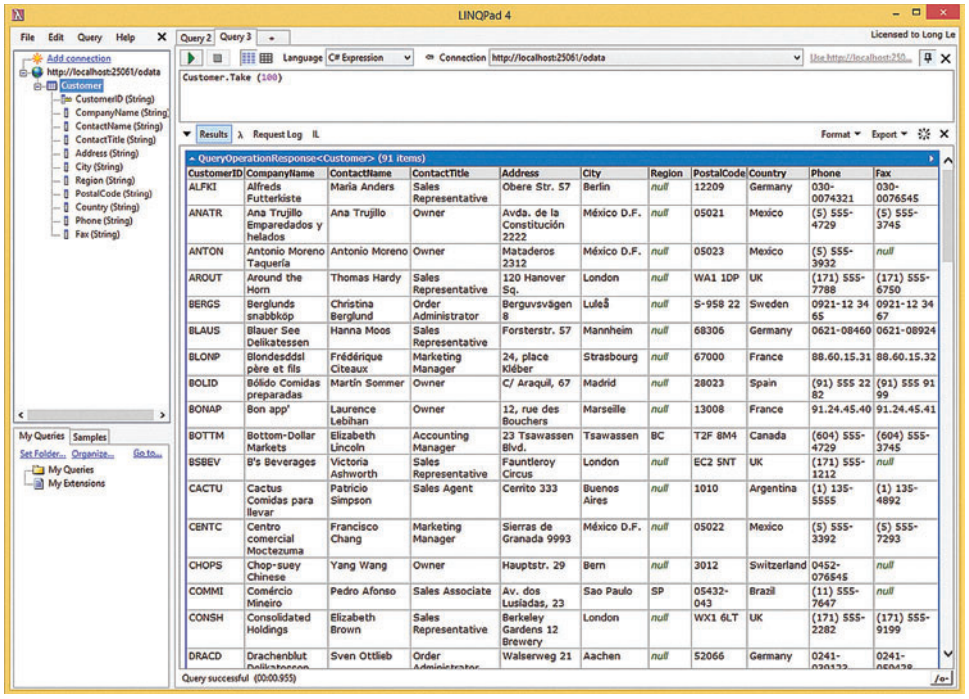


Figure 8 **Querying the Customer Controller Web API OData Via a LINQPad Query**

Setting Up the Solution Structure with Entity Framework **Figure 4** shows the solution structure, highlighting three projects: Northwind.Data (1), Northwind.Entity (2) and Northwind.Web (3). I'll briefly discuss each, along with Entity Framework Power Tools.

- **Northwind.Data:** This includes everything related to the Entity Framework Object-Relational Mapping (ORM) tool, for persistence.
- **Northwind.Entity:** This includes domain entities, composed of Plain Old CLR Object (POCO) classes. These are all the persistent-ignorant domain objects.
- **Northwind.Web:** This includes the ASP.NET MVC 5 Web Application, the presentation layer, where you'll build out the SPA with two previously

mentioned libraries—Kendo UI and RequireJS—and the rest of the server-side stack: Entity Framework, Web API and OData.

- **Entity Framework Power Tools:** To create all the POCO entities and mappings (database-first), I used the Entity Framework Power Tools from the Entity Framework team (bit.ly/1cdobhk). After the code generation, all I did here is simply copy the entities into a separate project (Northwind.Entity) to address separation concerns.

Note: Both the Northwind SQL install script and a backup of the database are included in the downloadable source code under the Northwind.Web/App_Data folder (bit.ly/1cph5qc).

Now that the solution is set up to access the database, I'll go ahead and write the MVC CustomerController.cs class to serve up the index and edit views. Because the controller's only responsibility is to serve up an HTML view for the SPA, the code here will be minimal.

Creating MVC Customer Controller (Northwind.Web/Controllers/CustomerController.cs) Here's how to create the Customer controller with the actions for the index and edit views:

```
public class CustomerController : Controller
{
  public ActionResult Index()
  {
    return View();
  }

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

Creating the View with the Customers Grid (Northwind.Web/Views/Customers/Index.cshtml) **Figure 5** shows how to create the view with the Customers grid.

If the markup in **Figure 5** isn't familiar, don't panic—it's just the Kendo UI MVVM (HTML) markup. It simply configures an HTML element, in this case the div with an ID of "grid." Later on when you bind this view to a view model with the Kendo UI MVVM framework, this markup will be converted to Kendo UI widgets. You can read more on this at bit.ly/1d2Bgfj.

Creating MVC (OData) Web API Customer Controller (Northwind.Web/Api/CustomerController.cs) Now I'll show how to create the MVC (OData) Web API Customer controller. OData is a data-access protocol for the Web that provides a uniform way to query

Figure 9 **Creating the Customer (Kendo UI Observable) Model**

```
define(['kendo'],
  function (kendo) {
    var customerModel = kendo.data.Model.define({
      id: "CustomerID",
      fields: {
        CustomerID: { type: "string", editable: false, nullable: false },
        CompanyName: { title: "Company", type: "string" },
        ContactName: { title: "Contact", type: "string" },
        ContactTitle: { title: "Title", type: "string" },
        Address: { type: "string" },
        City: { type: "string" },
        PostalCode: { type: "string" },
        Country: { type: "string" },
        Phone: { type: "string" },
        Fax: { type: "string" },
        State: { type: "string" }
      }
    });
    return customerModel;
  });
```

Figure 10 **The Customer Grid View Model**

```
define(['kendo', 'customerDatasource'],
  function (kendo, customerDatasource) {
    var lastSelectedDataItem = null;

    var onClick = function (event, delegate) {
      event.preventDefault();
      var grid = $("#grid").data("kendoGrid");
      var selectedRow = grid.select();
      var dataItem = grid.dataItem(selectedRow);
      if (selectedRow.length > 0)
        delegate(grid, selectedRow, dataItem);
      else
        alert("Please select a row.");
      };

      var indexViewModel = new kendo.data.ObservableObject({

        save: function (event) {
          onClick(event, function (grid) {
            grid.saveRow();
            $(".toolbar").toggle();
          });
        },

        cancel: function (event) {
          onClick(event, function (grid) {
            grid.cancelRow();
            $(".toolbar").toggle();
          });
        },

        details: function (event) {
          onClick(event, function (grid, row, dataItem) {
            router.navigate('/customer/edit/' + dataItem.CustomerID);
          });
        },

        edit: function (event) {
          onClick(event, function (grid, row) {
            grid.editRow(row);
            $(".toolbar").toggle();
          });
        },

        destroy: function (event) {
          onClick(event, function (grid, row, dataItem) {
            grid.dataSource.remove(dataItem);
            grid.dataSource.sync();
          });
        },

        onChange: function (arg) {
          var grid = arg.sender;
          lastSelectedDataItem = grid.dataItem(grid.select());
        },

        dataSource: customerDatasource,

        onDataBound: function (arg) {
          // Check if a row was selected
          if (lastSelectedDataItem == null) return;
          // Get all the rows
          var view = this.dataSource.view();
          // Iterate through rows
          for (var i = 0; i < view.length; i++) {
          // Find row with the lastSelectedProduct
            if (view[i].CustomerID == lastSelectedDataItem.CustomerID) {
              // Get the grid
              var grid = arg.sender;
              // Set the selected row
              grid.select(grid.table.find("tr[data-uid='" + view[i].uid + "']"));
              break;
            }
          }
        },
      });

      return indexViewModel;

  });
```

Figure 11 **RequireJS Configuration Additions**

```
paths: {
  // Packages
  'jquery': '/scripts/jquery-2.0.3.min',
  'kendo': '/scripts/kendo/2013.3.1119/kendo.web.min',
  'text': '/scripts/text',
  'router': '/scripts/app/router',
  // Models
  'customerModel': '/scripts/app/models/customerModel',
  // View models
  'customer-indexViewModel': '/scripts/app/viewmodels/customer/indexViewModel',
  'customer-editViewModel': '/scripts/app/viewmodels/customer/editViewModel',
  // Data sources
  'customerDatasource': '/scripts/app/datasources/customerDatasource',
  // Utils
  'util': '/scripts/util'
}
```

and manipulate data sets through CRUD operations. Using ASP.NET Web API, it's easy to create an OData endpoint. You can control which OData operations are exposed. You can host multiple OData endpoints alongside non-OData endpoints. You have full control over your data model, back-end business logic and data layer. **Figure 6** shows the code for the Customer Web API OData controller.

The code in **Figure 6** just creates an OData Web API controller to expose Customer data from the Northwind database. Once this is created, you can run the project, and with tools such as Fiddler (a free Web debugger at fiddler2.com) or LINQPad, you can actually query customer data.

Configuring and Exposing OData from the Customer Table for the Grid (Northwind.Web/App_Start/WebApiConfig.cs) **Figure 7** configures and exposes OData from the Customer table for the grid.

Querying OData Web API with LINQPad If you haven't used LINQPad (linqpad.net) yet, please add this tool to your developer toolkit; it's a must-have and is available in a free version. **Figure 8** shows LINQPad with a connection to the Web API OData (localhost:2501/odata), displaying the results of the LINQ query, "Customer.Take (100)."

Creating the (Observable) Customer Model (Northwind.Web/Scripts/app/models/customerModel.js) Next is creating the (Kendo UI Observable) Customer model. You can think of this as a client-side Customer entity domain model. I created the Customer model so it can easily be reused by both the Customer grid view and the edit view. The code is shown in **Figure 9**.

> OData is a data-access protocol for the Web that provides a uniform way to query and manipulate data sets through CRUD operations.

Creating a DataSource for the Customers Grid (Northwind.Web/Scripts/app/datasources/customersDatasource.js) If you're familiar with data sources from ASP.NET Web Forms, the concept is the same here, where you create a data source for the Customers grid (Northwind.Web/Scripts/app/datasources/customersDatasource.js). The Kendo UI DataSource (bit.ly/1d0Ycvd) component is an abstraction for using local (arrays of JavaScript objects) or remote (XML, JSON or JSONP) data. It fully supports CRUD data operations and provides both local and server-side support for sorting, paging, filtering, grouping and aggregates.

Creating the View Model for the Customers Grid View If you're familiar with MVVM from Windows Presentation Foundation (WPF) or Silverlight, this is the same exact concept, just on the client side (found in this project in Northwind.Web/Scripts/ViewModels/Customer/indexViewModel.cs). MVVM is an architectural separation pattern used to separate the view and its data and business logic. You'll see in a bit that all the data, business logic and so on is in the view model and that the view is purely HTML (presentation). **Figure 10** shows the code for the Customer grid view.

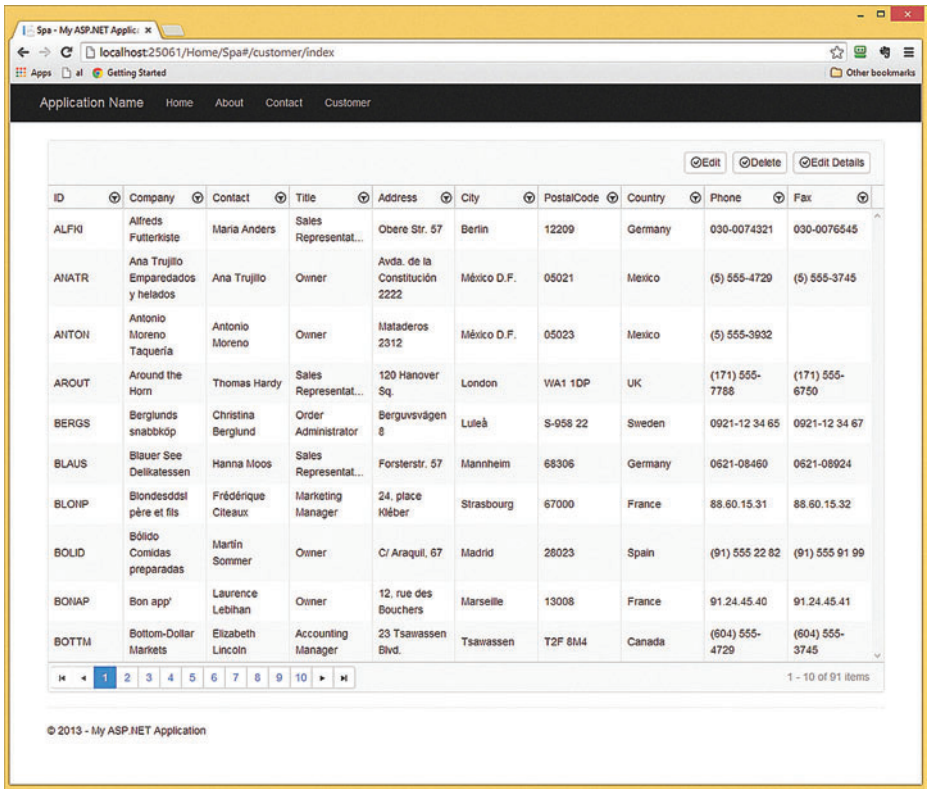
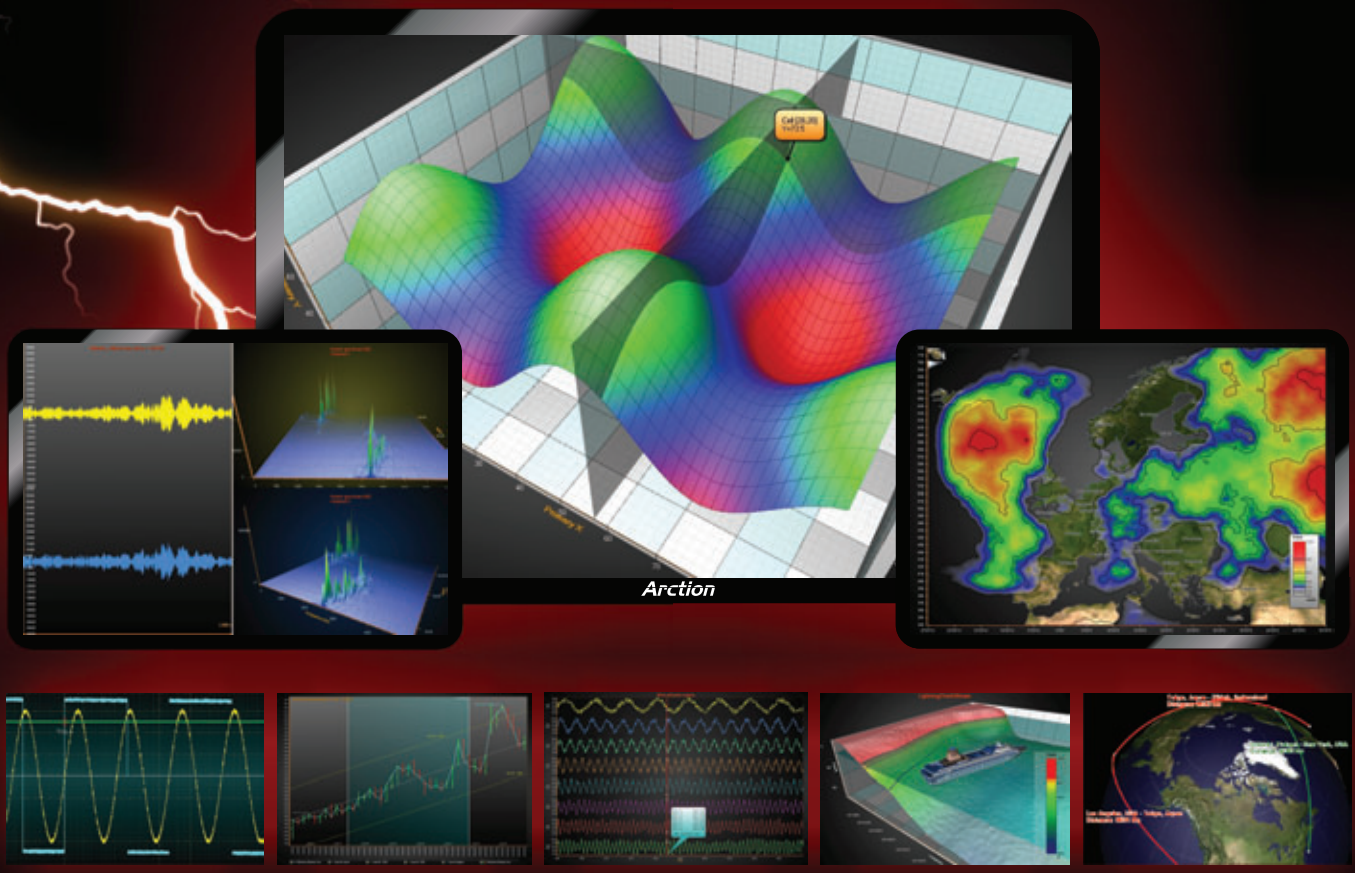


Figure 12 **The Customer Grid View with MVVM Using the Index View Model**

The fastest rendering data visualization components
for WPF and WinForms...

LightningChart



HEAVY-DUTY DATA VISUALIZATION TOOLS FOR SCIENCE, ENGINEERING AND TRADING

WPF charts performance comparison

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

Winforms charts performance comparison

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

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

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2013

Download a free 30-day evaluation from

www.LightningChart.com

Arction

Pioneers of high-performance data visualization

Figure 13 **Edit View Markup with an MVVM Widget and Event Binding**

```
<div class="demo-section">
  <div class="k-block" style="padding: 20px">
    <div class="k-block k-info-colored">
      <strong>Note: </strong>Please fill out all of the fields in this form.
    </div>
    <div>
      <dl>
        <dt>
          <label for="companyName">Company Name:</label>
        </dt>
        <dd>
          <input id="companyName" type="text"
            data-bind="value: Customer.CompanyName" class="k-textbox" />
        </dd>
        <dt>
          <label for="contactName">Contact:</label>
        </dt>
        <dd>
          <input id="contactName" type="text"
            data-bind="value: Customer.ContactName" class="k-textbox" />
        </dd>
        <dt>
          <label for="title">Title:</label>
        </dt>
        <dd>
          <input id="title" type="text"
            data-bind="value: Customer.ContactTitle" class="k-textbox" />
        </dd>
        <dt>
          <label for="address">Address:</label>
        </dt>
        <dd>
          <input id="address" type="text"
            data-bind="value: Customer.Address" class="k-textbox" />
        </dd>
        <dt>
          <label for="city">City:</label>
        </dt>
        <dd>
          <input id="city" type="text"
            data-bind="value: Customer.City" class="k-textbox" />
        </dd>
        <dt>
          <label for="zip">Zip:</label>
        </dt>
        <dd>
          <input id="zip" type="text"
            data-bind="value: Customer.PostalCode" class="k-textbox" />
        </dd>
        <dt>
          <label for="country">Country:</label>
        </dt>
        <dd>
          <input id="country" type="text"
          data-bind="value: Customer.Country" class="k-textbox" />
        </dd>
        <dt>
          <label for="phone">Phone:</label>
        </dt>
        <dd>
          <input id="phone" type="text"
            data-bind="value: Customer.Phone" class="k-textbox" />
        </dd>
        <dt>
          <label for="fax">Fax:</label>
        </dt>
        <dd>
          <input id="fax" type="text"
            data-bind="value: Customer.Fax" class="k-textbox" />
        </dd>
      </dl>

      <button data-role="button"
        data-bind="click: saveCustomer"
        data-sprite-css-class="k-icon k-i-tick">Save</button>
      <button data-role="button" data-bind="click: cancel">Cancel</button>

      <style scoped>
        dd
        {
          margin: 0px 0px 20px 0px;
          width: 100%;
        }

        label
        {
          font-size: small;
          font-weight: normal;
        }

        .k-textbox
        {
          width: 100%;
        }

        .k-info-colored
        {
          padding: 10px;
          margin: 10px;
        }
      </style>
    </div>
  </div>
</div>
```

I'll briefly describe various components of the code in **Figure 10**:

- **onClick (helper):** This method is a helper function, which gets an instance of the Customer grid, the current selected row and a JSON model of the representation of the Customer for the selected row.
- **save:** This saves changes when doing an inline edit of a Customer.
- **cancel:** This cancels out of inline edit mode.
- **details:** This navigates the SPA to the edit Customer view, appending the Customer's ID to the URL.
- **edit:** This activates inline editing for the current selected Customer.
- **destroy:** This deletes the current selected Customer.
- **onChange (event):** This fires every time a Customer is selected. You store the last selected Customer so you can maintain state. After performing any updates or navigating away from the Customer grid, when navigating back to the grid you reselect the last selected Customer.

Now add customerModel, indexViewModel and customersDataSource modules to your RequireJS configuration (Northwind.Web/Scripts/app/main.js). The code is shown in **Figure 11**.

Figure 14 **The Utility Module**

```
define([],
  function () {

    var util;

    util = {

      getId:
      function () {
        var array = window.location.href.split('/');
        var id = array[array.length - 1];
        return id;
      }
    };

    return util;

  });
```

Extreme Performance Linear Scalability

NCache™ 100% .NET

For .NET & Java Apps
(Windows Azure & Amazon AWS Supported)

JvCache™ 100% Java

Remove data storage performance bottlenecks and scale your apps to extreme transaction processing (XTP). Cache data in memory and reduce expensive database trips. NCache scales linearly by letting you add inexpensive cache servers at runtime. JvCache is 100% native Java implementation of NCache.

In-Memory 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 merge/minify

Runtime Data Sharing

- Powerful event notifications for pub/sub data sharing

FREE EXPRESS EDITION

www.alachisoft.com
sales@alachisoft.com

Download a FREE trial!

US: +1 (925) 236 3830
UK: +44 (20) 7993 8327

Figure 15 **RequireJS Configuration Additions for the Customer Edit Modules**

```
require.config({
  paths: {
    // Packages
    'jquery': '/scripts/jquery-2.0.3.min',
    'kendo': '/scripts/kendo/2013.3.1119/kendo.web.min',
    'text': '/scripts/text',
    'router': '/scripts/app/router',
    // Models
    'customerModel': '/scripts/app/models/customerModel',
    // View models
    'customer-indexViewModel': '/scripts/app/viewmodels/customer/indexViewModel',
    'customer-editViewModel': '/scripts/app/viewmodels/customer/editViewModel',
    // Data sources
    'customerDatasource': '/scripts/app/datasources/customerDatasource',
    // Utils
    'util': '/scripts/util'
    },
  shim : {
    'kendo' : ['jquery']
  },
  priority: ['text', 'router', 'app'],
  jquery: '2.0.3',
  waitSeconds: 30
  });

require([
  'app'
], function (app) {
  app.initialize();
});
```

Add a Route for the New Customers Grid View Note that in the loadView callback (in Northwind.Web/Scripts/app/router.js) you're binding the toolbar of the grid after it has been initialized and MVVM binding has taken place. This is because the first time you bind your grid, the toolbar hasn't initialized, because it exists in the grid. When the grid is first initialized via MVVM, it will load in the toolbar from the Kendo UI template. When it's loaded into the grid, you then bind only the toolbar to your view model so the buttons in your toolbar are bound to the save and cancel methods in your view model. Here's the relevant code to register the route definition for the Customer edit view:

```
router.route("/customer/index", function () {
  require(['customer-indexViewModel', 'text!/customer/index'],
    function (viewModel, view) {
      loadView(viewModel, view, function () {
        kendo.bind($("#grid").find(".k-grid-toolbar"), viewModel);
      });
    });
});
```

You now have a fully functional Customers grid view. Load up localhost:25061/Home/Spa#/customer/index (the port number will likely vary on your machine) in a browser and you'll see **Figure 12**.

Wiring Up the Customers Edit View Here are the key steps to add a Customer edit view to the SPA:

- Create a customer edit view bound to your Customer model via MVVM (Northwind.Web/Views/Customer/Edit.cshtml).
- Add an edit view model module for the Customer edit view (Northwind.Web/Scripts/app/viewModels/editViewModel.js).
- Add a utility helper module to get IDs from the URL (Northwind.Web/Scripts/app/util.js).

Because you're using the Kendo UI framework, go ahead and style your edit view with Kendo UI styles. You can learn more about that at bit.ly/1f3zWuC. **Figure 13** shows the edit view markup with an MVVM widget and event binding.

Create a Utility to Get the ID of the Customer from the URL Because you're creating concise modules with clean boundaries to create a nice separation of concerns, I'll demonstrate how to create a Util module where all of your utility helpers will reside. I'll start with a utility method that can retrieve the customer ID in the URL for the Customer DataSource (Northwind.Web/Scripts/app/datasources/customerDatasource.js), as shown in **Figure 14**.

Add the Edit View Model and Util Modules to the RequireJS Configuration (Northwind.Web/Scripts/app/main.js) The code in **Figure 15** shows RequireJS configuration additions for the Customer edit modules.

Add the Customer Edit View Model (Northwind.Web/Scripts/app/viewModels/editViewModel.js) The code in **Figure 16** shows how to add a Customer edit view model.

I'll briefly describe various components of the code in **Figure 16**:

- **saveCustomer:** This method is responsible for saving any changes on the Customer. It also resets the DataSource's filter so the grid will be hydrated with all Customers.
- **cancel:** This method will navigate the SPA back to the Customer grid view. It also resets the DataSource's filter so that the grid will be hydrated with all Customers.
- **filter:** This invokes the DataSource's filter method and queries for a specific Customer by the ID that's in the URL.

Figure 16 **Customer Edit View Model Module for the Customer View**

```
define(['customerDatasource', 'customerModel', 'util'],
  function (customerDatasource, customerModel, util) {

    var editViewModel = new kendo.data.ObservableObject({

      loadData: function () {
        var viewModel = new kendo.data.ObservableObject({
          saveCustomer: function (s) {
            customerDatasource.sync();

            customerDatasource.filter({});
            router.navigate('/customer/index');
          },
          cancel: function (s) {
            customerDatasource.filter({});
            router.navigate('/customer/index');
          }
        });

        customerDatasource.filter({
          field: "CustomerID",
          operator: "equals",
          value: util.getId()
        });

        customerDatasource.fetch(function () {
          console.log('editViewModel fetching');
          if (customerDatasource.view().length > 0) {
            viewModel.set("Customer", customerDatasource.at(0));
          } else
            viewModel.set("Customer", new customerModel());
        });
        return viewModel;
      },
    });

    return editViewModel;
  });
```

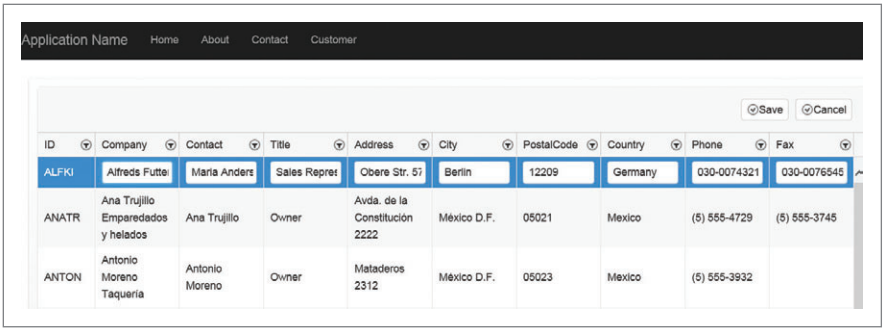


Figure 17 **The Customer Edit View**

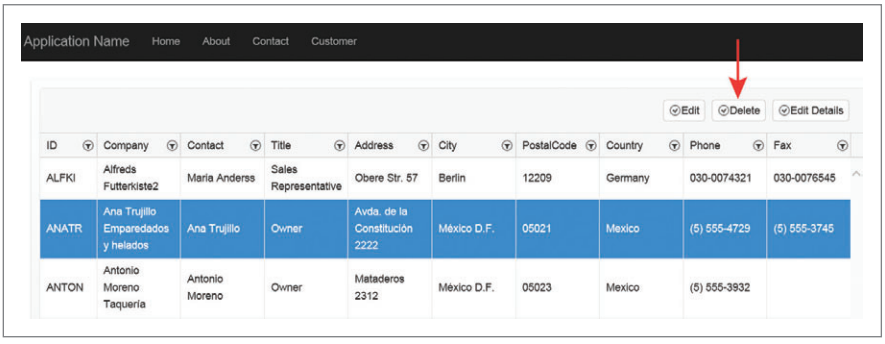


Figure 18 **The Customer Grid View**

- **fetch:** This invokes the DataSource's fetch method after setting up the filter. In the callback of the fetch, you set the Customer property of your view model with the Customer that was returned from your DataSource fetch, which will be used to bind to your Customer edit view.

When RequireJS loads a module, code within the "define" method body will only get invoked once—which is when RequireJS loads the module—so you expose a method (loadData) in your edit view model so you have a mechanism to load data after the edit view model module has already been loaded (see this in Northwind.Web/Scripts/app/router.js).

Add a Route for the New Customer Edit View (Northwind.Web/Scripts/app/router.js) Here's the relevant code to add the router:

```
router.route("/customer/edit/:id",
  function () {
  require(['customer-editViewModel',
    'text!/customer/edit'],
    function (viewModel, view) {
    loadView(viewModel.loadData(), view);
  });
});
```

Note that when the Customer edit view model is requested from RequireJS, you're able to retrieve the Customer by invoking the loadData method from the view model. This way you're able to load the correct Customer data based on the ID that's in the URL each and every time the Customer edit view is loaded. A route doesn't have to be just a hardcoded string. It can also contain parameters, such as a back-end server router (Ruby on Rails, ASP.NET MVC, Django and so on). To do this, you name a route segment with a colon before the variable name you want.

You can now load the Customer edit view in the browser (localhost:25061/Home/Spa#/customer/edit/ANATR) and see the screen depicted in **Figure 17**.

Note: Although the delete (destroy) functionality on the Customer grid view has been wired up, when clicking the "Delete" button in the toolbar (see **Figure 18**), you'll see an exception, as shown in **Figure 19**.

This exception is by design, because most Customer IDs are foreign keys in other tables, for example, Orders, Invoices and so on. You'd have to wire up a cascading delete that would delete all records from all tables where Customer ID is a foreign key. Although you aren't able to delete anything, I still wanted to show the steps and code for the delete functionality.

So there you have it. I've demonstrated how quick and easy it is to convert an out-of-the-box ASP.NET Web Application into an SPA using RequireJS and Kendo UI. Then I showed how easy it is to add CRUD-like functionality to the SPA.

You can see a live demo of the project at bit.ly/1bkMAIK and you can see the CodePlex project site (and downloadable code) at easyspa.codeplex.com.

Happy coding! ■

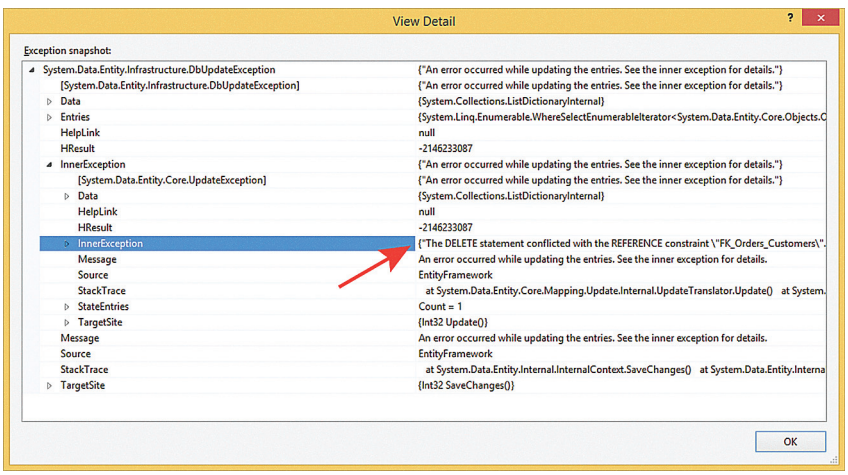


Figure 19 **Expected Exception When Deleting a Customer Due to CustomerID Foreign Key Referential Integrity**

Long Le *is the principal .NET app/dev architect at CBRE Inc. and a Telerik/Kendo UI MVP. He spends most of his time developing frameworks and application blocks, providing guidance for best practices and patterns and standardizing the enterprise technology stack. He has been working with Microsoft technologies for more than 10 years. In his spare time, he enjoys blogging (blog.longle.net) and playing Call of Duty. You can reach and follow him on Twitter at twitter.com/LeLong37.*

Thanks to the following technical experts for reviewing this article: *Derick Bailey (Telerik) and Mike Wasson (Microsoft)*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

LIVE AND LONG CODE

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA

LIVE! Visual Studio YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE LIVE! CHICAGO

This May, developers, software architects, engineers, and designers will blast off in the windy city for four days of unbiased and cutting-edge education on the Microsoft Platform. Live long and code with .NET gurus, launch ideas with industry experts and rub elbows with Microsoft stars in pre-conference workshops, 60+ sessions and fun networking events – all designed to make you better at your job. Plus, explore hot topics like Web API, jQuery, MongoDB, SQL Server Data Tools and more!

Tracks Include:

- Visual Studio/.NET Framework
- Windows Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint
- SQL Server

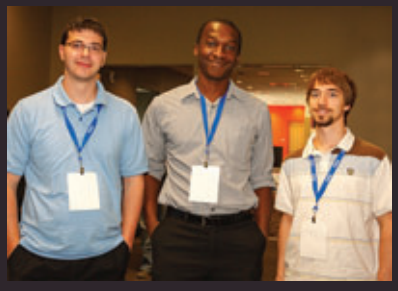
CHICAGO 2014

May 5 – 8 | Chicago Hilton

Register by April 2 and Save $200!

Use promo code VSLMAR4

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



TURN THE PAGE FOR MORE EVENT DETAILS

vslive.com/chicago

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

"SEVERAL OF THE PRESENTATIONS WERE CUTTING EDGE – THEY WOULD HAVE INSIDER TIPS THAT YOU CAN'T EASILY SEARCH FOR OR WOULDN'T KNOW TO LOOK FOR."

John Kilic
Web Application Developer
Grand Canyon University

AGENDA AT-A-GLANCE

Visual Studio / .NET Framework	Windows Client	Cloud Computing	Windows Phone	Cross-Platform Mobile Development

START TIME	END TIME	Visual Studio Live! Pre-Conference Workshops: Monday, May 5,	
7:30 AM	9:00 AM	Pre-Conference Workshop Registration	
9:00 AM	6:00 PM	**MW01** - Workshop: Modern UX Design - *Billy Hollis*	
6:00 PM	9:00 PM	Dine-A-Round Dinner	

START TIME	END TIME	Visual Studio Live! Day 1: Tuesday, May 6, 2014	
7:00 AM	8:00 AM	Registration	
8:00 AM	9:00 AM	Keynote: To Be Announced	
9:15 AM	10:30 AM	**T01** - What's New in WinRT Development - *Rockford Lhotka*	**T02** - What's New in the Visual Studio 2013 IDE
10:45 AM	12:00 PM	**T05** - What's New for XAML Windows Store Apps - *Ben Dewey*	**T06** - ALM with Visual Studio 2013 and Team Foundation Server 2013 - *Brian Randell*
12:00 PM	1:30 PM	Lunch - Visit Exhibitors	
1:30 PM	2:45 PM	**T09** - Interaction Design Principles and Patterns - *Billy Hollis*	**T10** - What's New for Web Developers in Visual Studio this Year? - *Mads Kristensen*
3:00 PM	4:15 PM	**T13** - Applying UX Design in XAML - *Billy Hollis*	**T14** - Why Browser Link Changes Things, and How You Can Write Extensions? - *Mads Kristensen*
4:15 PM	4:45 PM	Networking Break - Visit Exhibitors	
4:45 PM	6:00 PM	**T17** - What's New for HTML/WinJS Windows Store Apps - *Ben Dewey*	**T18** - Katana, OWIN, and Other Awesome Codenames: What's coming? - *Howard Dierking*
6:00 PM	7:30 PM	Exhibitor Welcome Reception	

START TIME	END TIME	Visual Studio Live! Day 2: Wednesday, May 7, 2014	
7:00 AM	8:00 AM	Registration	
8:00 AM	9:00 AM	Keynote: To Be Announced	
9:15 AM	10:30 AM	**W01** - Windows 8 HTML/JS Apps for the ASP.NET Developer - *Adam Tuliper*	**W02** - Creating Data-Driven Mobile Web Apps with ASPNET MVC and jQuery Mobile - *Rachel Appel*
10:45 AM	12:00 PM	**W05** - Developing Awesome 3D Applications with Unity and C#/JavaScript - *Adam Tuliper*	**W06** - Getting Started with Xamarin - *Walt Ritscher*
12:00 PM	1:30 PM	Birds-of-a-Feather Lunch & Exhibitor Raffle at 1:15pm MUST be present to win	
1:30 PM	2:45 PM	**W09** - What's New in WPF 4.5 - *Walt Ritscher*	**W10** - Building Multi-Platform Mobile Apps with Push Notifications - *Nick Landry*
3:00 PM	4:15 PM	**W13** - Implementing M-V-VM (Model-View-View Model) for WPF - *Philip Japikse*	W14 - Getting Started with Windows Phone Development - *Nick Landry*
4:30 PM	5:45 PM	**W17** - Build Maintainable Windows Store Apps with MVVM and Prism - *Brian Noyes*	W18 - Build Your First Mobile App in 1 Hour with Microsoft App Studio - *Nick Landry*
7:00 PM	9:00 PM	Blues after Dark at Buddy Guy's Legends	

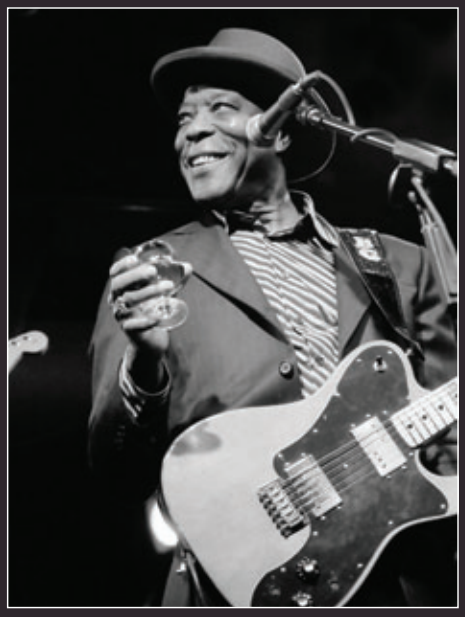
START TIME	END TIME	Visual Studio Live! Day 3: Thursday, May 8, 2014	
7:30 AM	8:00 AM	Registration	
8:00 AM	9:15 AM	**TH01** - Leveraging Windows Azure Web Sites (WAWS) - *Rockford Lhotka*	**TH02** - To Be Announced
9:30 AM	10:45 AM	**TH05** - Zero to Connected with Windows Azure Mobile Services - *Brian Noyes*	**TH06** - Essential C# 6.0 - *Mark Michaelis*
11:00 AM	12:15 PM	**TH09** - Building Services with ASP.NET MVC Web API Deep Dive - *Marcel de Vries*	**TH10** - Performance and Diagnostics Hub in Visual Studio 2013 - *Brian Peek*
12:15 PM	1:30 PM	Lunch	
1:30 PM	2:45 PM	**TH13** - Beyond Hello World: A Practical Introduction to Node.js on Windows Azure Websites - *Rick Garibay*	**TH14** - Visual Studio 2013 Release Manager: Reduce Your Cycle Time to Improve Your Value Delivery - *Marcel de Vries*
3:00 PM	4:15 PM	**TH17** - From the Internet of Things to Intelligent Systems: A Developer's Primer - *Rick Garibay*	**TH18** - Create Automated Cross Browser Testing of Your Web Applications with Visual Studio CodedUI - *Marcel de Vries*
4:30 PM	5:30 PM	Conference Wrap-Up Panel: *Andrew Brust, Miguel Castro, Rockford Lhotka, Ted Neward,*	

**Speakers and sessions subject to change*

CHICAGO 2014

May 5 – 8 | Chicago Hilton

ASP.NET	JavaScript / HTML5 Client	SharePoint	SQL Server
2014 (Separate entry fee required)			
MW02 - Workshop: Data-Centric Single Page Applications with Durandal, Knockout, Breeze, and Web API - *Brian Noyes*			**MW03** - Workshop: SQL Server for Developers - *Andrew Brust & Leonard Lobel*
	T03 - HTML5 for Better Web Sites - *Robert Boedigheimer*		**T04** - Introduction to Windows Azure - *Vishwas Lele*
	T07 - Great User Experiences with CSS 3 - *Robert Boedigheimer*		T08 - Windows Azure Cloud Services - *Vishwas Lele*
	T11 - Build Angular Applications Using TypeScipt - Part 1 - *Sergey Barskiy*		**T12** - Windows Azure SQL Database – SQL Server in the Cloud - *Leonard Lobel*
	T15 - Build Angular Applications Using TypeScipt - Part 2 - *Sergey Barskiy*		**T16** - Solving Security and Compliance Challenges with Hybrid Clouds - *Eric D. Boyd*
	T19 - Building Real Time Applications with ASP.NET SignalR - *Rachel Appel*		**T20** - Learning Entity Framework 6 - *Leonard Lobel*
	W03 - Leveraging Visual Studio Online - *Brian Randell*		**W04** - Programming the T-SQL Enhancements in SQL Server 2012 - *Leonard Lobel*
	W07 - JavaScript for the C# Developer - *Philip Japikse*		**W08** - SQL Server 2014: Features Drill-down - *Scott Klein*
	W11 - Build Data-Centric HTML5 Single Page Applications with Breeze - *Brian Noyes*		**W12** - SQL Server 2014 In-memory OLTP - Deep Dive - *Scott Klein*
	W15 - Knocking it Out of the Park, with Knockout.JS - *Miguel Castro*		**W16** - To Be Announced
	W19 - JavaScript: Turtles, All the Way Down - *Ted Neward*		**W20** - Building Apps for SharePoint - *Mark Michaelis*
TH03 - What's New in MVC 5 - *Miguel Castro*			**TH04** - Excel, Power BI and You: An Analytics Superhub - *Andrew Brust*
TH07 - What's New in Web API 2 - *Miguel Castro*			**TH08** - Big Data 101 with HDInsight - *Andrew Brust*
TH11 - Upgrading Your Existing ASP.NET Apps - *Pranav Rastogi*			**TH12** - NoSQL for the SQL Guy - *Ted Neward*
TH15 - Finding and Consuming Public Data APIs - *G. Andrew Duthie*			**TH16** - Git for the Microsoft Developer - *Eric D. Boyd*
TH19 - Provide Value to Customers and Enhance Site Stickiness By Creating an API - *G. Andrew Duthie & Brian Peek*			**TH20** - Writing Asynchronous Code Using .NET 4.5 and C# 5.0 - *Brian Peek*



Visual Studio Live! Chicago Blues After Dark Reception at Buddy Guy's Legends

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

REGISTER BY APRIL 2 AND SAVE $200!

Use Promo Code VSLMAR4

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

vslive.com/chicago

Building a Netduino-Based HID Sensor for WinRT

Donn Morse

The Human Interface Device (HID) protocol was originally intended to simplify the use of devices such as mice, keyboards and joysticks. However, because of its unique features—including its self-descriptive nature—device manufacturers use the protocol to support medical devices, health and fitness devices, and custom sensors. If you're new to the HID API, refer to the USB HID Information site (bit.ly/1mbtyTz) to find more information. Another great resource is Jan Axelson's book, "USB Complete: The Devloper's Guide, Fourth Edition" (Lakeview Research LLC, 2009).

Prior to Windows 8.1, if you were writing an application for a HID device you wrote a native Win32 app. But if you were a Web or a .NET developer, the ramp was steep. To address this, Microsoft introduced the HID Windows Runtime (WinRT) API with Windows 8.1 (bit.ly/1aot1by). This new API lets you write Windows Store apps for your device using JavaScript, Visual Basic, C# or C++.

In addition, Microsoft recently added support for several new transports, so you aren't limited to a USB cable. Today, you can create a HID device that transmits and receives packets over USB, Bluetooth, Bluetooth LE, and I2C. (For more information, see "HID Transports" at bit.ly/1asvwg6.)

In this article, I'll show how you can build a simple temperature sensor that's compatible with the HID protocol. Then I'll describe a sample Windows Store app that can display temperature data from the device.

This article discusses:

- Building a temperature-sensor device
- The beta firmware for the Netduino
- The sensor firmware
- The HID protocol
- The HID temperature-sensor app

Technologies discussed:

Windows 8.1, HID WinRT API, Microsoft .NET Micro Framework, C#, Netduino Board

Code download available at:

msdn.microsoft.com/magazine/msdnmag0314

Constructing the Temperature Sensor

The sample device is based on the Netduino development board (netduino.com). This open source board is used by hobbyists, academics and industrial engineers to build working prototype devices. And, because the Netduino is pin-compatible with the Arduino, you can attach your Arduino shields to quickly add functionality. (A shield is a board with specific functionality, such as wireless communication, motor control, Ethernet, RS232, LCD display and so on.) My sample device uses an RS232 shield to download the firmware. It uses the onboard USB connector to transmit and receive data.

The Netduino supports the .NET Micro Framework and its firmware is created with a free copy of Visual C# Express.

To obtain temperature data, the sample device uses the Texas Instruments LM35 sensor. The sensor takes 5 volts of input from

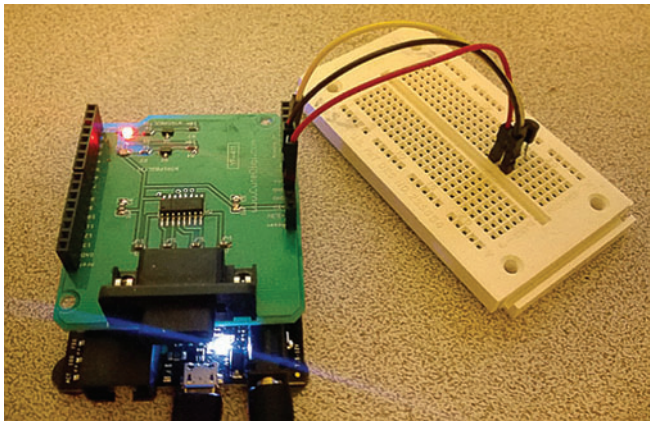


Figure 1 **The Complete HID Temperature Sensor Setup**

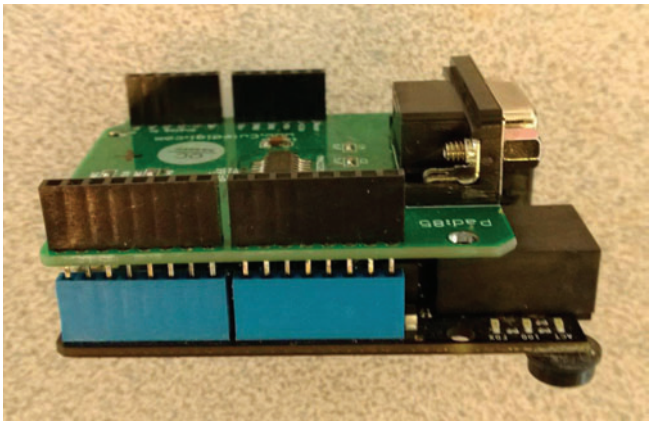


Figure 2 **Attaching the RS232 Shield to the Netduino**

the Netduino and converts it into an output voltage proportional to the current Celsius temperature.

Here are the parts you need to build your own HID sensor:

- Netduino 1 or Netduino Plus 1 (Amazon, amzn.to/1dvTeLh): A development board with programmable microcontroller that supports the .NET Micro Framework.
- RS232 shield (CuteDigi, bit.ly/1j7uaMR): The RS232 module for downloading and debugging the firmware. (This shield is required for the beta version of the firmware being used.)
- LM35 Sensor (DigiKey, bit.ly/KjbQkN): The temperature sensor that converts input voltage to output voltage based on the current temperature.
- RS232-to-USB converter cable (Parallax, bit.ly/1iVmP0a): The cable for downloading the temperature-sensor firmware via the RS232 shield. (Note that an FTDI chipset is required for compatibility with the shield.)
- 9V 650mA power supply (Amazon, amzn.to/1d6R8LH): The power supply for the Netduino board.
- USB to Micro-USB cable (Amazon, amzn.to/Kjc8li): The cable for sending HID packets from the Netduino to your Windows 8.1 tablet or laptop.

Figure 1 shows the complete HID temperature sensor setup.

The RS232 shield is attached to the top of the Netduino. The breadboard contains the LM35 sensor, which is attached to 5V, ground and Pin 0. (Pin 0 is one of six analog-to-digital [ADC] pins on the board). So, let's get started.

The firmware your Netduino 1 (or Netduino Plus 1) comes with doesn't support the HID protocol. You'll need to configure your development board by installing version 4.1.1 of the beta firmware, which includes support for HID. You'll find a zip folder containing the beta firmware at bit.ly/1a7f6MB. (You'll need to create an account by registering with Secret Labs in order to download the file.)

The download page on the Web site includes instructions for updating the firmware. However, these instructions are fairly complex, particularly if you're new to the Netduino. The video at bit.ly/1d73P9x is a helpful, concise description of the firmware upgrade process.

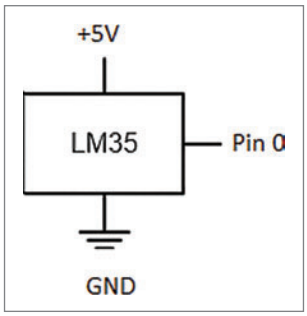


Figure 3 **The Sensor Pin-Outs**

After you've upgraded the firmware on your board, you're ready to begin constructing the temperature-sensor circuit. The first step requires you to attach the RS232 shield to your board. (As I already mentioned, the Netduino is pin-compatible with the Arduino, so if you've been working with the Arduino and have an RS232 shield handy, you can use it.) Snap the RS232 shield onto the Netduino as shown in **Figure 2**.

After you've attached the RS232 shield, the next step is to attach the temperature sensor to the 5V power source, ground and pin 0 of the Netduino. **Figure 3**, from the TI datasheet for the sensor, shows the pin-outs.

Installing the Sensor Firmware

There are two layers, or instances, of firmware on the Netduino. The first is the manufacturer's firmware, which includes the .NET Micro Framework; the second is your device's firmware. The manufacturer's firmware processes requests from the device firmware. The manufacturer's firmware is loaded once onto the development board and executes each time you power up the device. In contrast, you typically refresh your device firmware multiple times during the development and prototyping process.

In order to install any device firmware, you first need to install an instance of Visual C# Express 2010 on your development machine. You'll find a link to the download at bit.ly/1eRBed1.

For most Netduino projects, you can download and debug your firmware using the native USB connection. However, the beta version of the manufacturer's firmware requires an RS232 connection (which is why the RS232 shield is required).

Once Visual C# Express is installed, attach the RS232-to-USB cable and open Windows Device Manager to determine which COM port Windows assigned to that cable.

When I attached the Parallax RS232-to-USB converter to my development machine, Windows mapped it to COM6, as **Figure 4** shows.

Now that I know the COM port associated with the converter, I can power up my Netduino Plus 1, attach the RS232-to-USB cable, and start an instance of Visual C# Express to complete the download.

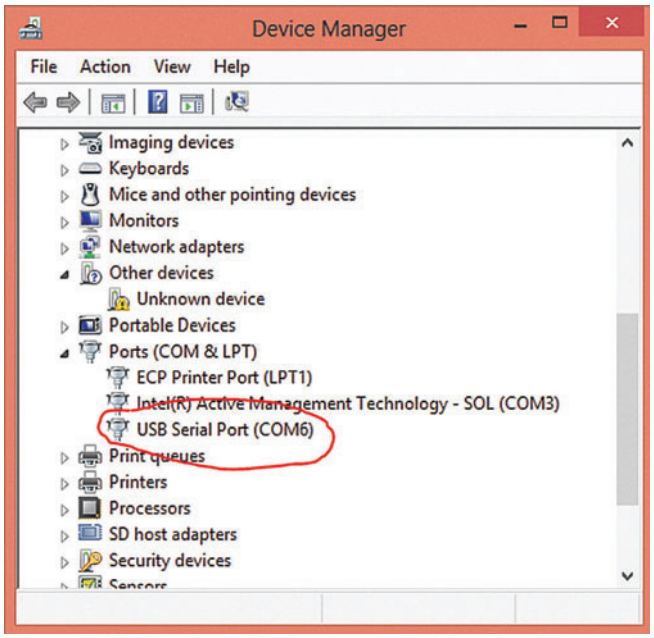


Figure 4 **The COM Port Assigned to the RS232-to-USB Cable**

The first thing to do after starting Visual C# Express is to identify the correct transport and COM port. You do this by right-clicking on the project name in the Solution Explorer pane and choosing the Properties menu.

When the Properties dialog appears, choose the .NET Micro Framework tab and make the necessary selections, as shown in **Figure 5**.

After specifying the Transport and Device, you can deploy the firmware. Again, right-click the project name in the Solution Explorer pane and, this time, choose Deploy.

When the deployment completes, Visual C# Express will report the success in the Output pane.

You're now ready to attach your device to a Windows 8.1 tablet or laptop and test it with the Custom Temperature Sensor sample app.

First, detach the RS232 cable, power down the Netduino, and then restart it with the auxiliary power supply. Give the device several seconds to power up and then attach the USB cable to the Netduino. After doing this, you should see your device added to the collection of HID devices in Device Manager. (The VID and PID in **Figure 6** correspond to the VID and PID of the sample device; these are the vendor and product IDs.)

Once the device is installed on your Windows 8.1 machine, you'll want to install and build the sample app. When the app starts, you can select the sensor and begin monitoring the ambient temperature in your office.

The Device Firmware

Now let's take a detailed look at the device firmware for the temperature sensor. At the outset, I'd like to thank the folks at Secret Labs (the manufacturers of the Netduino) for the work they've done to support HID over USB on the Netduino platform. The starting point for this firmware was a sample on the forum, the UsbHidEchoNetduinoApp, available at bit.ly/1eUYxAM.

Supporting the USB Transport As I noted earlier, Microsoft supports HID devices running over USB, Bluetooth, Bluetooth LE and I2C. However, the sample device described in this article uses the USB transport. What this actually means is that USB drivers will be moving packets in both directions: packets originating with the device are passed up to the HID driver (which passes them on to the API if there are interested apps); packets originating with the HID driver are passed back down to the device.

Windows uses specific data issued by the device upon connection to identify which USB drivers it should load.

> The first thing to do after starting Visual C# Express is to identify the correct transport and COM port.

Defining the Firmware Classes The firmware for the temperature-sensor device is built around two classes: Program and Sensor. The Program class supports a single Main routine that's invoked at startup. The Sensor class defines the USB and HID settings for the temperature sensor. In addition, it supports the methods that send input reports and read output reports.

The Sensor class contains all of the code required to configure the USB transport. This includes the code that:

- Configures a read endpoint
- Configures a write endpoint
- Specifies the vendor ID (VID)
- Specifies the product ID (PID)
- Specifies friendly names (manufacturer name, product name and so on)
- Specifies other required USB settings for a HID device

Most of the USB configuration code is found in the ConfigureHID method in the Sensors.cs module. This method, in turn, creates

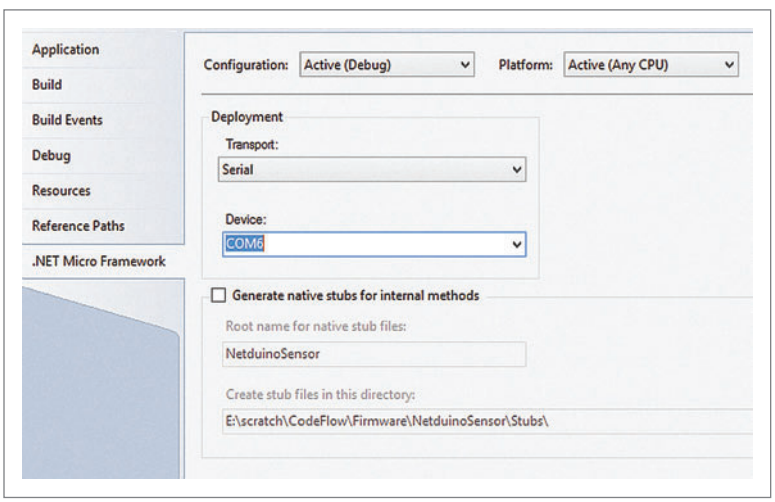
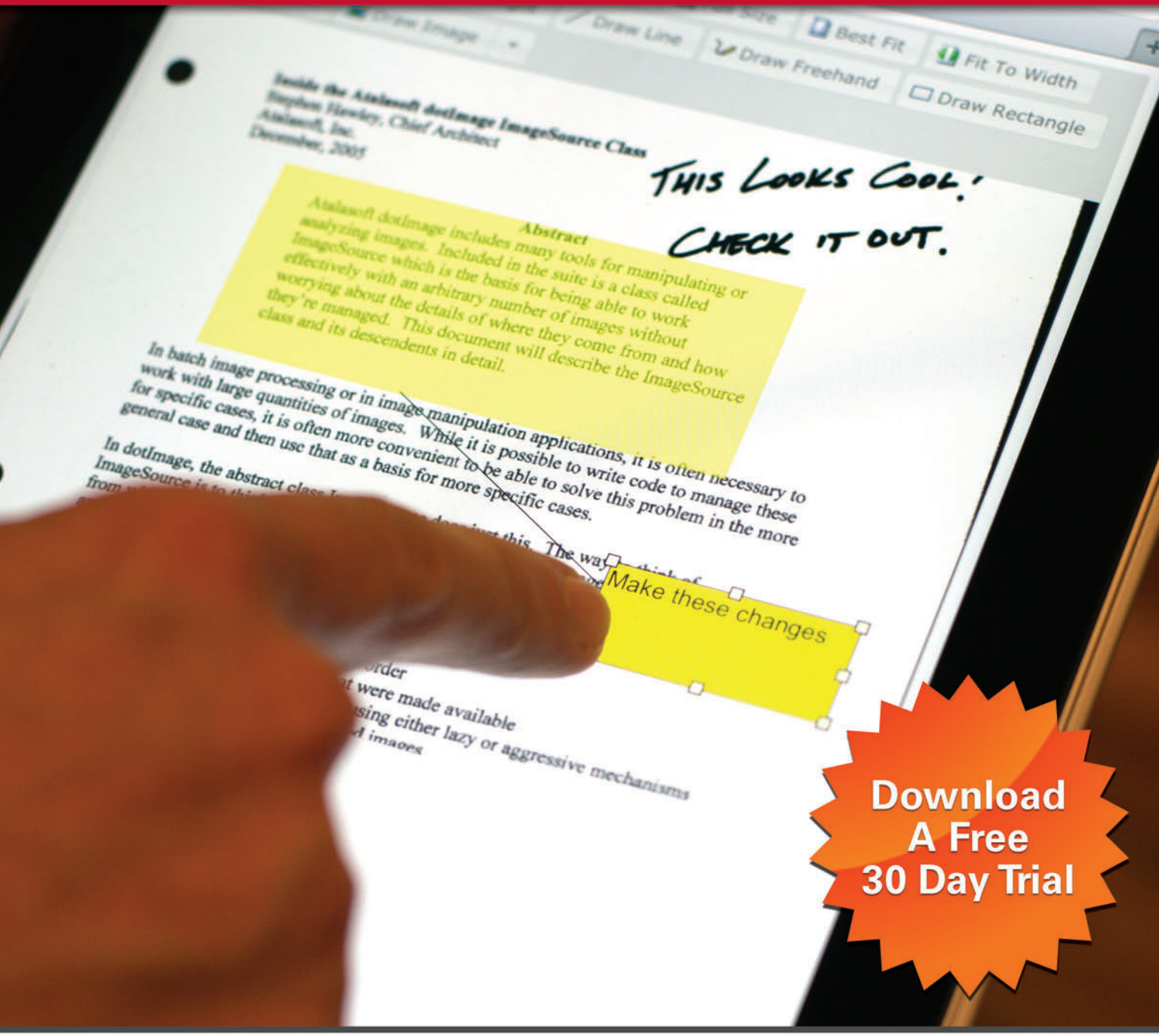


Figure 5 **Configuring the .NET Micro Framework Properties**

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



Download A Free 30 Day Trial

Atalasoft
A Kofax Company

.NET Web Scanning & Imaging SDKs

1.413.572.4443 ◆ www.atalasoft.com

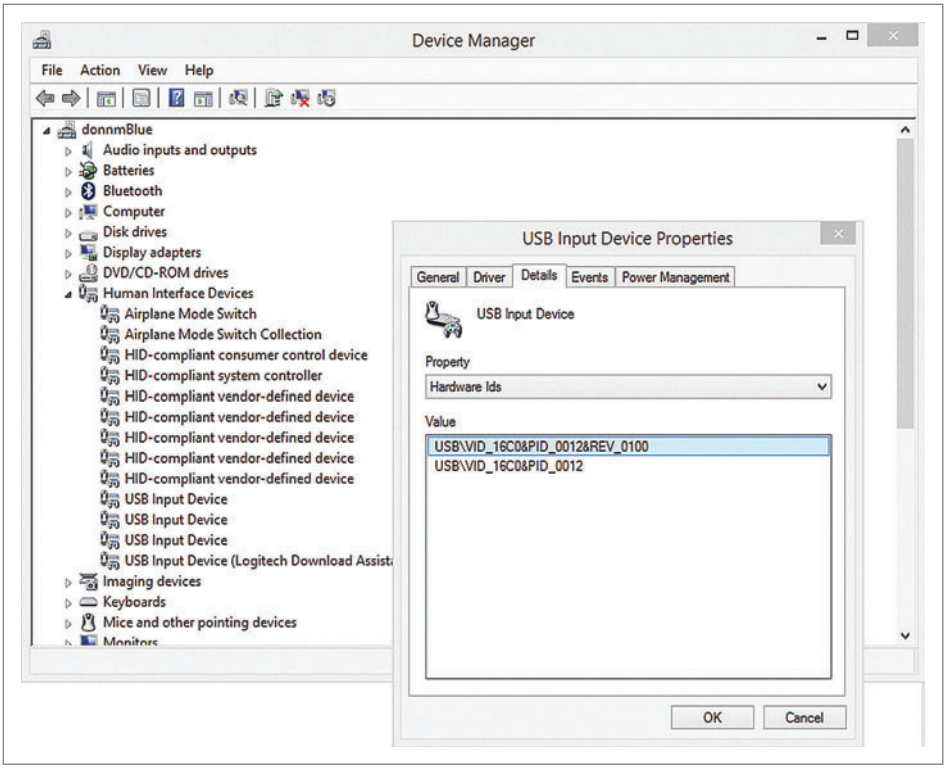


Figure 6 **The Vendor and Product IDs of the Sample Device**

and initializes a Configuration object (bit.ly/1i1IcQ3) that contains the device's USB settings (endpoints, VID, PID and so on).

The read endpoint allows the device to receive packets from the API and the HID driver. The write endpoint allows the driver to send packets up through the driver stack to the API.

Windows uses the vendor ID, product ID, and other USB settings (which were specified in the ConfigureHID method) to determine whether the device is a valid USB device and then to load the appropriate drivers.

Opening the Device Connection The Sensor class includes an Open method that's called from within the Main routine of the Program class. As you can see in **Figure 7**, the Open method:

- Retrieves the available USB controllers
- Invokes the ConfigureHID method to establish the device's USB and HID settings
- Invokes the Start method on the first available controller
- Creates a USB stream object with read and write endpoints

The Sensor class also includes a Close method, which is called when the device is detached from the host laptop or tablet.

Supporting the HID Protocol

The HID protocol is based on reports: feature reports, input reports and output reports. Feature reports can be sent by either the host (that is, a connected laptop or tablet) or the device. Input reports are sent by the device to the host. Output reports are sent by the host to the device.

In the case of our sample temperature sensor, the input report is a very simple two-byte packet. The first byte specifies the current temperature in degrees Fahrenheit; the second byte indicates the current report interval in milliseconds. (The sensor firmware issues an input report at the frequency specified by the report interval.)

The output report for the sample device is even simpler—it's a single byte that specifies the report interval. (This is an integer value that represents the interval in milliseconds.)

Creating the Report Descriptor As I mentioned earlier, one of the features of a HID device is its self-reporting nature: Upon connecting to a host, the device provides a description of its purpose, capabilities and packet format in what's called a report descriptor. This descriptor indicates where the device fits in the HID universe (is it a mouse, a keyboard, a vendor-defined device?). The descriptor also specifies the format of the individual feature reports, input reports and output reports.

The report descriptor for the temperature sensor is found in Sensors.cs, as shown in **Figure 8**.

The first two lines of the descriptor inform the host that this particular device is vendor-defined:

```
0x06,0x55,0xFF,     //HID_USAGE_PAGE_VENDOR_DEFINED
0x09,0xA5,          //HID_USAGE (vendor_defined)
```

Lines four through 15 indicate the format of the two-byte input report. Lines four through nine describe the first byte of the input report, which specifies the temperature reading:

```
0x09,0xA7,          //HID_USAGE (vendor_defined)
0x15,0x00,          //HID_LOGICAL_MIN_8(0), // Minimum temp is 0 degrees F
0x25,0x96,          //HID_LOGICAL_MAX_8(150), // Max supported temp is
                                              // 150 degrees F
0x75,0x08,          //HID_REPORT_SIZE(8),
0x95,0x01,          //HID_REPORT_COUNT(1),
0x81,0x02,          //HID_INPUT(Data_Var_Abs),
```

> The HID protocol is based on reports: feature reports, input reports and output reports.

The 10th through 15th lines describe the second byte of the input report, which specifies the report interval (in milliseconds):

```
0x09,0xA8,          //HID_USAGE (vendor_defined)
0x15,0x4B,          //HID_LOGICAL_MIN_8(75),   // minimum 75 ms
0x25,0xFF,          //HID_LOGICAL_MAX_8(255),  // maximum 255 ms
0x75,0x08,          //HID_REPORT_SIZE(8),
0x95,0x01,          //HID_REPORT_COUNT(1),
0x81,0x02,          //HID_INPUT(Data_Var_Abs),
```

The report descriptor for the sample device is included as part of the UsbController.Configuration object (bit.ly/1cvcq5G) that's created within the ConfigureHID method in Sensor.cs.

Spreadsheets Made Easy.

Fastest Calculations

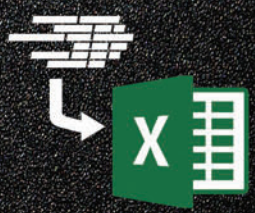
Evaluate complex Excel-based models and business rules with the fastest and most complete Excel-compatible calculation engine available.

Powerful Controls

Add powerful Excel-compatible viewing, editing, formatting, calculating, filtering, sorting, charting, printing and more to your WinForms, WPF and Silverlight applications.

Comprehensive Charting

Enable users to visualize data with comprehensive Excel-compatible charting which makes creating, modifying, rendering and interacting with complex charts easier than ever before.



Scalable Reporting

Easily create richly formatted Excel reports without Excel from any ASP.NET, Windows Forms, WPF or Silverlight application.

Download your free fully functional evaluation at SpreadsheetGear.com

SpreadsheetGear

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

Figure 7 **The Open Method**

```
public bool Open()
{
  bool succeed = true;

  started = false;

  UsbController[] usbControllers = UsbController.GetControllers();

  if (usbControllers.Length < 1)
  {
    succeed = false;
  }

  if (succeed)
  {
    usbController = usbControllers[0];

    try
    {

      succeed = ConfigureHID();

      if (succeed)
      {
        succeed = usbController.Start();
      }

      if (succeed)
      {
        stream = usbController.CreateUsbStream(WRITE_ENDPOINT, READ_ENDPOINT);
      }
    }
    catch (Exception)
    {
      succeed = false;
    }
  }

  started = true;
  return succeed;
}
```

Figure 8 **The Report Descriptor for the Temperature Sensor**

```
  hidGenericReportDescriptorPayload = new byte[]
  {
    0x06,0x55,0xFF,     //HID_USAGE_PAGE_VENDOR_DEFINED
    0x09,0xA5,          //HID_USAGE (vendor_defined)

    0xA1,0x01,          //HID_COLLECTION(Application),

    // Input report (device-transmits)
    0x09,0xA7,          //HID_USAGE (vendor_defined)
    0x15,0x00,          //HID_LOGICAL_MIN_8(0), // Minimum temp is 0 degrees F
    0x25,0x96,          //HID_LOGICAL_MAX_8(150), // Max supported temp is
                                                  // 150 degrees F
    0x75,0x08,          //HID_REPORT_SIZE(8),
    0x95,0x01,          //HID_REPORT_COUNT(1),
    0x81,0x02,          //HID_INPUT(Data_Var_Abs),

    0x09,0xA8,          //HID_USAGE (vendor_defined)
    0x15,0x4B,          //HID_LOGICAL_MIN_8(75),   // minimum 75 ms
    0x25,0xFF,          //HID_LOGICAL_MAX_8(255),  // maximum 255 ms
    0x75,0x08,          //HID_REPORT_SIZE(8),
    0x95,0x01,          //HID_REPORT_COUNT(1),
    0x81,0x02,          //HID_INPUT(Data_Var_Abs),

    // Output report (device-receives)
    0x09,0xA9,          //HID_USAGE (vendor_defined)
    0x15,0x4B,          //HID_LOGICAL_MIN_8(75),   // minimum 75 ms
    0x25,0xFF,          //HID_LOGICAL_MAX_8(255),  // maximum 255 ms
    0x75,0x08,          //HID_REPORT_SIZE(8),
    0x95,0x01,          //HID_REPORT_COUNT(1),
    0x91,0x02,          //HID_OUTPUT(Data_Var_Abs),

    0xC0                //HID_END_COLLECTION

   };
```

Supporting the HID Input Report The input report is defined as a structure in the Sensor.cs module:

```
struct InputReport
{
  public byte Temperature; // Temperature in degrees Fahrenheit
  public byte Interval;    // Report interval (or frequency) in seconds
}
```

The firmware issues input reports using the UsbStream object (bit.ly/1kElfUZ) it created in the Open method. These input reports are issued from the SendInputReport method when the firmware invokes the stream.Write method:

```
protected void SendInputReport(InputReport report)
{

  byte[] inputReport = new byte[2];

  inputReport[0] = (byte)report.Temperature;
  inputReport[1] = (byte)report.Interval;

  stream.Write(inputReport, 0, 2);
}
```

Issuing Temperature Data with Input Reports The Update method in the Sensor class issues an input report to the connected host:

```
public int Update(int iTemperature, int iInterval)
{
  InputReport inputReport = new InputReport();
  byte Interval = 0;

  inputReport.Temperature = (byte)iTemperature;
  inputReport.Interval = (byte)iInterval;

  SendInputReport(inputReport);

  Interval = GetOutputReport();
  return (int)Interval;
}
```

The Update method is invoked from within an infinite while loop, shown in **Figure 9**, which executes in the firmware's Main routine (found in Program.cs).

> This new API lets your app retrieve data from HID devices, and control them as well.

Supporting the HID Output Report The output report is defined as a structure in the Sensor.cs module:

```
struct OutputReport
{
  public byte Interval; // Report interval (or frequency) in seconds
}
```

The firmware receives output reports via the same UsbStream object it created in the Open method. These output reports are received within the GetOutputReport method:

```
protected byte GetOutputReport()
{
  byte[] outputReport = new byte[1];
  int bytesRead = 0;
  if (stream.CanRead)
  {
  bytesRead = stream.Read(outputReport, 0, 1);
  }
  if (bytesRead > 0)
  return outputReport[0];
  else
  return 0;
}
```

Figure 9 **The While Loop That Invokes the Update Method**

```
while (true)
{
  // Retrieve the current temperature reading

  milliVolts = (double)voltsPin.Read();  // Read returns a value in the
                                         // specified range
  tempC = milliVolts / 10.0;             // Sensor returns 10mV per
                                         // degree Centigrade
  tempF = 1.8 * tempC + 32;              // Convert to degrees Fahrenheit
  simpleTemp = (int)tempF;

  // Because there are voltage fluctuations when the external
  // power supply is connected to the Netduino, use a running
  // average to "smooth" the values

  if (firstReading)
  {
    firstReading = false;
    currentTemp = simpleTemp;
    for (i = 0; i < 12; i++)
      tempArray[i] = simpleTemp;

    }
  else
  {
    tempArray = Shift(simpleTemp, tempArray);  // Shift the array elements and
                                               // insert the new temp
    currentTemp = Average(tempArray);          // Compute a running average of
                                               // the last 12 readings
  }

  RequestedInterval = sensor.Update(currentTemp, CurrentInterval);

  // Check for a possible new interval requested via an
  // output report

  if (RequestedInterval != 0)
  {
    CurrentInterval = RequestedInterval;
  }

  led.Write(true);
  Thread.Sleep(CurrentInterval);
  led.Write(false);
  }
}
```

Adjusting the Report Interval with Output Reports The firmware supports a report interval specified in milliseconds. The minimum supported interval is 75 ms; the maximum interval is 255 ms. An app requests a new interval by sending an output report to the device. The device, in turn, reports the current interval in each input report that it sends to any connected app.

The firmware applies the current interval by invoking the Thread.Sleep method (bit.ly/LaSYVF) for the number of seconds specified by the current interval:

```
led.Write(true);
Thread.Sleep(CurrentInterval);
led.Write(false);
```

By pausing the while loop for this duration, registered apps receive input reports at the specified interval.

The HID Temperature-Sensor App

The sample app demonstrates how you can display temperature data from an attached HID temperature sensor using the new HID WinRT API for Windows 8.1. This new API lets your app retrieve data from HID devices, and control them as well.

The sample is designed to work with an attached HID device that detects temperatures from 0 to 150 degrees Fahrenheit. The app monitors and then displays the temperature sensor's current reading.

The app supports three "scenarios," each of which maps to specific features in the app's UI. In addition, each scenario maps to a corresponding XAML and C# source file. The following lists each scenario, its corresponding modules and its function:

Device Connect (Scenario1_ConnectToSensor.xaml; Scenario1_ConnectToSensor.xaml.cs)

- Supports connecting a HID device to a Windows 8.1 PC.
- Enumerates the connected temperature sensors so the user can select one.
- Establishes a device watcher that monitors the status of the device. (The device watcher fires an event when the user disconnects or reconnects the selected HID device.)

Get Temperature Data (Scenario2_GetTemperatureData.xaml; Scenario2_GetTemperatureData.xaml.cs)

- Monitors the selected temperature sensor.
- Depicts a temperature gauge and renders the current reading using a slider control.

Set Report Interval (Scenario3_SetReportInterval.xaml; Scenario3_SetReportInterval.xaml.cs)

- Allows the user to control the frequency at which the temperature sensor reports its status. (The default interval is 250 ms, but users can choose intervals from 75 ms to 250 ms.)

Supporting Device Connections

The device-connect scenario enables several aspects of connecting a HID device to a Windows 8.1 PC: enumerating connected devices, establishing a device watcher, handling device disconnection and handling device reconnection.

Establishing a Device Connection The code that handles the device connection is found in three modules: Scenario1_ConnectToSensor.xaml.cs, EventHandlerForDevices.cs and DeviceList.cs. (The first module contains the primary code for this scenario; the other two contain supporting functionality.)

The first phase of the connection occurs before the UI is visible. In this phase, the app creates a DeviceWatcher object that notifies the app when devices are added, removed or changed. The second phase

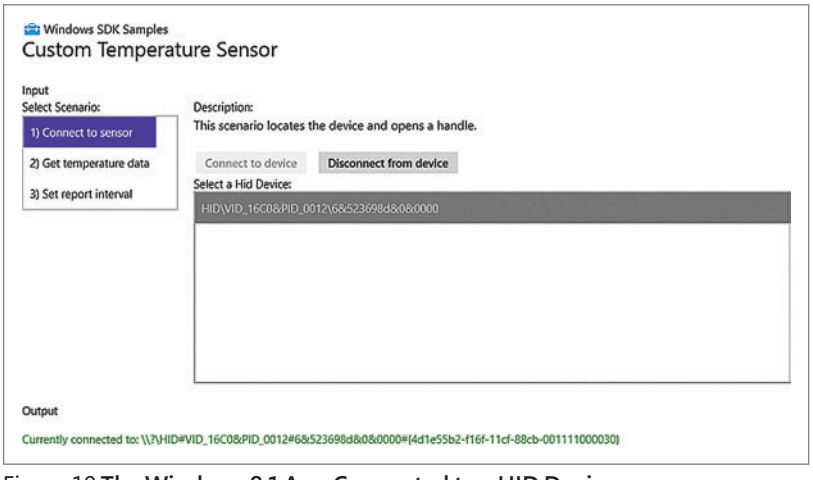


Figure 10 **The Windows 8.1 App Connected to a HID Device**

Figure 11 **The InitializeDeviceWatchers Method**

```
private void InitializeDeviceWatchers()
{
  // Custom sensor
  var CustomSensorSelector =
    HidDevice.GetDeviceSelector(CustomSensor.Device.UsagePage,
    CustomSensor.Device.UsageId, CustomSensor.Device.Vid,
    CustomSensor.Device.Pid);

  // Create a device watcher to look for instances of the custom sensor
  var CustomSensorWatcher =
    DeviceInformation.CreateWatcher(CustomSensorSelector);

  // Allow EventHandlerForDevice to handle device watcher events that
  // relate or affect the device (device removal, addition, app
  // suspension/resume)
  AddDeviceWatcher(CustomSensorWatcher, CustomSensorSelector);
}
```

occurs after the UI is displayed and the user is able to choose a specific device from the connected HID devices. The app displays a DeviceInstanceId string for each connected device; the string includes the VID and PID for the given device. In the case of the sample temperature sensor, the DeviceInstanceId string has the form:

```
HID\VID_16C0&PID_0012\6&523698d&0&0000
```

Figure 10 shows the app as it appears after enumeration has completed and the user has connected to the device.

The First Stage of Device Connection Here are the methods called during the first stage of device connection (before the UI is displayed), along with tasks accomplished by each method:

DeviceConnect (Scenario1_DeviceConnect.xaml.cs) invokes the CustomTemperatureSensor.InitializeComponent method, which initializes the app's UI components such as the text blocks and buttons.

InitializeDeviceWatchers (Scenario1_DeviceConnect.xaml.cs) invokes the HidDevice.GetDeviceSelector method to retrieve a device selector string. (The selector is required in order to create a device watcher.) Once the selector is obtained, the app invokes DeviceInformation.CreateWatcher to create the DeviceWatcher object and then EventHandlerForDevice.Current.AddDeviceWatcher. (This last method allows the app to monitor changes in device status.)

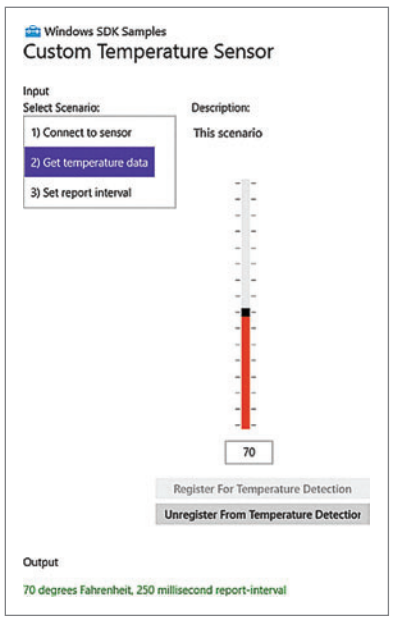


Figure 12 **Displaying the Current Temperature**

AddDeviceWatcher (EventHandlerForDevices.cs) creates the event handlers for three device events: Enumeration Completed, Device Added and Device Removed.

SelectDeviceInList (Scenario1_DeviceConnect.xaml.cs) checks to see if the user has selected a device and, if so, it saves the index for that device.

In terms of the HID API, the primary code of interest is found in the InitializeDeviceWatchers method, shown in **Figure 11**. This code invokes the HidDevice.GetDeviceSelector method (bit.ly/1eGQl1k) and passes the UsagePage, UsageId, VID and PID for the temperature sensor.

The UsagePage and UsageId values are defined in the file constants.cs:

```
public class Device
{
  public const UInt16 Vid = 0x16C0;
  public const UInt16 Pid = 0x0012;
  public const UInt16 UsagePage = 0xFF55;
  public const UInt16 UsageId = 0xA5;
}
```

These class members correspond to values specified in the HID report descriptor that's defined in the device's firmware:

```
hidGenericReportDescriptorPayload = new byte[]
{
  0x06,0x55,0xFF,     //HID_USAGE_PAGE_VENDOR_DEFINED
  0x09,0xA5,          //HID_USAGE (vendor_defined)
```

The GetDeviceSelector method returns an Advanced Query Syntax (AQS) string in the CustomSensorSelector variable. The app then uses this string when it creates a device watcher and when it enumerates the DeviceInformation objects.

The Second Stage of Device Connection The second stage of device connection allows the user to make a selection from the list of connected devices. This stage establishes the currently selected device. Here are the methods (all in EventHandlerForDevices.cs) called and what each one does.

OpenDeviceAsync opens the connection to the device.

RegisterForAppEvents registers for app suspension and resume events.

RegisterForDeviceAccessStatusChange listens for changes in device-access permissions.

RegisterForDeviceWatcherEvents registers for the added and removed events.

StartDeviceWatcher starts the device watcher.

SelectDeviceInList checks to see if the user has selected a device and, if so, saves the index for that device. It also writes a "Currently connected …" string to the output window if the connection is successful.

Figure 13 **Reading and Displaying Sensor Data**

```
private async void OnInputReportEvent(HidDevice sender,
  HidInputReportReceivedEventArgs eventArgs)
{
  // Retrieve the sensor data
  HidInputReport inputReport = eventArgs.Report;
  IBuffer buffer = inputReport.Data;
  DataReader dr = DataReader.FromBuffer(buffer);
  byte[] bytes = new byte[inputReport.Data.Length];
  dr.ReadBytes(bytes);

  // Render the sensor data
  await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
  {
    CurrentReadingText.TextAlignment = TextAlignment.Center;
    CurrentReadingText.Text = bytes[1].ToString();
    TemperatureSlider.Value = (int)bytes[1];

    rootPage.NotifyUser(bytes[1].ToString() + " degrees Fahrenheit, " +
      bytes[2].ToString() +
      " millisecond report-interval", NotifyType.StatusMessage);
  });
}
```

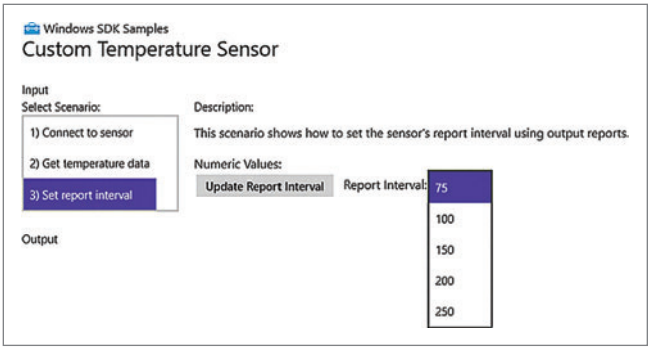


Figure 14 **Setting the Report Interval**

In terms of the HID API, the primary code of interest is in the OpenDeviceAsync method. This code invokes the HidDevice.FromIdAsync method (bit.ly/1hyhVpI), which returns a HidDevice object (bit.ly/1dsD2rR) that the app uses to access the device, retrieve input reports and send output reports:

```
public async Task<Boolean> OpenDeviceAsync(DeviceInformation deviceInfo,
  String deviceSelector)
  {
    // This sample uses FileAccessMode.ReadWrite to open the device
    // because you don't want other apps opening the device and
    // changing the state of the  device.
    // FileAccessMode.Read can be used instead.
    device = await HidDevice.FromIdAsync(deviceInfo.Id, FileAccessMode.ReadWrite);
...
```

Supporting the Device Watcher Device enumeration occurs when the app is first started and begins even before the UI is displayed. After enumeration completes, the app monitors device status.

Device status is reported by a DeviceWatcher object (bit.ly/1dBPMPd). As the name implies, this object "watches" the connected devices—if the user removes or connects his device, the watcher reports the event to the app. (These events are only reported after the enumeration process is finished.)

Retrieving Input Reports

The temperature-retrieval scenario monitors the input reports issued by the temperature-sensor and uses a slider control to display the current temperature, as shown in **Figure 12**. (Note that this control is limited to displaying temperature data. The properties IsDoubleTapEnabled, IsHoldingEnabled, IsRightTapEnabled and IsTapEnabled have all been set to False.)

Figure 15 **SetReportIntervalAsync**

```
private async Task SetReportIntervalAsync(Byte valueToWrite)
{
  var outputReport =
    EventHandlerForDevice.Current.Device.CreateOutputReport();
  var dataWriter = new DataWriter();

  // First byte contains the report id
  dataWriter.WriteByte((Byte)outputReport.Id);
  dataWriter.WriteByte((Byte)valueToWrite);
  outputReport.Data = dataWriter.DetachBuffer();

  uint bytesWritten =
    await EventHandlerForDevice.Current.Device.
    SendOutputReportAsync(outputReport);

  rootPage.NotifyUser("Bytes written:  " + bytesWritten.ToString() + ";
    Value Written: " + valueToWrite.ToString(), NotifyType.StatusMessage);
}
```

The primary method supporting this scenario is the OnInputReportEvent event handler, found in Scenario2_GetTemperatureData.xaml.cs. The app registers this event handler when the user chooses the Get temperature data scenario and presses the Register for Temperature Detection button. The app registers the event handler within the RegisterForInputReportEvents method. In addition to registering the handler, this method saves an event token so it can unregister.

```
private void RegisterForInputReportEvents()
{
  if (!isRegisteredForInputReportEvents)
  {
    inputReportEventHandler = new TypedEventHandler<HidDevice,
      HidInputReportReceivedEventArgs>(this.OnInputReportEvent);

    registeredDevice = EventHandlerForDevice.Current.Device;

    registeredDevice.InputReportReceived += inputReportEventHandler;

    isRegisteredForInputReportEvents = true;
  }
}
```

Once the event handler is registered, it reads each input report issued by the sensor and uses the new temperature value to update the TemperatureSlider control. After it updates the control, this method writes the current temperature and report interval values to the Output section of the app, as shown in **Figure 13**.

Sending Output Reports

The report-interval scenario sends an output report to the temperature sensor and writes the count of bytes as well as the value written to the Output area of the app's window. The app sends an output report after the user chooses the Set report interval scenario, selects a value from the Value to Write dropdown, and then presses the Send Output Report button.

Figure 14 shows the Set report interval scenario and the dropdown that's populated with the report interval options. (These values represent a report-interval in milliseconds; so, by selecting 100, the app will receive 10 readings every second.)

The primary method of the report-interval scenario is SetReportIntervalAsync, found in the Scenario3_SetReportInterval.xaml.cs module (see **Figure 15**). This method invokes the HidDevice.SendOutputReportAsync method (bit.ly/1ad6unK) to send an output report to the device.

Wrapping Up

First I gave you a quick look at building a HID device that monitors the voltage emitted by a simple sensor. For an example of how you could monitor a sensor that toggles a digital I/O pin (rather than emitting a range of voltages), see the motion-sensor sample on MSDN at bit.ly/1gWOlcC.

Then you took a quick look at writing a simple app that monitors and controls a HID device. For more information about the HID WinRT API, visit bit.ly/1aot1by. ■

Donn Morse *is a senior content developer at Microsoft. Reach him at donnm@microsoft.com.*

THANKS to the following technical expert for reviewing this article: *Arvind Aiyar (Microsoft)*

TED NEWARD

Getting Started with Oak: Data Validation and Wrapping Up

For three columns now, I've been exploring the "dynamic-y" object approach that Oak brings to the Web application space, and it's been an interesting ride, complete with a few challenges to long-held beliefs about how Web applications need to be built (or about the platform on which they're built). But every ride has to come to an end sometime, and it's about time to wrap up my exploration of Oak. I've got to figure out how to ensure data put into the system by the user is actually good data, for starters.

But first ...

Commentary

If you go back and look at the system as I left off last time, trying to add a comment yields another of those helpful errors, this time informing you that "Blog.Controllers.Blog does not contain a definition for 'AddComment.'" Contrary to what might happen in a statically typed system—where the lack of this method would trip a compilation error at the front of the compile/deploy/run cycle—in a dynamic system, these errors won't be seen until they're actually attempted. Some dynamic-language proponents claim this is part of the charm of dynamic languages, and certainly not having to worry about keeping everything consistent across the entire system can be a great boon when the mind is on fire with an idea and you just need to get that idea out into the world. But most Ruby-on-Rails developers I know who've done a project larger than your typical Todo list application will be the first to admit that in a dynamic-language application, comprehensive tests are critical to keeping the project's quality high and the developer's sanity strong. So testing has to be a part of any serious effort with Oak.

> Notice how the dynamic nature of the system lets you be extremely frugal in the code.

Unfortunately, as soon as I start talking about testing, I start getting into several areas of discussion (unit tests, behavior tests, integration tests, Test-Driven Development and so on) that could easily consume another half-dozen magazine issues on their own, and I don't want to crack that Pandora's box here. Whatever your testing methodology or preference, suffice it to say that you must have some kind of testing presence in an Oak application (or any application, but the need is much higher in any dynamically typed environment), however you choose to test.

Meanwhile, the AddComment method is still missing.

Comment Away

In this particular case, when the user types a comment into the view, it POSTs to the HomeController Comments method, which looks like so:

```
[HttpPost]
public ActionResult Comments(dynamic @params)
{
  dynamic blog = blogs.Single(@params.BlogId);

  blog.AddComment(@params);

  return RedirectToAction("Index");
}
```

As you can see, the controller is first obtaining the blog ID tucked away in the BlogId parameter coming from the form, then using it to find the corresponding blog entry via the Single method on the DynamicRepository, then calling blog.AddComment to store the comment. (Again, just to make the points, both "pro" and "con": This code has been in place since the second part of this series, and I'm just now running into the fact that the AddComment method hasn't existed until now.)

Defining this method is pretty straightforward; on the Blog class, add this method:

```
void AddComment(dynamic comment)
{
  // Ignore addition if the body is empty
  if (string.IsNullOrEmpty(comment.Body)) return;

  // Any dynamic property on this instance can be accessed
  // through the "_" property
  var commentToSave = _.NewComment(comment);

  comments.Insert(commentToSave);
}
```

The only real question mark in this method is the use of the underscore (_.NewComment(comment)), which is a placeholder for the "this" reference. The underscore has full awareness of the dynamic nature of this object, which the "this" reference wouldn't; rather than having to worry about the differences, the underscore lets you use everything "this" would, plus more.

Notice how the dynamic nature of the system lets you be extremely frugal in the code. The form parameters are captured in a named bundle in "@params" coming in to the controller, and those are passed without unpacking any of them directly into AddComment, which in turn passes them into NewComment, which

constructs a dynamic object out of them, and the resulting object just gets inserted into the comments DynamicRepository. Where do the names of the object's properties come from? From the HTML form that originated all of this.

Wacky, huh? Almost feels like you're being lazy or something.

Anyway, give it a run, and sure enough, now comments are being added in.

Validate Me

As written, though, the system has a major flaw (well, according to user requirements, anyway): It's perfectly permissible for two blog entries to have the exact same title, and that's not OK. (Readers might get confused as to which one to read, the one titled "LOL Kittehs" or the other one titled "LOL Kittehs.") Thus, you need a way of enforcing some kind of uniqueness on the code, so users can't accidentally put in duplicate-titled blog entries.

In a traditional Web framework, this would be a two-part job. First, the model object (Blog) would have to define some kind of "ask me if I'm valid" method, which I'll call IsValid for lack of anything really original, and a definition of said validation within the object, which I'll call Validates. And, as one might suspect from the way the Associates method worked to help define the link between Blog and Comment objects (meaning, it's a predefined name for which Oak knows to look), the Validates method works in the same way—if an object defines a Validates method, then Oak will call the already-defined IsValid method on the object, which in turn will look for a Validates method and ask it for all the conditions that make this object valid.

In code, that looks like this:

```
IEnumerable<dynamic> Validates()
{
  // And define the association
  // For other examples of validations, check out the Oak wiki
  yield return new Uniqueness("Name", blogs);
}
```

Again, you see the use of a stream of objects that describe the validation requirements, handed back as an IEnumerable<dynamic> to the stream, generated through the use of the "yield return" facility of C#. And, as with the Schema class from last time, the way to extend this is to just tack on additional elements that are "yield returned," like so:

```
IEnumerable<dynamic> Validates()
{
  // And define the association
  // For other examples of validations check out the Oak wiki
  yield return new Uniqueness("Name", blogs);
  yield return new Length("Name") { Minimum=10, Maximum=199 };
}
```

The Oak Wiki defines the full list and usage of validation objects, but some of the notable ones are:

- **Presence:** A field is not optional and must be present on the object.
- **Acceptance:** A field on the object must contain a particular value, such as a LegalDocument object containing a TypedOutAcceptance field in which the user typed the string "I Accept" in order to indicate he accepted the legal restrictions.
- **Exclusion:** A field can't include certain values.
- **Inclusion:** A field must be one of a set of certain values.
- **Format:** The general-purpose regular-expression (using the Microsoft .NET Framework Regex class) validation.
- **Numericality:** Pretty much everything to do with numeric values, including marking a field as "integer-only," greater-than and less-than restrictions, or simple even/odd tests.
- **Conditional:** The catch-all "escape hatch" for any validation not covered elsewhere—a field must satisfy a condition described using a lambda function.

The last one, Conditional, isn't actually a validation type in and of itself, but a feature present on most (if not all) of the other validation types, and therefore deserves a little more explanation. Imagine an Order object, for orders in a traditional e-commerce system. For said systems, a credit card number is only necessary if the user wants to use a credit card for payment. Similarly, an address to which to ship the product is only necessary if the user purchased anything other than a digital download. These two contingencies are neatly expressed using two conditional validations, as shown in **Figure 1**.

> While it's usually better to use Oak as a whole, it does support the idea of ripping out bits of it.

Each of the Conditional objects is making use of a property on the Presence object—along with a lambda that yields a true/false value—to indicate whether the Presence validates successfully. In the first case, Presence returns true (pass) if d.PaidWithCard, a local method that returns true if the PaymentType field equals "Card," returns true. In the second case, Presence returns true unless isDigitalPurchase returns true, meaning that if it's a digital item, no address is necessary.

All of these are ready for use with any Oak DynamicModel-derived object, and, as noted in the prior column (msdn.microsoft.com/magazine/dn519929) and in the introduction of this one, the DynamicModel-derived object needn't explicitly define the fields that these validations reference. Should these validations not be sufficient to the task, by the way, these are all defined in the Validations.cs file inside the Oak folder of the scaffolded project. It's pretty straightforward to define a new one if desired: Just inherit from Oak.Validation and define at minimum a Validate method that returns true/false. The Exclusion validation, for example, is this:

```
public class Exclusion : Validation
{
  public Exclusion(string property)
    : base(property)
  {

  }
  public dynamic[] In { get; set; }

  public bool Validate(dynamic entity)
  {
    return !In.Contains(PropertyValueIn(entity) as object);
  }
}
```

The In property in this code is the field in which the excluded values are stored; beyond that, this is pretty straightforward. If a descriptive error message needs to be included, Validation provides

a base property, ErrorMessage, in which a descriptive message can be stored for use if validation fails.

(For those who are curious about the "associations" from the database discussions last time, these are defined in Association.cs in the same folder, derive from Oak.Association, and—as one might expect—are a little bit trickier to explain. Fortunately, Oak has most of the traditional relational associations already defined, so there shouldn't be much need to customize here.)

Pieces of Oak

Sometimes parts of a library seem really cool, but obstacles stand in the way of adopting the whole thing, and you just wish you could rip out a small part of it and keep going. While it's usually better to use Oak as a whole, it does support the idea of ripping out bits of it (such as the dynamic database portions, or perhaps just the dynamic object portions, called Gemini, which I covered in the August 2013 issue, at msdn.microsoft.com/magazine/dn342877) and using them standalone, without the rest of the system. The Oak GitHub page on the subject (bit.ly/1cjGuou) has the NuGet packages for each of the standalone Oak parts, reproduced here (as of this writing) for your convenience:

- **install-package oak:** This is the full Oak suite, it includes MVC model binders, schema generation, the Oak DynamicModel and supporting classes, an altered version of Massive (DynamicRepository), and the Oak core dynamic construct Gemini.
- **install-package oak-json:** This is the part of Oak with regard to JSON serialization (can be used in REST APIs).
- **install-package cambium:** This is the part of Oak that excludes the MVC-specific components and excludes schema generation. Cambium includes the Oak DynamicDb, DynamicModel, an altered version of Massive (DynamicRepository) and the Oak core dynamic construct Gemini.
- **install-package seed:** This is the schema generation of Oak. This NuGet package also includes the altered version of Massive (used to insert sample data). It doesn't contain any of the MVC model binders, or the Oak DynamicModel or supporting classes.
- **install-package gemini:** This will install just the core dynamic construct upon which all the dynamic goodness in Oak is built.

Figure 1 **Conditional Validation**

```
public class Order : DynamicModel
{
  public Order()
  {

  }

  public IEnumerable<dynamic> Validates()
  {
    yield return new Presence("CardNumber") {
      If = d => d.PaidWithCard()
    };

    yield return new Presence("Address") {
      Unless = d => d.IsDigitalPurchase()
    };
  }

  public bool PaidWithCard()
  {
    // Could use This().PaymentType instead
    return _.PaymentType == "Card";

  }

  public bool IsDigitalPurchase()

  {
    // Could use This().ItemType instead
    return _.ItemType == "Digital";
  }
}
```

This is one of those times when a whole lot of interesting functionality and ideas come out of a pretty small (relatively speaking) package.

Before trying out any of them in pieces, I'd suggest trying the whole experience to get a feel for how each of them fit into the larger picture.

Benefits, Cost and Pain

As might be inferred from these four columns, there are definite benefits to being able to just "wing it" and work with a more dynamically typed system. Without question, costs and pain will raise their ugly heads in such a system (particularly for the unwary, and those unused to writing tests), but even those who are the most diehard statically typed bigots can learn some valuable ideas from a system like Oak. More important, Oak can be a hugely valuable tool for prototyping the early development of a system, when the object model is still highly mutable and undefined. Best of all, thanks to the underlying platform of Oak (that is, .NET), it becomes quite feasible to suggest building an MVC app in Oak in the early stages, then slowly flipping parts of it over to a more statically typed (and, thus, compiler-checked and compiler-enforced) approach as the details of the application get more tightly locked down.

Personally speaking, without a doubt, Oak is a cool little project. To my mind, this is one of those times when a whole lot of interesting functionality and ideas come out of a pretty small (relatively speaking) package. Oak definitely goes into my personal toolbox of tricks.

Happy coding!

Ted Neward *is the principal of 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's an F# MVP and speaks at 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, or read his blog at blogs.tedneward.com.*

Thanks to the following technical expert for reviewing this article:
Amir Rajan (Oak project creator)

RACHEL APPEL

A Look at the Hub Project and Control in Windows Store Apps

When it comes to development on Windows with Visual Studio, the built-in project templates are a good place to start. If you're new to Windows Store (or any Microsoft stack) development, the templates can serve as a learning tool. In this article, I'll look at the Hub control, but in context of the Hub project template. I'll examine all the important things to know about the Hub project and control for both HTML and XAML apps.

The Hub project in particular enables you to deliver a large volume of content to the user while using a modern UX. This is because you can break the app's content into parts called HubSections, so the app doesn't overwhelm the user visually with large amounts of data. While this is just my opinion, I find the Hub project to be the most aesthetically interesting of all the Windows Store app templates. The content layout is in distinct sections that are easy to digest. You can parade a favorite piece of content in the front-and-center "hero" section of the hub, while the remaining content items are easily accessible in groups.

> The Hub control is what you use to create a modern layout that's more than just boring groups of squares.

Of course, it's not mandatory that you use the templates—you can start from a blank project. However, for many developers, it's far easier to customize and expand upon the templates, as the code is set up for you.

The Hub Project Template

Visual Studio 2013 contains Hub project templates for both HTML and XAML. Upon creating a new HTML project using the template, you'll see some familiar project folders such as the css, images and js folders. In addition to the customary folders are the Hub-specific folders: pages\hub, pages\item and pages\section. As you might expect, each of these folders contains files that correspond to their purpose in the app. In the project root is the file for the package manifest as well as default.html, the app's starting point, which loads default.js and performs functions related to the app and lifecycle management. Default.html contains references to not just the \js\default.js file but also \js\data.js, which contains sample data, and \js\navigator.js, which performs navigation. For a refresher on navigation, see my August 2013 column, "Navigation Essentials in Windows Store Apps," at msdn.microsoft.com/magazine/dn342878. In short, the Hub project template, like other templates, is a quick way to publish visually interesting modern apps.

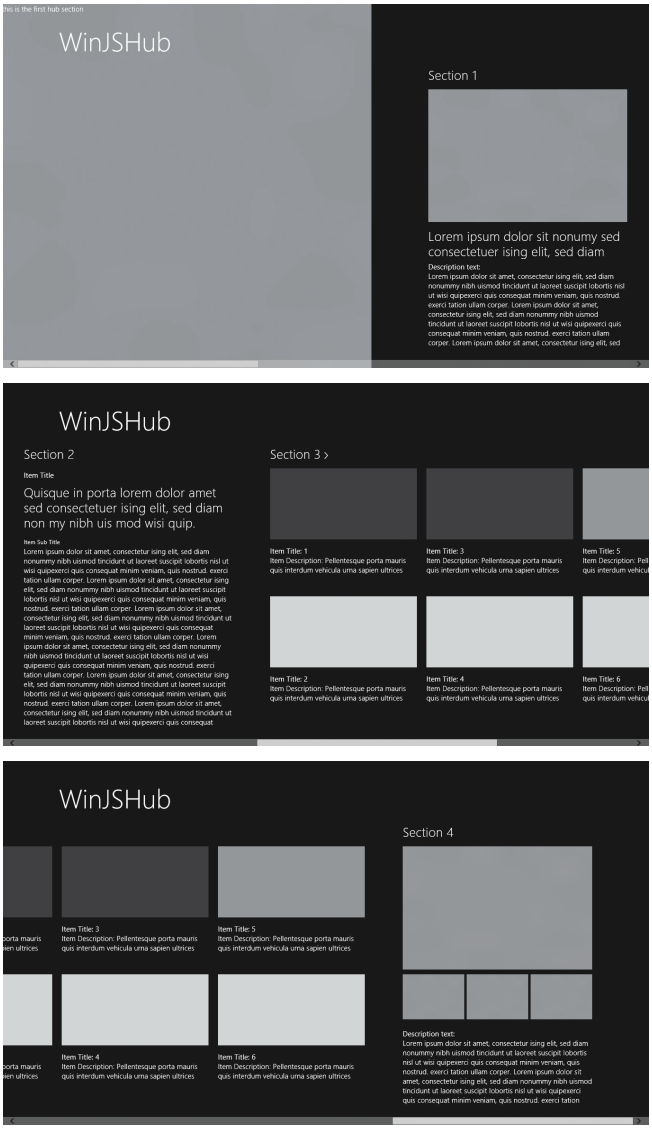


Figure 1 **The Hub Control at Run Time for Both HTML and XAML Apps**

ALMFORUMSEATTLE

expertise worth sharing

Washington State Convention Center • April 1-3, 2014

Register now at alm-forum.com follow us on

keynote speakers

Scott Ambler
ALM Thought Leader and Co-creator of DAD Framework

Disciplined Agile Delivery: The Foundation for Scaling Agile

Steve Denning
Award-winning Author

Transforming Management through Agile

Ken Schwaber
Industry Legend and Co-Creator of Scrum

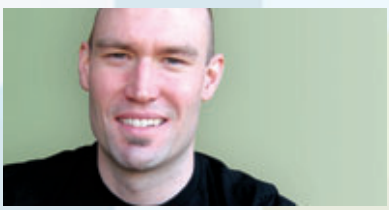
The State of Agile

Sam Guckenheimer
Product Owner, Microsoft Visual Studio

Transforming software development in a world of devices and services

plenary speakers

Mike Brittain
Director of Engineering, Etsy



Principles and Practices of Continuous Deployment

James Whittaker
Distinguished Technical Evangelist, Microsoft

Intent Commerce

Dave West
Chief Product Officer, Tasktop

Lean ALM

sponsors

diamond

platinum

Of course, the centerpiece of the Hub project is the Hub control. While default.html is the project starting point in an app built with the Windows Library for JavaScript (WinJS), once it loads, it immediately navigates to the hub.html file. Hub.html contains the Hub control and lives in the \pages\hub directory. The Hub control is what you use to create a modern layout that's more than just boring groups of squares. Instead, the Hub control, coupled with asynchronous data fetching, enables you to present large amounts of data—or data that has distinct groups—in an organized yet fashionable manner.

The Hub template implements the hub, or hierarchical, navigational pattern. This means that from the starting point (that is, hub page), the user can navigate to a page containing all the members of a particular section, or the user can navigate to an individual item from the hub page. The template also contains navigation to an item page from a section page. While the template contains navigation code only between section 3 and its groups and items (see **Figure 1**), you can use the ListView or Repeater controls to do the same type of navigation for other sections if it makes sense for your app. **Figure 1** illustrates what the default Hub app with sample data looks like at run time.

With the reimagining of Windows came the notion of putting content front and center, and, as you can see, this template does just that.

The XAML Hub template project works the same conceptually as does the HTML template, relying on the hub as the main entry point, being navigable to sections and details. Of course, the implementation is different, and you can see this by examining the folder structure, which reveals the following directories: Assets, Common, DataModel and Strings. These folders contain what you might expect: assets such as graphics, data in the DataModel folder and localized strings in the Strings folder. In the root of the project lies the following working files needed so the app can run:

- **App.xaml/.cs:** This is the XAML equivalent of default.html. It has a tiny bit of code that assists in navigation and general tasks.
- **HubPage.xaml/.cs:** This is the crowning jewel of the app, containing the Hub control.
- **ItemPage.xaml/.cs:** This contains the individual items you can navigate to from the hub or section pages.
- **SectionPage.xaml/.cs:** This shows all individual data members that belong to a particular group.
- **Package.appmanifest:** This contains the app settings.

> In HTML apps, the Hub control works just like any other WinJS control.

The XAML Hub project template's HubPage.xaml file reveals the Hub control firmly seats itself in a Grid control that serves as the root container for the page and Hub.

In the DataModel folder is a file named SampleData.json containing sample data. Also in the folder is a SampleDataSource.cs file that transforms the JSON data into usable classes for C# or Visual

Figure 2 **The HTML that Creates the Hub Control**

```
<div class="hub" data-win-control="WinJS.UI.Hub">
  <div class="hero" data-win-control="WinJS.UI.HubSection"></div>
  <div class="section1" data-win-control="WinJS.UI.HubSection"
    data-win-options="{ isHeaderStatic: true }"
    data-win-res="{ winControl: {'header': 'Section1'} }">
    <img src="/images/gray.png" width="420" height="280" />
    <div class="subtext win-type-x-large" data-win-res="
      { textContent: 'Section1Subtext' }"></div>
    <div class="win-type-medium"
      data-win-res="{ textContent: 'DescriptionText' }"></div>
    <div class="win-type-small">
      <span data-win-res="{ textContent: 'Section1Description' }"></span>
      <span data-win-res="{ textContent: 'Section1Description' }"></span>
      <span data-win-res="{ textContent: 'Section1Description' }"></span>
    </div>
  </div>
  <div class="section2" data-win-control="WinJS.UI.HubSection"
    data-win-options="{ isHeaderStatic: true }"
    data-win-res="{ winControl: {'header': 'Section2'} }">
  <div class="item-title win-type-medium"
    data-win-res="{ textContent: 'Section2ItemTitle' }"></div>
    <div class="article-header win-type-x-large"
      data-win-res="{ textContent: 'Section2Subtext' }"></div>
    <div class="win-type-xx-small"
      data-win-res="{ textContent: 'Section2ItemSubTitle' }"></div>
    <div class="win-type-small">
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
      <span data-win-res="{ textContent: 'Section2Description' }"></span>
    </div>
  </div>

  <div class="section3" data-win-control="WinJS.UI.HubSection"
    data-win-res="{ winControl: {'header': 'Section3'} }
    "data-win-options="{ onheaderinvoked:
    select('.pagecontrol').winControl.section3HeaderNavigate }">
    <div class="itemTemplate" data-win-control="WinJS.Binding.Template">
      <img src="#" data-win-bind="src: backgroundImage; alt: title" />
      <div class="win-type-medium" data-win-bind="textContent: title"></div>
      <div class="win-type-small"
        data-win-bind="textContent: description"></div>
      </div>
      <div class="itemslist win-selectionstylefilled" data-win-control=
        "WinJS.UI.ListView" data-win-options=
        "{layout: {type: WinJS.UI.GridLayout},
        selectionMode: 'none',
        itemTemplate: select('.section3 .itemTemplate'), itemDataSource:
        select('.pagecontrol').winControl.section3DataSource, oniteminvoked:
        select('.pagecontrol').winControl.section3ItemNavigate
        }">
      </div>
    </div>
  <div class="section4" data-win-control="WinJS.UI.HubSection"
    data-win-options="{ isHeaderStatic: true }"
    data-win-res="{ winControl: {'header': 'Section4'} }">
    <div class="top-image-row">
      <img src="/images/gray.png" />
    </div>
    <div class="sub-image-row">
      <img src="/images/gray.png" />
      <img src="/images/gray.png" />
      <img src="/images/gray.png" />
    </div>
    <div class="win-type-medium"
      data-win-res="{ textContent: 'DescriptionText' }"></div>
    <div class="win-type-small">
      <span data-win-res="{ textContent: 'Section4Description' }"></span>
      <span data-win-res="{ textContent: 'Section4Description' }"></span>
    </div>
  </div>
</div>
```

Basic .NET consumption and XAML data binding. You can replace this with your own data, much like the data.js file in WinJS apps.

The Common folder contains several files that perform a variety of tasks such as navigation and other generally app-related tasks for working with data in view models. In addition, the Common folder contains the SuspensionManager.cs file, which performs process lifecycle tasks. Finally, the Strings folder contains localized strings for publishing in different locales.

The Hub Control

Both HTML and XAML project templates use the Hub control. In HTML apps, the Hub control works just like any other WinJS control. Use the data-win-control attribute of an HTML element, usually a , to define it as a Hub control, as this code shows:

```
<div class="hub" data-win-control="WinJS.UI.Hub"></div>
```

This means the WinJS.UI.Hub object is the brains behind the Hub control. The Hub control acts as a container for the HubSection

Figure 3 **The XAML for a Hub Control**

```
<Hub SectionHeaderClick="Hub_SectionHeaderClick">
  <Hub.Header>
    <!-- Back button and page title -->
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="80"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Button  x:Name="backButton" Style=
        "{StaticResource NavigationBackButtonNormalStyle}"
        Margin="-1,-1,39,0"
        VerticalAlignment="Top"
        Command="{Binding NavigationHelper.GoBackCommand,
        ElementName=pageRoot}"
        AutomationProperties.Name="Back"
        AutomationProperties.AutomationId="BackButton"
        AutomationProperties.ItemType="Navigation Button"/>
      <TextBlock x:Name="pageTitle" Text="{StaticResource AppName}"
        Style="{StaticResource HeaderTextBlockStyle}" Grid.Column="1"
        VerticalAlignment="Top" IsHitTestVisible="false"
        TextWrapping="NoWrap" />
    </Grid>
  </Hub.Header>
  <HubSection Width="780" Margin="0,0,80,0">
    <HubSection.Background>
      <ImageBrush ImageSource="Assets/MediumGray.png"
        Stretch="UniformToFill" />
    </HubSection.Background>
  </HubSection>
  <HubSection Width="500" x:Uid="Section1Header" Header="Section 1">
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
          <RowDefinition Height="*" />
          </Grid.RowDefinitions>
          <Image Source="Assets/MediumGray.png" Stretch="Fill"
            Width="420" Height="280"/>
          <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
            Grid.Row="1" Margin="0,10,0,0" TextWrapping="Wrap"
            x:Uid="Section1Subtitle" Text="Lorem ipsum dolor sit nonumy
            sed consectetuer ising elit, sed diam"/>
          <TextBlock Style="{StaticResource TitleTextBlockStyle}"
            Grid.Row="2" Margin="0,10,0,0" x:Uid="DescriptionHeader"
            Text="Description text:"/>
          <TextBlock Style="{StaticResource BodyTextBlockStyle}"
            Grid.Row="3"
            x:Uid="Section1DescriptionText"
            Text="Lorem ipsum dolor sit amet... "/>
      </Grid>
    </DataTemplate>
  </HubSection>
  <HubSection Width="520" x:Uid="Section2Header" Header="Section 2">
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
          <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <TextBlock Style="{StaticResource TitleTextBlockStyle}"
          Margin="0,0,0,10" x:Uid="ItemTitle" Text="Item Title" />
        <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
          Grid.Row="1" x:Uid="Section2UnderTitle" Text="Quisque in porta
            lorem dolor amet sed consectetuer ising elit, sed diam non
            my nibh uis mod wisi quip."/>
          <TextBlock Style="{StaticResource SubtitleTextBlockStyle}"
            Grid.Row="2" Margin="0,20,0,0" x:Uid="ItemSubTitle"
            Text="Item Sub Title"/>
          <TextBlock Style="{StaticResource BodyTextBlockStyle}" Grid.Row="3"
            x:Uid="LongText" Text="Lorem ipsum dolor sit amet..."/>
        </Grid>
      </DataTemplate>
    </HubSection>
    <HubSection IsHeaderInteractive="True"
      DataContext="{Binding Section3Items}" d:DataContext="{Binding Groups[3],
      Source={d:DesignData Source=/DataModel/SampleData.json,
      Type=data:SampleDataSource}}" x:Uid="Section3Header" Header="Section 3"
      Padding="40,40,40,32">
      <DataTemplate>
        <GridView
          x:Name="itemGridView"
          ItemsSource="{Binding Items}"
          Margin="-9,-14,0,0"
          AutomationProperties.AutomationId="ItemGridView"
          AutomationProperties.Name="Items In Group"
          ItemTemplate="{StaticResource Standard310x260ItemTemplate}"
          SelectionMode="None"
          IsSwipeEnabled="false"
          IsItemClickEnabled="True"
          ItemClick="ItemView_ItemClick">
        </GridView>
      </DataTemplate>
    </HubSection>
    <HubSection x:Uid="Section4Header" Header="Section 4">
      <DataTemplate>
        <!-- width of 400 -->
        <StackPanel Orientation="Vertical">
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="130"/>
              <ColumnDefinition Width="5"/>
              <ColumnDefinition Width="130"/>
              <ColumnDefinition Width="5"/>
              <ColumnDefinition Width="130"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
              <RowDefinition Height="270"/>
              <RowDefinition Height="95"/>
              <RowDefinition Height="Auto" />
              <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Image Source="Assets/MediumGray.png"
              Grid.ColumnSpan="5" Margin="0,0,0,10" Stretch="Fill" />
            <Image Source="Assets/MediumGray.png" Grid.Row="1" Stretch="Fill"/>
             <Image Source="Assets/MediumGray.png" Grid.Row="1"
               Grid.Column="2" Stretch="Fill"/>
            <Image Source="Assets/MediumGray.png" Grid.Row="1"
              Grid.Column="4" Stretch="Fill"/>
            <TextBlock Style="{StaticResource TitleTextBlockStyle}"
              Grid.Row="2" Grid.ColumnSpan="5"  Margin="0,15,0,0"
              x:Uid="DescriptionHeader" Text="Description text:"/>
            <TextBlock Style="{StaticResource BodyTextBlockStyle}"
              Grid.Row="3" Grid.ColumnSpan="5" x:Uid="LongText"
              Text="Lorem ipsum dolor sit amet...."/>
          </Grid>
        </StackPanel>
      </DataTemplate>
    </HubSection>
</Hub>
```

Figure 4 **The CSS That Shapes and Styles the HTML Hub Control**

```
.hubpage header[role=banner] {
  position: relative;
  z-index: 2;
}
.hubpage section[role=main] {
  -ms-grid-row: 1;
  -ms-grid-row-span: 2;
  z-index: 1;
}
.hubpage .hub .win-hub-surface {
  height: 100%;
}
.hubpage .hub .hero {
  -ms-high-contrast-adjust: none;
  background-image: url(/images/gray.png);
  background-size: cover;
  margin-left: -80px;
  margin-right: 80px;
  padding: 0;
  width: 780px;
}
  .hubpage .hub .hero:-ms-lang(
    ar, dv, fa, he, ku-Arab, pa-Arab, prs, ps, sd-Arab,
    syr, ug, ur, qps-plocm) {
    margin-left: 80px;
    margin-right: -80px;
  }
  .hubpage .hub .hero .win-hub-section-header {
    display: none;
  }
  .hubpage .hub .section1 {
    width: 420px;
  }
  .hubpage .hub .section1 .win-hub-section-content {
    overflow-y: hidden;
  }
  .hubpage .hub .section1 .subtext {
    margin-bottom: 7px;
    margin-top: 9px;
  }
.hubpage .hub .section2 {
  width: 440px;
}
  .hubpage .hub .section2 .win-hub-section-content {
    overflow-y: hidden;
  }
  .hubpage .hub .section2 .item-title {
      margin-top: 4px;
      margin-bottom: 10px;
  }
  .hubpage .hub .section2 .article-header {
    margin-bottom: 15px;
  }
.hubpage .hub .section3 {
}
  .hubpage .hub .section3 .itemslist {
    height: 100%;
    margin-left: -10px;
    margin-right: -10px;
    margin-top: -5px;
  }
  .hubpage .hub .section3 .win-container {
    margin-bottom: 36px;
    margin-left: 10px;
    margin-right: 10px;
  }
  .hubpage .hub .section3 .win-item {
      height: 229px;
      width: 310px;
  }
      .hubpage .hub .section3 .win-item img {
        height: 150px;
        margin-bottom: 10px;
        width: 310px;
      }
.hubpage .hub .section4 {
  width: 400px;
}
  .hubpage .hub .section4 .win-hub-section-content {
        overflow-y: hidden;
  }
  .hubpage .hub .section4 .top-image-row {
    height: 260px;
    margin-bottom: 10px;
    width: 400px;
  }
    .hubpage .hub .section4 .top-image-row img {
      height: 100%;
       width: 100%;
    }
.hubpage .hub .section4 .sub-image-row {
  margin-bottom: 20px;
  display: -ms-flexbox;
  -ms-flex-flow: row nowrap;
  -ms-flex-pack: justify;
}
  .hubpage .hub .section4 .sub-image-row img {
    height: 95px;
     width: 130px;
  }
```

controls, which define sections or groups of data. HubSections can contain any valid HTML tags, such as or <img>, or a WinJS control, such as the ListView control. By default, the hub.html file's Hub control encloses five sections, one named hero and four more designated by their class attributes (such as section1, section2 and so on). In the HubSections, the and <img> tags are the most common child elements, but any valid HTML or WinJS controls will work to display data in a different layout. Changing the layout is a great way to personalize your app, but don't forget to adhere to the Windows UX guidelines at bit.ly/1gBDHaW. **Figure 2** shows a complete sample of the necessary HTML (you'll see its CSS later) to create a Hub control with five sections. Inspecting the code in **Figure 2** shows section 3 is the navigable section, while the rest are not navigable.

In XAML, the Hub control uses a <Hub> element that contains <Hub.Header> and <HubSection> elements. In turn, the child headings and sections contain Grid and other XAML controls, such as the StackPanel, as well as text blocks. **Figure 3** shows the XAML required to create the Hub control used in the Visual Studio templates.

As you can see, XAML syntax is a bit more verbose than HTML. That's because you code layout and styles right in the XAML page (though XAML styles can be placed in a resource dictionary), while in HTML the layout and style rules are CSS (more on styling later).

Data Binding and the Hub Control

Arrays or JSON (which usually serializes to an array anyway) are the customary ways to work with data in WinJS, as well as in many other Web or client languages. This is no different with the Hub project. You can replace the data in \js\data.js with custom data broken into however many groups you plan to use. You'll find two arrays as sample data in the data.js file, one for grouping and one for individual items that tie into a specific group. If you're familiar with some of the other WinJS project templates, then you'll notice this is the same sample data.

In the \pages\hub\hub.js file are the calls to the members of the Data namespace that obtain group and item data:

```
var section3Group = Data.resolveGroupReference("group4");
var section3Items = Data.getItemsFromGroup(section3Group);
```

The section3Group and section3Items are global objects. **Figure 2** shows the data-binding syntax for the ListView control. In hub.js, after the ready function, the code sets section3DataSource, a property of the Hub control:

```
section3DataSource: section3Items.dataSource,
```

The Hub control uses the preceding code to data bind to the ListView (**Figure 2** shows the data-bound ListView code).

> In Windows Store apps built with JavaScript, you can style the Hub control with CSS.

In XAML apps using C#, you have the same basic occurrences, as code from the HubPage.xaml.cs file indicates the following declaration for a view model of type ObservableDictionary, along with its corresponding property declaration (this is where you can return your own data):

```
private ObservableDictionary defaultViewModel = new ObservableDictionary();
public ObservableDictionary DefaultViewModel
{
  get { return this.defaultViewModel; }
}
```

Later in the file, code sets a page-level view model by calling GetGroupAsync, which, as its name implies, runs asynchronously:

```
var sampleDataGroup = await SampleDataSource.GetGroupAsync("Group-4");
```

Although the call obtains Group-4 data, you assign it to a view model named Section3Items to assign it to those items. Consider the hero section as Section 0, meaning the Section 3 items will align with the Group-4 data:

```
this.DefaultViewModel["Section3Items"] = sampleDataGroup;
```

This is all you need in the codebehind. In XAML, notice the DataContext attribute binds to Section3Items.The other attributes aren't necessary for data binding, but act as an aid for the design tools in Visual Studio or Blend, as designated by the "d" namespace:

```
<HubSection IsHeaderInteractive="True" DataContext="{Binding Section3Items}"
  d:DataContext="{Binding Groups[3], Source={d:DesignData
  Source=/DataModel/SampleData.json, Type=data:SampleDataSource}}"
  x:Uid="Section3Header" Header="Section 3" Padding="40,40,40,32">
```

While working with local sample data, you have many options for data access, including File IO, SQLite, Web Storage, IndexedDB, REST services and Windows Azure, to name a few. If you want to review what data options are available, see my March 2013 article, "Data Access and Storage Options in Windows Store Apps," at msdn.microsoft.com/magazine/jj991982.

Styling the Hub Control

In Windows Store apps built with JavaScript, you can style the Hub control with CSS. The \hub\hub.css file contains all the default CSS related to the Hub control. Feel free to add your own styles to change the size of the elements or their layout. **Figure 4** shows the complete CSS in hub.css. Notice there's a .hubpage class selector that uses HTML5 semantic role attributes such as header[role=banner] and section[role=main] to designate the general styles for the hub. After that, the CSS in **Figure 4** shows the ".hubpage .hub .hero" descendant selector, which creates the featured (hero) section of the Hub control. The hero fills roughly half of the left side of the viewable part of screen with a light gray background and, of course, it's a great way to put a special piece of content where no user can miss it! You can fill it with lots of data, and graphic data or multimedia works quite nicely to show off here.

As you can see, the CSS in **Figure 4** shapes and styles the Hub control, and most of it deals with the layout and sizing of the HubSections. Elements and WinJS controls inside the HubSections apply the styles from ui-light.css or ui-dark.css, until you overwrite them with your own styles.

HTML apps rely on CSS for styling. XAML apps rely on XAML for styling. This means that XAML has several attributes you apply to tags to enforce styling definitions called resources. For example, the code that styles a TextBlock is the Style attribute and it references a built-in (static resource dictionary) style named SubheaderTextBlockStyle:

```
<TextBlock Style="{StaticResource SubheaderTextBlockStyle} />
```

The layout of a page is also XAML, as all the Hubs, Grids and other elements contain inline coordinates for their on-screen position as well as size. You can see throughout **Figure 3** there are margins, positioning, and row and column settings that position elements, all inline in the XAML. HTML is originally a Web technology, and conserving bandwidth by using CSS instead of HTML is a real benefit. Here in the land of XAML, it's all client-side, so UI caching isn't so much of an issue and styles can go inline. A nice upside of XAML is that you need to do very little to ensure a responsive design. Just be sure to set two <RowDefinition> elements to a height of "Auto" and "*":

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

The rows will automatically respond to app view state changes, making the layout fluid while saving extra code. **Figure 3** shows a few references to auto-height row definitions.

Samples Available

Once you've modified the Hub control, performed data retrieval and binding, and set styles, you're good to go. Don't forget to add modern touches such as tiles, search and other Windows integration to your app. The Hub project template is an easy way to build and publish apps quickly, whether in HTML or XAML. Using the hub navigational pattern with the Hub control enables you to build an effective and rich UX that adheres to modern UI principles. You can download Hub control samples covering many aspects of Windows app development at the following locations:

- HTML sample: bit.ly/1mOsWTE
- XAML sample: bit.ly/1eGsVAH ■

Rachel Appel *is a consultant, author, mentor and former Microsoft employee with more than 20 years of experience in the IT industry. She speaks at top industry conferences such as Visual Studio Live!, DevConnections, MIX and more. Her expertise lies within developing solutions that align business and technology focusing on the Microsoft dev stack and open Web. For more about Appel, visit her Web site at rachelappel.com.*

Thanks to the following technical expert for reviewing this article: *Frank La Vigne (Microsoft)*

LAS VEGAS

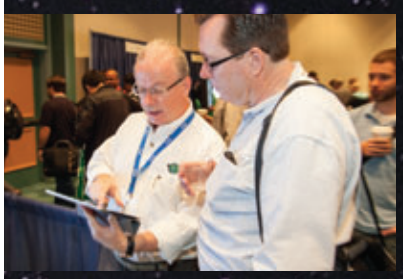
LIVE! 360 DEV

PLANET HOLLYWOOD RESORT AND CASINO

COMPREHENSIVE TRAINING FOR THE DEVELOPER WORLD

LIVE! Visual Studio | LIVE! SQL Server | LIVE! SharePoint | LIVE! Web Dev | LIVE! Modern Apps

COMPREHENSIVE TRAINING



The Developer World is always changing; new technologies emerge, current ones evolve and demands on your time grow. Live! 360 DEV offers comprehensive training through 5 co-located events on the most relevant and leading edge technologies in your world today. You'll learn from pre-eminent experts in the industry, network with like-minded peers, and return home with the knowledge and solutions you need to tackle your biggest development challenges.

CHARLES PETZOLD

Triangles and Tessellation

The triangle is the most basic two-dimensional figure. It's nothing more than three points connected by three lines, and if you try to make it any simpler, it collapses into a single dimension. On the other hand, any other type of polygon can be decomposed into a collection of triangles.

Even in three dimensions, a triangle is always flat. Indeed, one way to define a plane in 3D space is with three non-collinear points, and that's a triangle. A square in 3D space isn't guaranteed to be flat because the fourth point might not be in the same plane as the other three. But that square can be divided into two triangles, each of which is flat, although not necessarily on the same plane.

> In Direct2D, tessellation is the process of decomposing a two-dimensional area into triangles.

In 3D graphics programming, triangles form the surfaces of solid figures, starting with the simplest of all three-dimensional figures, the triangular pyramid, or tetrahedron. Assembling a seemingly solid figure from triangle "building blocks" is the most fundamental process in 3D computer graphics. Of course, the surfaces of real-world objects are often curved, but if you make the triangles small enough, they can approximate curved surfaces to a degree sufficient to fool the human eye.

The illusion of curvature is enhanced by exploiting another characteristic of triangles: If the three vertices of a triangle are associated with three different values—for example, three different colors or three different geometric vectors—these values can be interpolated over the surface of the triangle and used to color that surface. This is how triangles are shaded to mimic the reflection of light seen in real-world objects.

Triangles in Direct2D

Triangles are ubiquitous in 3D computer graphics. Much of the work performed by a modern graphics processing unit (GPU) involves rendering triangles, so of course Direct3D programming involves working with triangles to define solid figures.

Code download available at msdn.microsoft.com/magazine/msdnmag0314.

In contrast, triangles aren't found at all in most 2D graphics programming interfaces, where the most common two-dimensional primitives are lines, curves, rectangles and ellipses. So it's somewhat surprising to find triangles pop up in a rather obscure corner of Direct2D. Or maybe it's really not that surprising: Because Direct2D is built on top of Direct3D, it seems reasonable for Direct2D to take advantage of the triangle support in Direct3D and the GPU.

The triangle structure defined in Direct2D is simple:

```
struct D2D1_TRIANGLE
{
  D2D1_POINT_2F point1;
  D2D1_POINT_2F point2;
  D2D1_POINT_2F point3;
};
```

As far as I can determine, this structure is used in Direct2D only in connection with a "mesh," which is a collection of triangles stored in an object of type ID2D1Mesh. The ID2D1RenderTarget (from which ID2D1DeviceContext derives) supports a method named CreateMesh that creates such an object:

```
ID2D1Mesh * mesh;
deviceContext->CreateMesh(&mesh);
```

(To keep things simple, I'm not showing the use of ComPtr or checking HRESULT values in these brief code examples.) The ID2D1Mesh interface defines a single method named Open. This method returns an object of type ID2D1TessellationSink:

```
ID2D1TessellationSink * tessellationSink;
mesh->Open(&tessellationSink);
```

In general, "tessellation" refers to the process of covering a surface with a mosaic pattern, but the term is used somewhat differently in

Figure 1 **The Relevant Code of InterrogableTessellationSink**

```
// ID2D1TessellationSink methods
void InterrogableTessellationSink::AddTriangles(_In_ const D2D1_TRIANGLE *triangles,
                                                UINT trianglesCount)
{
  for (UINT i = 0; i < trianglesCount; i++)
  {
    m_triangles.push_back(triangles[i]);
  }
}

HRESULT InterrogableTessellationSink::Close()
{
  // Assume the class accessing the tessellation sink knows what it's doing
  return S_OK;
}

// Method for this implementation
std::vector<D2D1_TRIANGLE> InterrogableTessellationSink::GetTriangles()
{
  return m_triangles;
}
```

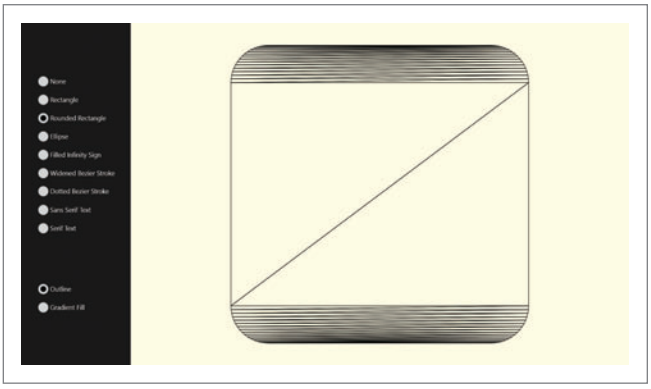


Figure 2 **A Rounded Rectangle Decomposed into Triangles**

Direct2D and Direct3D programming. In Direct2D, tessellation is the process of decomposing a two-dimensional area into triangles.

The ID2D1TessellationSink interface has just two methods: AddTriangles (which adds a collection of D2D1_TRIANGLE objects to the collection) and Close, which makes the mesh object immutable.

Although your program can call AddTriangles itself, often it will pass the ID2D1TessellationSink object to the Tessellate method defined by the ID2D1Geometry interface:

```
geometry->Tessellate(IdentityMatrix(), tessellationSink);
tessellationSink->Close();
```

The Tessellate method generates triangles that cover the areas enclosed by the geometry. After you call the Close method, the sink can be discarded and you're left with an ID2D1Mesh object. The process of generating the contents of an ID2D1Mesh object using an ID2D1TessellationSink is similar to defining an ID2D1PathGeometry using an ID2D1GeometrySink.

You can then render this ID2D1Mesh object using the FillMesh method of ID2D1RenderTarget. A brush governs how the mesh is colored:

```
deviceContext->FillMesh(mesh, brush);
```

Keep in mind that these mesh triangles define an area and not an outline of an area. There is no DrawMesh method.

FillMesh has a limitation: Anti-aliasing can't be enabled when FillMesh is called. Precede FillMesh with a call to SetAntialiasMode:

```
deviceContext->SetAntialiasMode(D2D1_ANTIALIAS_MODE_ALIASED);
```

You might wonder: What's the point? Why not just call FillGeometry on the original geometry object? The visuals should be the same (aside from the anti-aliasing). But there's actually a profound difference between ID2D1Geometry and ID2D1Mesh objects that's revealed by how you create these two objects.

Figure 3 **Text Decomposed into Triangles**

Geometries are mostly just collections of coordinate points, so geometries are device-independent objects. You can create various types of geometries by calling methods defined by ID2D1Factory.

A mesh is a collection of triangles, which are just triplets of coordinate points, so a mesh should be a device-independent object as well. But you create an ID2D1Mesh object by calling a method defined by ID2D1RenderTarget. This means the mesh is a device-*dependent* object, like a brush.

This tells you the triangles that comprise the mesh are stored in a device-dependent manner, most likely in a form suitable for processing by the GPU, or actually on the GPU. And this means that FillMesh should execute much faster than FillGeometry for the equivalent figure.

Shall we test that hypothesis?

Among the downloadable code for this article is a program named MeshTest that creates a path geometry for a 201-point star, and slowly rotates it while calculating and displaying the frame rate. When the program is compiled in Debug mode for the x86 and runs on my Surface Pro, I get a frame rate of less than 30 frames per second (FPS) when rendering the path geometry (even if the geometry is outlined to eliminate overlapping areas and flattened to eliminate curves), but the frame rate leaps up to 60FPS when rendering the mesh.

Geometries are mostly just collections of coordinate points, so geometries are device-independent objects.

Conclusion: For complex geometries, it makes sense to convert them to meshes for rendering. If the need to disable anti-aliasing to render this mesh is a deal-breaker, you might want to check out ID2D1GeometryRealization, introduced in Windows 8.1. This combines the performance of ID2D1Mesh but allows anti-aliasing. Keep in mind meshes and geometry realizations must be recreated if the display device is recreated, just as with other device-dependent resources such as brushes.

Examining the Triangles

I was curious about the triangles generated by the tessellation process. Could they actually be visualized? The ID2D1Mesh object doesn't allow you to access the triangles that comprise the mesh, but it's possible to write your own class that implements the ID2D1TessellationSink interface, and pass an instance of that class to the Tessellate method.

I called my ID2D1TessellationSink implementation InterrogableTessellationSink, and it turned out to be embarrassingly simple. It contains a private data member for storing triangle objects:

```
std::vector<D2D1_TRIANGLE> m_triangles;
```

Most of the code is dedicated to implementing the IUnknown interface. **Figure 1** shows the code required to implement the two ID2D1TessellationSink methods and obtain the resultant triangles.

I incorporated this class in a project named TessellationVisualization. The program creates geometries of various sorts—everything from a simple rectangle geometry to geometries generated from text glyphs—and uses InterrogableTessellationSink to obtain the collection of triangles created by the Tessellate method. Each triangle is then converted into an ID2D1PathGeometry object consisting of three straight lines. These path geometries are then rendered using DrawGeometry.

As you might expect, an ID2D1RectangleGeometry is tessellated into just two triangles, but the other geometries are more interesting. **Figure 2** shows the triangles that comprise an ID2D1RoundedRectangleGeometry.

This isn't the way a human being would tessellate the rounded rectangle. A human being would probably divide the rounded rectangle into five rectangles and four quarter-circles, and tessellate each of those figures separately. In particular, a human would slice the four quarter-circles into pie wedges.

In other words, a human being would define several more points in the interior of the geometry to aid in the tessellation. But the tessellation algorithm defined by the geometry object doesn't use any points beyond those created by the flattening of the geometry.

Figure 3 shows two characters rendered with the Pescadero font decomposed into triangles.

I was also curious about the order in which these triangles were generated, and by clicking the Gradient Fill option at the bottom left of the window, you can find out. When this option is checked, the program calls FillGeometry for each of the triangle geometries. A solid color brush is passed to FillGeometry but the color depends on the triangle's index in the collection.

What you'll find is that the FillGeometry option renders something akin to a top-down gradient brush, which means that triangles are stored in the collection in a visual top-down order. It appears the tessellation algorithm attempts to maximize the width of horizontal scan lines in the triangles, which probably maximizes the rendering performance.

Although I clearly recognize the wisdom of this approach, I must confess I was a little disappointed. I was hoping that a widened Bézier curve (for example) might be tessellated beginning at one end of the line and continuing to the other, so the triangles could be rendered with a gradient from one end to the other, which is not a type of gradient commonly seen in a DirectX program! But this was not to be.

Interestingly, I needed to turn off anti-aliasing before the FillGeometry calls in TessellationVisualization or faint lines appeared between the rendered triangles. These faint lines result from the anti-aliasing algorithm, which involves partially transparent pixels that don't become opaque when overlapped. This leads me to suspect that using anti-aliasing with FillMesh isn't a hardware or software limitation, but a restriction mandated to avoid visual anomalies.

Figure 4 **Tessellation and Rendering Code in SparklingTextRenderer**

```
void SparklingTextRenderer::Tessellate()
{
  ...

  // Tessellate geometry into triangles
  ComPtr<InterrogableTessellationSink> tessellationSink =
    new InterrogableTessellationSink();
  pathGeometry->Tessellate(IdentityMatrix(), tessellationSink.Get());
  std::vector<D2D1_TRIANGLE> triangles = tessellationSink->GetTriangles();

  if (m_useMeshesNotGeometries)
  {
    // Generate a separate mesh from each triangle
    ID2D1DeviceContext* context = m_deviceResources->GetD2DDeviceContext();

    for (D2D1_TRIANGLE triangle : triangles)
    {
      ComPtr<ID2D1Mesh> triangleMesh;
      context->CreateMesh(&triangleMesh);
      ComPtr<ID2D1TessellationSink> sink;
      triangleMesh->Open(&sink);
      sink->AddTriangles(&triangle, 1);
      sink->Close();

      m_triangleMeshes.push_back(triangleMesh);
    }
  }
  else
  {
    // Generate a path geometry from each triangle
    for (D2D1_TRIANGLE triangle : triangles)
    {
      ComPtr<ID2D1PathGeometry> triangleGeometry;
      d2dFactory->CreatePathGeometry(&triangleGeometry);
      ComPtr<ID2D1GeometrySink> geometrySink;
      triangleGeometry->Open(&geometrySink);
      geometrySink->BeginFigure(triangle.point1, D2D1_FIGURE_BEGIN_FILLED);
      geometrySink->AddLine(triangle.point2);
      geometrySink->AddLine(triangle.point3);
      geometrySink->EndFigure(D2D1_FIGURE_END_CLOSED);
      geometrySink->Close();

      m_triangleGeometries.push_back(triangleGeometry);
    }
  }
}

void SparklingTextRenderer::Render()
{
  ...

  Matrix3x2F centerMatrix = D2D1::Matrix3x2F::Translation(
    (logicalSize.Width - (m_geometryBounds.right + m_geometryBounds.left)) / 2,
    (logicalSize.Height - (m_geometryBounds.bottom + m_geometryBounds.top)) / 2);

  context->SetTransform(centerMatrix *
    m_deviceResources->GetOrientationTransform2D());
  context->SetAntialiasMode(D2D1_ANTIALIAS_MODE_ALIASED);

  if (m_useMeshesNotGeometries)
  {
    for (ComPtr<ID2D1Mesh>& triangleMesh : m_triangleMeshes)
    {
      float gray = (rand() % 1000) * 0.001f;
      m_solidBrush->SetColor(ColorF(gray, gray, gray));
      context->FillMesh(triangleMesh.Get(), m_solidBrush.Get());
    }
  }
  else
  {
    for (ComPtr<ID2D1PathGeometry>& triangleGeometry : m_triangleGeometries)
    {
      float gray = (rand() % 1000) * 0.001f;
      m_solidBrush->SetColor(ColorF(gray, gray, gray));
      context->FillGeometry(triangleGeometry.Get(), m_solidBrush.Get());
    }
  }

  ...
}
```

Figure 5 **The SparklingText Display**

Triangles in 2D and 3D

After working just a little while with ID2D1Mesh objects, I began visualizing all two-dimensional areas as mosaics of triangles. This mindset is normal when doing 3D programming, but I had never extended such a triangle-centric vision to the 2D world.

The documentation of the Tessellate method indicates the generated triangles are "clockwise-wound," which means that the point1, point2 and point3 members of the D2D1_TRIANGLE structure are ordered in a clockwise direction. This isn't very useful information when using these triangles in 2D graphics programming, but it becomes quite important in the 3D world, where the ordering of the points in a triangle usually indicates the front or back of the figure.

Of course, I'm very interested in using these two-dimensional tessellated triangles to break through the third dimension, where triangles are most comfortably at home. But I don't want to be in such a rush that I neglect to explore some interesting effects with tessellated triangles in two dimensions.

Coloring Triangles Uniquely

For me, the biggest thrill in graphics programming is creating images on the computer screen of a sort I've never seen before, and I don't think I've ever seen text tessellated into triangles whose colors change in a random manner. This happens in a program I call SparklingText.

Keep in mind that both FillGeometry and FillMesh involve only a single brush, so if you need to render hundreds of triangles with different colors, you'll need hundreds of FillGeometry or FillMesh calls, each rendering a single triangle. Which is more efficient? A FillGeometry call to render an ID2D1PathGeometry that consists of three straight lines? Or a FillMesh call with an ID2D1Mesh containing a single triangle?

I assumed that FillMesh would be more efficient than FillGeometry only if the mesh contained multiple triangles, and it would be slower for one triangle, so I originally wrote the program to generate path geometries from the tessellated triangles. Only later did I add a CheckBox labeled "Use a Mesh for each triangle instead of a PathGeometry" and incorporated that logic as well.

The strategy in the SparklingTextRenderer class of SparklingText is to use the GetGlyphRunOutline method of ID2D1FontFace to obtain a path geometry for the character outlines. The program then calls the Tessellate method on this geometry with the InterrogableGeometrySink to get a collection of D2D1_TRIANGLE objects. These are then converted into path geometries or meshes (depending on the CheckBox value) and stored in one of two vector collections named m_triangleGeometries and m_triangleMeshes.

> After working just a little while with ID2D1Mesh objects, I began visualizing all two-dimensional areas as mosaics of triangles.

Figure 4 shows a pertinent chunk of the Tessellate method that fills these collections, and the Render method that renders the resultant triangles. As usual, HRESULT-checking has been removed to simplify the code listings.

Based on the video frame rate (which the program displays), my Surface Pro renders the meshes faster than the path geometries, despite the fact that each mesh contains just a single triangle.

The animation of the colors is unnervingly reminiscent of a scintillating migraine aura, so you might want to exercise some caution when viewing it. **Figure 5** shows a still image from the program, which should be much safer.

Moving the Tessellated Triangles

The remaining two programs use a strategy similar to SparklingText to generate a collection of triangles to form glyph outlines, but then move the little triangles around the screen.

Figure 6 **A Still from the OutThereAndBackAgain Program**

Figure 7 **The TextMorphing Display**

For OutThereAndBackAgain, I envisioned text that would fly apart into its composite triangles, which would then come back to form the text again. **Figure 6** shows this process at 3 percent into the flying-apart animation.

The CreateWindowSizeDependentResources method in the OutThereAndBackAgainRenderer class assembles information about each triangle in a structure I call TriangleInfo. This structure contains a single-triangle ID2D1Mesh object, as well as information necessary to take that triangle on a journey outward and back again. This journey takes advantage of a feature of geometries you can use independently of rendering. The ComputeLength method in ID2D1Geometry returns the total length of a geometry, while ComputePointAtLength returns a point on the curve and a tangent to the curve at any length. From that information you can derive translate and rotate matrices.

As you can see in **Figure 6**, I used a gradient brush for the text so that triangles of slightly different colors would cross paths and intermingle a bit. Even though I'm using only one brush, the desired effect requires the Render method to call SetTransform and FillMesh for every single-triangle mesh. The gradient brush is applied as if the mesh were in its original position prior to the transform.

I wondered if it would be efficient for the Update method to transform all the individual triangles "manually" with calls to the TransformPoint method of the Matrix3x2F class, and to consolidate these in a single ID2D1Mesh object, which would then be rendered with a single FillMesh call. I added an option for that, and sure enough, it was faster. I woudn't have imagined that creating an ID2D1Mesh in each Update call would work well, but it does. The visuals are slightly different, however: The gradient brush is applied to the transformed triangles in the mesh, so there's no intermingling of colors.

Figure 8 **Update and Render in TextMorphing**

```
void TextMorphingRenderer::Update(DX::StepTimer const& timer)
{
  ...

  // Calculate an interpolation factor
  float t = (float)fmod(timer.GetTotalSeconds(), 10) / 10;
  t = std::cos(t * 2 * 3.14159f);     // 1 to 0 to -1 to 0 to 1
  t = (1 - t) / 2;                    // 0 to 1 to 0

  // Two functions for interpolation
  std::function<D2D1_POINT_2F(D2D1_POINT_2F, D2D1_POINT_2F, float)>
    InterpolatePoint =
      [](D2D1_POINT_2F pt0, D2D1_POINT_2F pt1, float t)
  {
    return Point2F((1 - t) * pt0.x + t * pt1.x,
      (1 - t) * pt0.y + t * pt1.y);
  };

  std::function<D2D1_TRIANGLE(D2D1_TRIANGLE, D2D1_TRIANGLE, float)>
    InterpolateTriangle =
      [InterpolatePoint](D2D1_TRIANGLE tri0, D2D1_TRIANGLE tri1, float t)
  {
    D2D1_TRIANGLE triangle;
    triangle.point1 = InterpolatePoint(tri0.point1, tri1.point1, t);
    triangle.point2 = InterpolatePoint(tri0.point2, tri1.point2, t);
    triangle.point3 = InterpolatePoint(tri0.point3, tri1.point3, t);
    return triangle;
  };

  // Interpolate the triangles
  int count = m_triangleInfos.size();
  std::vector<D2D1_TRIANGLE> triangles(count);

  for (int index = 0; index < count; index++)
  {
    triangles.at(index) =
      InterpolateTriangle(m_triangleInfos.at(index).triangle[0],
                          m_triangleInfos.at(index).triangle[1], t);
  }

  // Create a mesh with the interpolated triangles
  m_deviceResources->GetD2DDeviceContext()->CreateMesh(&m_textMesh);
  ComPtr<ID2D1TessellationSink> tessellationSink;
  m_textMesh->Open(&tessellationSink);
  tessellationSink->AddTriangles(triangles.data(), triangles.size());
  tessellationSink->Close();
}

// Renders a frame to the screen
void TextMorphingRenderer::Render()
{
  ...

  if (m_textMesh != nullptr)
  {
    Matrix3x2F centerMatrix = D2D1::Matrix3x2F::Translation(
      (logicalSize.Width - (m_geometryBounds.right + m_geometryBounds.left)) / 2,
      (logicalSize.Height - (m_geometryBounds.bottom + m_geometryBounds.top)) / 2);

    context->SetTransform(centerMatrix *
      m_deviceResources->GetOrientationTransform2D());
    context->SetAntialiasMode(D2D1_ANTIALIAS_MODE_ALIASED);
    context->FillMesh(m_textMesh.Get(), m_blueBrush.Get());
  }

  ...
}
```

Text Morphing?

Suppose you tessellate the glyph outline geometries of two text strings—for example, the words "DirectX" and "Factor" that make up the name of this column—and pair up the triangles for interpolation. An animation could then be defined that transforms one word into the other. It's not exactly a morphing effect, but I don't know what else to call it.

Figure 7 shows the effect midway between the two words, and with a little imagination you can almost make out either "DirectX" or "Factor" in the image.

Optimally, each pair of morphing triangles should be spatially close, but minimizing the distances between all the pairs of triangles is akin to the Traveling Salesman Problem. I took a relatively simpler approach by sorting the two collections of triangles by the X coordinates of the triangle center, and then separating the collections into groups representing ranges of X coordinates, and sorting those by the Y coordinates. Of course, the two triangle collections are different sizes, so some triangles in the word "Factor" correspond to two triangles in the word "DirectX."

Figure 8 shows the interpolation logic in Update and the rendering logic in Render.

With that, I think I've satisfied my curiosity about 2D triangles and I'm ready to give those triangles a third dimension. ■

Charles Petzold *is a longtime contributor to* MSDN Magazine *and the author of "Programming Windows, 6th edition" (Microsoft Press, 2012), a book about writing applications for Windows 8. His Web site is charlespetzold.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Jim Galasyn and Mike Riches*

DAVID S. PLATT

The Peasants Are Revolting!

I've always enjoyed the comic strip, "Wizard of Id," which is set in medieval times. Its creators died in 2008, but their descendants have kept the strip current for today's Internet age (see bit.ly/1d7elYK). Peasants (known, of course, as Idiots) rampage through the town waving signs that read, "The king is a fink!" **Figure 1** shows the king's response.

That same scenario is now exploding in the field of enterprise software. Last December, Avon (the makeup guys) pulled the plug on a new version of their order management software based on SAP. *The Wall Street Journal* in December reported the company's sales force of independent reps "found the new system so burdensome and disruptive to their daily routine that many left Avon."

A spokesman for SAP was later quoted saying that Avon's order management system "is working as designed, despite any issues with the implementation of this project."

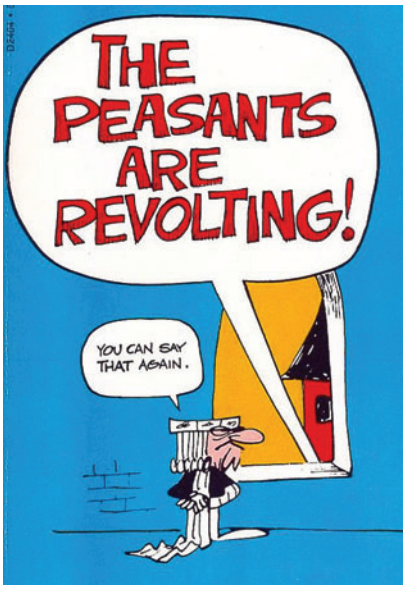


Figure 1 **Avon's management was slow to recognize unrest in the ranks.**

Really? That means unless Avon's goal was to reduce its workforce through bad software instead of layoffs, the company implemented a terrible design. And that weasel spokesman (but, like Mark Twain, I repeat myself) should read my column about the word "issue." (See msdn.microsoft.com/magazine/ff955613.)

As smoking in public was once common, it was once common to force users to contort themselves into five-dimensional hyper-pretzels to match their software—to become "computer literate," in the term of that day. UX guru Alan Cooper wrote that a computer literate user is one who "has been hurt so often that the scar tissue is so thick that he no longer feels the pain." Users accepted this as the price of getting their computing jobs done. That attitude doesn't cut it anymore.

Success in consumer-sector software and hardware has been driven by usability for seven years now, since the first Apple iPhone. But it's taken much longer for that requirement to cross over into the enterprise sector. The whole bring-your-own-device movement arose from early adopter iPhone and iPad users wanting their enterprise software to work as easily as their consumer apps. And now, like the Wizard's newspaper pagemate Popeye the Sailor, enterprise users have stood up and roared, "That's all I can stands! I can't stands no more!" (See bit.ly/1a7BiWZ.)

You'd think that enterprise developers by now would've realized the importance of usability, as they directly benefit from greater user productivity, fewer catastrophic errors, and lower training and support costs. But the strongest bastions of bad usability are places where users are locked in and can't choose. Cormac Herley of Microsoft Research, investigating the burden security policies place on users, found them highest not where data was most sensitive, but rather in captive situations, especially governments and universities, where the enterprise didn't suffer the market consequences of its bad usability (see bit.ly/1eK6Dhu). Avon is the tipping point where this phenomenon starts to change.

Whether you're dealing with the enterprise or consumer sector, UX design has to happen before anything else can. To meet today's standard of care, you can't wait until your program works and then throw it over the fence for the decorators to pretty up. The decorators can round off the corners of the File Open/Save dialog box and give it nice color gradients. But the UX interaction designer determines whether to make the user save documents manually (a la Word), or implement automatic saving (a la OneNote). That choice very much dictates the code to write. So UX design has to come first. And with Avon, clearly, it didn't.

That needs to change. As Steve Rosenbush wrote in his CIO Journal blog on wsj.com: "People who are accustomed to using simple, well-designed applications in their personal lives have no patience for disappointing technology at work." Amen.

And so, my friends, when you work on your enterprise apps, you had better start paying attention to usability. Because the enterprise-sector peasants are indeed revolting. And there's no stopping them. If your boss won't let you put UX first, ask him how he feels about wearing tar and feathers. ■

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.*

Work With Relational Data, Not Complex APIs or Services

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

Give RSSBus a try today and see what mean:

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

rssbus

INTEGRATION YOUR WAY

ASPOSE.TOTAL

Every Aspose component combined in *ONE* powerful suite!

Powerful
File Format APIs

- **Aspose.Words**
  DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.
- **Aspose.Cells**
  XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.
- **Aspose.BarCode**
  JPG, PNG, BMP, GIF, TIF, WMF, ICON & other image formats.
- **Aspose.Pdf**
  PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.
- **Aspose.Email**
  MSG, EML, PST, EMLX & other formats.
- **Aspose.Slides**
  PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.
- **Aspose.Diagram**
  VSD, VSDX, VSS, VST, VSX & other formats.

... and many others!

Aspose.Total for .NET
Aspose.Total for Java
Aspose.Total for Cloud
Aspose.Total for Android

Get your FREE evaluation copy at www.aspose.com

.NET Java Cloud Android

Aspose.Cells

Work with spreadsheets and data without depending on Microsoft Excel

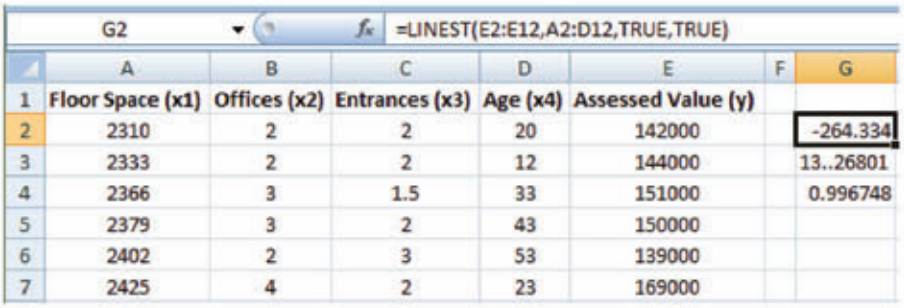
- Solution for spreadsheet creation, manipulation and conversion.
- Import and export data.

ASPOSE.CELLS IS A PROGRAMMING API that allows developers to create, manipulate and convert Microsoft Excel spreadsheet files from within their own applications. Its powerful features make it easy to convert worksheets and charts to graphics or save reports to PDF.

A flexible API for simple and complex spreadsheet programming.

Aspose.Cells speeds up working with Microsoft Excel files. The API is a flexible tool for simple tasks such as file conversion, as well as complex tasks like building models. Developers control page layout, formatting, charts and formulas. They can read and write spreadsheet files and save out to a wide variety of image and text file formats.

Fast and reliable, Aspose.Cells saves time and effort compared to using Microsoft Office Automation.



G2 =LINEST(E2:E12,A2:D12,TRUE,TRUE)

	A	B	C	D	E	F	G
1	Floor Space (x1)	Offices (x2)	Entrances (x3)	Age (x4)	Assessed Value (y)		
2	2310	2	2	20	142000		-264.334
3	2333	2	2	12	144000		13..26801
4	2366	3	1.5	33	151000		0.996748
5	2379	3	2	43	150000		
6	2402	2	3	53	139000		
7	2425	4	2	23	169000		

Aspose.Cells lets developers work with data sources, formatting, even formulas.

Common Uses

- Building dynamic reports on the fly.
- Creating Excel dashboards with charts and pivot tables.
- Rendering and printing spreadsheets and graphics with high fidelity.
- Exporting data to, or importing from, Excel spreadsheets.
- Generating, manipulating and editing spreadsheets.
- Converting spreadsheets to images or other file formats.

Key Features

- A complete spreadsheet manipulation solution.
- Flexible data visualization and reporting.
- Powerful formula engine.
- Complete formatting control.

Supported File Formats

XLS, XLSX, XLSM, XMPS, XLTX, XLTM, ODS, SpreadsheetML, tab delim., CSV, TXT, PDF, HTML, and many image formats including TIFF, JPEG, PNG and GIF.

Format support varies across platforms.

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$999	$1498	Site Small Business	$4995	$7490
Developer OEM	$2997	$4494	Site OEM	$13986	$20972

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

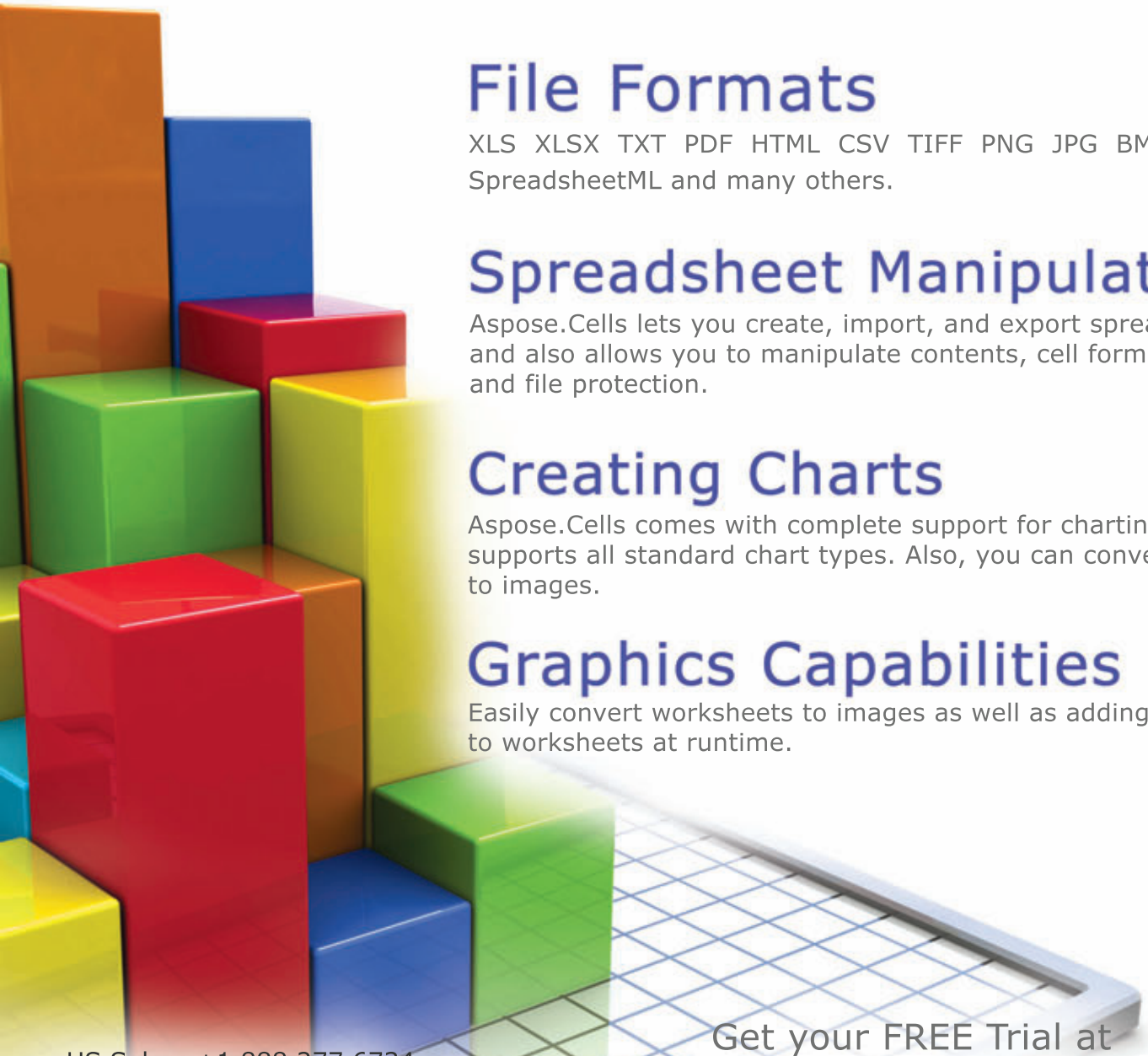
www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Aspose.Cells for
.NET, Java, Cloud & Android



File Formats

XLS XLSX TXT PDF HTML CSV TIFF PNG JPG BMP SpreadsheetML and many others.

Spreadsheet Manipulation

Aspose.Cells lets you create, import, and export spreadsheets and also allows you to manipulate contents, cell formatting, and file protection.

Creating Charts

Aspose.Cells comes with complete support for charting and supports all standard chart types. Also, you can convert charts to images.

Graphics Capabilities

Easily convert worksheets to images as well as adding images to worksheets at runtime.

Get your FREE Trial at
http://www.aspose.com

US Sales: +1 888 277 6734
FAX: +1 866 810 9465
sales@aspose.com

EU Sales: +44 141 416 1112
sales.europe@aspose.com

ASPOSE
Your File Format APIs

No Office Automation

Aspose.Cells does not require Microsoft Office to be installed on the machine in order to work.

Aspose.Words

Program with word processing documents independently of Microsoft Word

- Solution for document creation, manipulation and conversion.
- Advanced mail merge functionality.

ASPOSE.WORDS IS AN ADVANCED PROGRAMMING API that lets developers perform a wide range of document processing tasks with their own applications. Aspose.Words makes it possible to generate, modify, convert, render and print documents without Microsoft Office Automation. It provides sophisticated and flexible access to, and control over, Microsoft Word files.

Aspose. Words is powerful, user-friendly and feature rich. It saves developers time and effort compared to using Microsoft Office Automation and makes gives them powerful document management tools.

Generate, modify, convert, render and print documents without Microsoft Office Automation.

Aspose.Words makes creating, changing and converting DOC and other word processing file formats fast and easy.

	Table			
	Column 1	**Column 2**	**Column 3**	**Column 4**
Row 1	Cell 1	Cell 2	Cell 3	Cell 4
Row 2	Cell 1	Cell 2	Cell 3	
Row 3	Cell 1	Cell 2		

Aspose.Words has sophisticated controls for formatting and managing tables and other content.

Common Uses

- Generating reports with complex mail merging; mail merging images.
- Populating tables and documents with data from a database.
- Inserting formatted text, paragraphs, tables and images into Microsoft Word documents.
- Adding barcodes to documents.
- Inserting diagrams and watermarks into Word documents.
- Formatting date and numeric fields.

Key Features

- A complete Microsoft Word document manipulation solution.
- Extensive mail merge features.
- Complete formatting control.
- High-fidelity conversion, rendering and printing.

Supported File Formats

DOC, DOCX, ODT, OOXML, XML, HTML, XHTML, MHTML, EPUB, PDF, XPS, RTF, and a number of image formats, including TIFF, JPEG, PNG and GIF.

Format support varies across platforms.

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$999	$1498	Site Small Business	$4995	$7490
Developer OEM	$2997	$4494	Site OEM	$13986	$20972

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Case Study: Aspose.Words for .NET

Lulu helps authors, publishers, businesses, and educators publish and sell print on demand books and ebooks. Why do they use Aspose?

LULU IS A TECHNOLOGY COMPANY THAT PROVIDES AN OPEN PUBLISHING PLATFORM where customers from all over the world can create, publish and sell print-on-demand books, ebooks, photobooks and calendars.

The basic function of Lulu's publishing platform is to receive manuscripts from customers and send them to printers for printing. The printers receive the manuscripts in PDF format but that is not always its original format.

Customers can submit manuscripts in any number of formats: many use Microsoft Word. Lulu's publishing platform converts incoming manuscripts to PDFs that can then be sent to the printer. The conversion is automatic: the document comes in, is converted and goes off to print withouthuman intervention or review.

Updating the Platform

Lulu has been running for several years. The original conversion platform depended on Microsoft Automation for converting DOC files to PDFs. As the business grew and had to accommodate a much higher number of manuscripts, some problems with the existing platform became apparent.

- It did not scale,
- it did not support the latest Microsoft Word document formats, and
- it was not robust.

Looking for a Solution

> It took only 10 lines of code to integrate Aspose.Words for .NET into the new solution.

The company decided to build a new platform using components that could support their continued growth.

Lulu's engineering team tested Aspose.Words for .NET alongside the existing Microsoft Automation system and other applications. Each solution had its strengths and weaknesses but in the end, Aspose.Words for .NET won because

- it took only 10 lines of code to integrate it into the new platform,
- it is robust and scalable,
- it supports all the file formats that Lulu needs, and
- the licensing structure is straight-forward and cheaper over time than other solutions.

Outcome

The result was a product that can take any Microsoft Word document that a customer submits, regardless of how the customer may have embellish their manuscript, and convert it to a PDF file that can be printed anywhere.

This is an extract from a case study on our website. For the full version, go to: www.aspose.com/corporate/customers/case-studies.aspx

Customers create manuscripts that can be printed by any printer.

GroupDocs™
work faster. together..

http://groupdocs.com
sales@groupdocs.com

POWERFUL DOCUMENT MANAGEMENT LIBRARIES AND CLOUD APIS FOR YOUR WEB AND MOBILE APPLICATIONS

.NET, JAVA, CLOUD APIS

GroupDocs
Viewer

True text, high-fidelity embedded document viewer with support for over 50 file formats.

GroupDocs
Signature

Electronic signature capture API that gives your apps legally binding e-signature capabilities.

GroupDocs
Annotation

A powerful API that lets you annotate Microsoft Office, PDF and other documents within your own applications.

GroupDocs
Assembly

Incorporates data entered by users in online forms into PDF and Microsoft Office documents using merge fields.

GroupDocs
Conversion

Universal document converter with an independent engine for fast conversion between more than 50 file formats.

GroupDocs
Comparison

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

Sales: +1 (214) 329-9760

Powerful Document Management Solutions for Your Web and Mobile Applications

GroupDocs offers professional stand-alone .NET & Java libraries along with cloud APIs that allow end-users to view, annotate, convert, e-sign, assemble and compare documents and images of more than 35 file formats within your own web and mobile applications. Key benefits include:

- All GroupDocs APIs are 100% independent, and don't require any 3rd party software installation.
- Being extremely lightweight, .NET & Java libraries can be integrated with just a single DLL.
- Cloud APIs are supported by SDKs to help developers on .NET, Java, JavaScript, PHP, Python and Ruby seamlessly integrate GroupDocs solutions into any web or mobile apps.
- No need for client-side installation. End-users can work with documents from any web-enabled device and modern web-browser.
- All GroupDocs' products come with a 30-day fully-functional trial and free support during the integration period.

Stand-Alone .NET & Java Libraries Pricing

GroupDocs .NET & Java licenses are based on the number of developers and the number of locations where the components will be used:

	Developer Small Business	Developer OEM	Site Small Business	Site OEM
License for one developer	✓	✓		
Licenses for up to 10 developers			✓	✓
Use derived work at one location	✓			
Use derived work at up to 10 locations			✓	
Royalty free/deploy to unlimited locations		✓		✓
Discount applied to multiple purchases	✓	✓	✓	✓
Can be used to create unlimited applications	✓	✓	✓	✓
Updates and hotfixes for one year	✓	✓	✓	✓
Free technical support	✓	✓	✓	✓
Price	**$2,499**	**$7,497**	**$9,996**	**$29,988**

Cloud API Pricing

GroupDocs' cloud APIs use a different licensing model. Instead of licenses, they are charged by use: the number of calls made to the API. To find out more about our cloud API pricing, please visit our website: www.groupdocs.com

GroupDocs Viewer

A powerful document viewer API that allows you to display over 35 document formats in your web or mobile application. The viewer can both rasterize documents and convert them to SVG+HTML+CSS, delivering true-text high-fidelity rendering.
Supported file formats include: Microsoft Office, Visio, Project and Outlook documents, PDFs, AutoCAD, image files (TIFF, JPG, BMP, GIF, TIFF, etc.) and more.

GroupDocs Annotation

With support for over 35 file formats, this API allows your app users to annotate documents of all common business formats, including Microsoft Office and PDF. And thanks to the advanced document management options, users can store, share, print, download and export the annotated documents easily - all from within your own application.

GroupDocs Conversion

GroupDocs Conversion API allows end users to convert back and forth between over 25 document formats within your own application. It supports all Microsoft Office document formats as well as PDF, HTML and common image file formats (TIFF, JPEG, GIF, PNG, BMP). Your users can convert documents one by one on the fly, or add several documents at a time to a conversion queue.

GroupDocs Signature

GroupDocs Signature API is an easy way to give your apps legally binding e-signature capabilities. Your users are then able to get documents signed electronically using only a web-browser.
The API gives developers access to sophisticated online signature features, from e-signature capture control and different signing workflows, to reminder management, contact management and signer roles.

GroupDocs Assembly

GroupDocs Assembly automatically incorporates data submitted through online forms into existing document templates in PDF or Microsoft Word formats. For each completed form a new custom document is generated. The API lets you build document assembly solutions without getting into the details of working with templates, fields and merging data.

GroupDocs Comparison

A document comparison API that allows users to quickly and easily find differences between two revisions of a document right in your web or mobile app. It merges two uploaded documents into a single one and displays it, highlighting differences with the redline view approach - similar to the Microsoft Word change tracking feature, but online. Works with Microsoft Word, Excel, and PowerPoint documents, as well as Adobe Acrobat PDF files.

Adding File Conversion and Manipulation to Business Systems

How often do people in your organization complain that they can't get information in the file format and layout they want? Converting documents from one format to another without losing layout and formatting should be simple, but it can be frustrating for both users and developers.

EXTRACTING DATA FROM A DATABASE AND DELIVERING IT TO THE SALES TEAM AS A REPORT, complete with charts and corporate branding, is fine. Until the sales team says that they want it as a Microsoft Excel file, and could you add a dashboard?

Using information from online forms in letters that can are printed and posted is easy. But what if you also want to add tracking barcodes and archive a digital copy as a PDF?

Aspose creates APIs that work independently of Microsoft Office Automation.

Ensuring that your business system supports all the different Microsoft Office file formats your users want can be difficult. Sometimes the native file format support of your system lets you down. When that is the case, use tools that extend that capability. A good tool can save you time and effort.

Document Conversion Options

Building your own solution: Time-consuming and costly, this option is only sensible if the solution you develop is central to your business.

Using Microsoft Office Automation: Microsoft Office Automation lets you use Microsoft Office programs server-side. It is not how the Office products were designed to be used. It can work well but you might notice issues with the stability, security and speed of the system, as well as cost.

Using an API: The API market has lots of free and commercial solutions, some very focused, some feature-rich. An API integrates with your code and gives you access to a range of new features.

Look to Aspose

Aspose are API experts. We create APIs, components and extensions that work independently of Microsoft Automation to extend a platform's native file format manipulation capabilities.

Aspose have developed APIs for .NET, Java, Cloud and Android that lets developers convert, create and manipulate Microsoft Office files – Microsoft Word, Excel, PowerPoint, Visio and Project – and other popular business formats, from PDFs and images to emails. We also have APIs for working with images, barcodes and OCR. The APIs are optimised for stability, speed and ease of use. Our APIs save users weeks, sometimes months, of effort.

Finding the Right Tool

To find the product that's right for you, take a systematic approach:

- List must-have and nice-to-have features.
- Research the market.
- Ask for recommendations.
- Select a few candidates .
- Run trials.
- Evaluate
  - ease of use,
  - support and documentation,
  - performance, and
  - current and future needs.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Aspose.BarCode

A complete toolkit for barcode generation and recognition

- Generate barcodes with customer defined size and color.
- Recognize a large number of barcode types from images.

ASPOSE.BARCODE IS A ROBUST AND RELIABLE BARCODE GENERATION AND RECOGNITION API that allows developers to add barcode generation and recognition functionality to their applications quickly and easily.

Aspose.BarCode supports most established barcode specifications. It can export generated barcodes to multiple image formats, including BMP, GIF, JPED, PNG and TIFF.

Aspose.BarCode gives you full control over every aspect of the barcode image, from background and bar color, through image quality, rotation angle, X-dimension, captions, and resolution.

Robust and reliable barcode generation and recognition.

Aspose.BarCode can read and recognize most common 1D and 2D barcodes from any image and at any angle. Filters help developers clean up difficult to read images to improve recognition.

Aspose.BarCode offers a large number of symbologies and formatting options.

Common Uses

- Generating and recognizing barcode images.
- Printing barcode labels.
- Enhancing workflow by adding barcode functionality.
- Using recognition functions to drive real-life work processes.

Key Features

- Barcode generation and recognition.
- Comprehensive support for 1D and 2D symbologies.
- Image processing for improved recognition.

Supported File Formats

JPG, TIFF, PNG, BMP, GIF, EMF, WMF, EXIP and ICON.

Format support varies across platforms.

Supported Barcodes

Linear: EAN13, EAN8, UPCA, UPCE, Interleaved2of5, Standard2of5, MSI, Code11, Codabar, EAN14(SCC14), SSCC18, ITF14, Matrix 2 of 5, PZN, Code128, Code39 Extended, Code39 Standard, OPC, Code93 Extended, Code93 Standard, IATA 2 of 5, GS1Code128, ISBN, ISMN, ISSN, ITF6, Pharmacode, DatabarOmniDirectional, VIN, DatabarTruncated, DatabarLimited, DatabarExpanded, PatchCode, Supplement **2D**: PDF417, MacroPDF417, DataMatrix, Aztec, QR, Italian Post 25, Code16K, GS1DataMatrix **Postal**: Postnet, Planet, USPS OneCode, Australia Post, Deutsche Post Identcode, AustralianPosteParcel, Deutsche Post Leticode, RM4SCC, SingaporePost, SwissPostParcel

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$599	$1098	Site Small Business	$2995	$5490
Developer OEM	$1797	$3294	Site OEM	$8386	$15372

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Aspose.Email

Work with emails and calendars without Microsoft Outlook

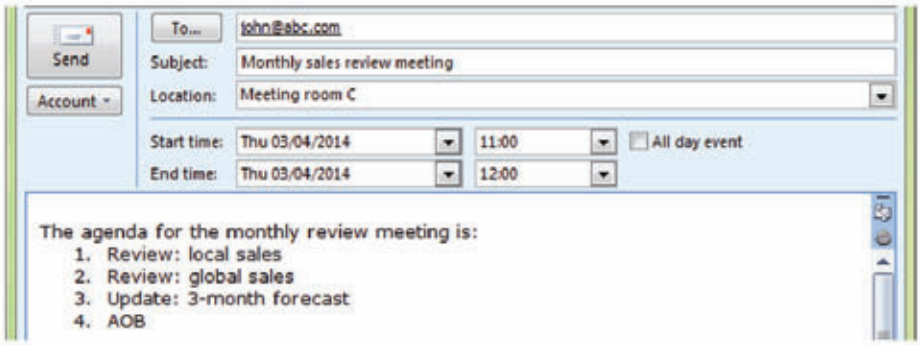
- Complete email processing solution.
- Message file format support.

ASPOSE.EMAIL IS AN EMAIL PROGRAMMING API that allows developers to access and work with PST, EML, MSG and MHT files. It also offers an advanced API for interacting with enterprise mail systems like Exchange and Gmail.

Aspose.Email can work with HTML and plain text emails, attachments and embedded OLE objects. It allows developers to work against SMTP, POP, FTP and Microsoft Exchange servers. It supports mail merge and iCalendar features, customized header and body, searching archives and has many other useful features.

Aspose.Email works with HTML and plain text emails, attachments and embedded OLE objects.

Aspose.Email allows developers to focus on managing email without getting into the core of email and network programming. It gives you the controls you need.



Aspose.Email lets your applications work with emails, attachments, notes and calendars .

Common Uses

- Sending email with HTML formatting and attachments.
- Mail merging and sending mass mail.
- Connecting to POP3 and IMAP mail servers to list and download messages.
- Connecting to Microsoft Exchange Servers to list, download and send messages.
- Create and update tasks using iCalendar.
- Load from and save messages to file or stream (EML, MSG or MHT formats).

Key Features

- A complete email processing solution.
- Support for MSG and PST formats.
- Microsoft Exchange Server support.
- Complete recurrence pattern solution.

Supported File Formats

MSG, MHT, OST, PST, EMLX, TNEF, and EML.

Format support varies across platforms.

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$599	$1059	Site Small Business	$2995	$5490
Developer OEM	$1797	$3294	Site OEM	$8386	$15372

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Aspose.Pdf

Create PDF documents without using Adobe Acrobat

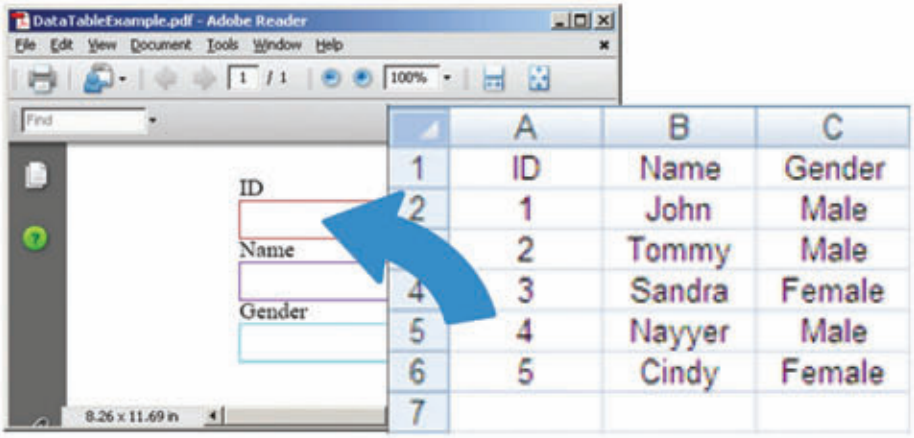
- A complete solution for programming with PDF files.
- Work with PDF forms and form fields.

ASPOSE.PDF IS A PDF DOCUMENT CREATION AND MANIPULATION API that developers use to read, write and manipulate PDF documents without using Adobe Acrobat. Aspose.Pdf is a sophisticated product that integrates with your application to add PDF capabilities.

Aspose.Pdf offers a wealth of features that lets developers compress files, create tables, work with links, add and remove security, handle custom fonts, integrate with external data sources, manage bookmarks, create table of contents, create forms and manage form fields.

Read, write and manipulate PDF documents independently of Adobe Acrobat.

It helps developers add, work with attachments, annotations and PDF form data, add, replace or remove text and images, split, concatenate, extract or inset pages, and print PDF documents.



Aspose.Pdf can be used to automatically complete PDF forms with external data.

Common Uses

- Creating and editing PDF files.
- Inserting, extracting, appending, concatenating and splitting PDFs.
- Working with text, images, tables, images, headers, and footers.
- Applying security, passwords and signatures.
- Working with forms and form fields.

Key Features

- PDF creation from XML or XLS-FO documents.
- PDF form and field support.
- Advanced security and encryption.
- High-fidelity printing and conversion.
- Supported File Formats
- PDF, PDF/A, PDF/A_1b, PCL, XLS-FO, LaTeX, HTML, XPS, TXT and a range of image formats.

Format support varies across platforms.

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$799	$1298	Site Small Business	$3995	$6490
Developer OEM	$2397	$3894	Site OEM	$11186	$18172

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Banckle
Customer Service 2.0

http://banckle.com/
sales@banckle.com

Browser Based

Powerful APIs

Device Compatibility

Customer Service Tools that Run in a Web-Browser

Banckle's online customer service tools help companies large and small engage with customers. Just log in and go: no plugin or software installation necessary. Platform, OS and browser independent.

Build powerful SaaS apps with Banckle's robust Cloud APIs.

Contact **sales@banckle.com** for API pricing information and an extended 30 day free trial. Alternatively, visit **banckle.com** and use promo code **MiniMag2014** at checkout for a 20% discount.

Banckle.com - Customer Service 2.0

Customer Service Apps that Help Grow Your Business

Banckle's professional web apps help companies engage with customers.

Banckle Chat

Chat live with your customers to give them the help they want, when they need it. Embeddable HTML makes it easy to integrate Banckle's live chat service into your website.

Prices start from $6.30/month.

Banckle Meeting

Give your customers a great online meeting experience with an intuitive interface. Share screen, presentations, work with a whiteboard and use video conferencing from a web browser.

Prices start from $10.50/month.

Banckle CRM

Keep the team up to date with projects and customer contacts. Keep an eye on the pipeline, manage tasks and log contacts in one central place to improve team work and customer focus.

Prices start from $4.20/month.

Banckle Helpdesk

Stay on top of your customers' issues with an online service desk and ticketing system. Never lose a ticket but let the team collaborate to deliver the best possible customer support.

Prices start from $11.00/month.

Banckle Campaign

Find out how effective your email campaigns are and keep in touch with your customers. Design, test and send campaigns, manage mailing lists and learn from campaign reports.

Prices start from $9.80/month.

Banckle Email

Manage email from an affordable, secure and intuitive online platform. Use all the email features you are used to: attachments, folders and simple account administration.

Prices start from $13.30/month.

Banckle Total

All Banckle's customer service tools in one package, Banckle Total helps you take customer support to the next level.

Subscriptions start at $31.00/month.

Try Our APIs

Want to build your own solution?, Consider our RESTful API's

Contact **sales@banckle.com** for API pricing information and an extended 30 day free trial. Alternatively, visit **banckle.com** and use promo code **MiniMag2014** at checkout for a 20% discount.

Aspose.Slides

Work with presentations without using Microsoft PowerPoint

- Complete solution for working with presentation files.
- Export presentations and slides to portable or image formats.

ASPOSE.SLIDES IS A FLEXIBLE PRESENTATION MANAGEMENT API that helps developers read, write and manipulate Microsoft PowerPoint documents. Slides and presentations can be saved to PDF, HTML and image file formats without Microsoft Office Automation.

Aspose.Slides offers a number of advanced features that make it easy to perform tasks such as rendering slides, exporting presentations, exporting slides to SVG and printing. Developers use Aspose.Slides to build customizable slide decks, add or remove standard graphics and automatically publish presentations to other formats.

> Aspose.Slides gives you the tools you need to work with presentation files.

Aspose.Slides gives developers the tools they need to work with presentation files. It integrates quickly and saves time and money.

Aspose.Slides has advanced features for working with every aspect of a presentation.

Common Uses

- Creating new slides and cloning existing slides from templates.
- Handling text and shape formatting.
- Applying and removing protection.
- Exporting presentations to images and PDF.
- Embedding Excel charts as OLE objects.
- Generate presentations from database.

Key Features

- A complete presentation development solution.
- Control over text, formatting and slide elements.
- OLE integration for embedding external content.
- Wide support for input and output file formats.

Supported File Formats

PPT, POT, PPS, PPTX, POTX, PPSX, ODP, PresentationML, XPS, PDF and image formats including TIFF and JPG.

Format support varies across platforms.

Platforms

Pricing Info					
	Standard	Enhanced		Standard	Enhanced
Developer Small Business	$799	$1298	Site Small Business	$3995	$6490
Developer OEM	$2397	$3894	Site OEM	$11186	$18172

The pricing info above is for .NET: prices for other platforms may differ. For the latest, contact sales.

www.aspose.com

US: +1 888 277 6734
sales@aspose.com

EU: +44 141 416 1112
sales.europe@aspose.com

Oceania: +61 2 8003 5926
sales.asiapacific@aspose.com

Support Services

Get the assistance you need, when you need it, from the people who know our products best.

- Use experienced Aspose developers for your projects
- Get the level of support that suits you and your team

NO ONE KNOWS OUR PRODUCTS AS WELL AS WE DO. We develop them, support them and use them. Our experience is available to you, whether you want us to develop a solution for you, or you just need a little help to solve a particular problem.

Consulting

Aspose's developers are expert users of Aspose APIs. They understand how to use our products and have hands-on experience of using them for software development. Aspose's developers are skilled not just with Aspose tools but in a wide range of programming languages, tools and techniques.

Aspose's file format experts are here to help you with a project or your support questions

When you need help to get a project off the ground, Aspose's developers can help.

Work with the most experienced Aspose developers in the world.

Consulting Benefits

- Use Aspose engineers to work on your products
- Get peace of mind from a fully managed development process
- Get a custom-built solution that meets your exact needs

Support Options

Free

Everyone who uses Aspose products have access to our free support. Our software developers are on stand-by to help you succeed with your project, from the evaluation to roll-out of your solution.

Priority

If you want to know when you'll hear back from us on an issue, and know that your issue is prioritized, Priority Support is for you. It provides a more formal support structure and has its own forum that is monitored by our software engineers.

Enterprise

Enterprise customers often have very specific needs. Our Enterprise Support option gives them access to the product development team and influence over the roadmap. Enterprise Support customers have their own, dedicated issue tracking system.

Pricing Info
Each consulting project is evaluated individually; no two projects have exactly the same requirements.
To see the Priority and Enterprise support rates, refer to the product price list, or contact our sales team.

We're Here to Help YOU

Aspose has 4 Support Services to best suit your needs

Free Support	Support Forums with no Charge
Priority Support	24 hour response time in the week, issue escalation, dedicated forum
Enterprise Support	Communicate with product managers, influence the roadmap
Sponsored Support	Get the feature you need built now

Technical Support is an issue that Aspose takes very seriously. Software must work quickly and dependably. When problems arise, developers need answers in a hurry. We ensure that our clients receive useful answers and solutions quickly.

Email • Live Chat • Forums

CONTACT US

US Sales: +1 888 277 6734
sales@aspose.com

EU Sales: +44 141 416 1112
sales.europe@aspose.com

AU Sales: +61 2 8003 5926
sales.asiapacific@aspose.com