THE MICROSOFT JOURNAL FOR DEVELOPERS

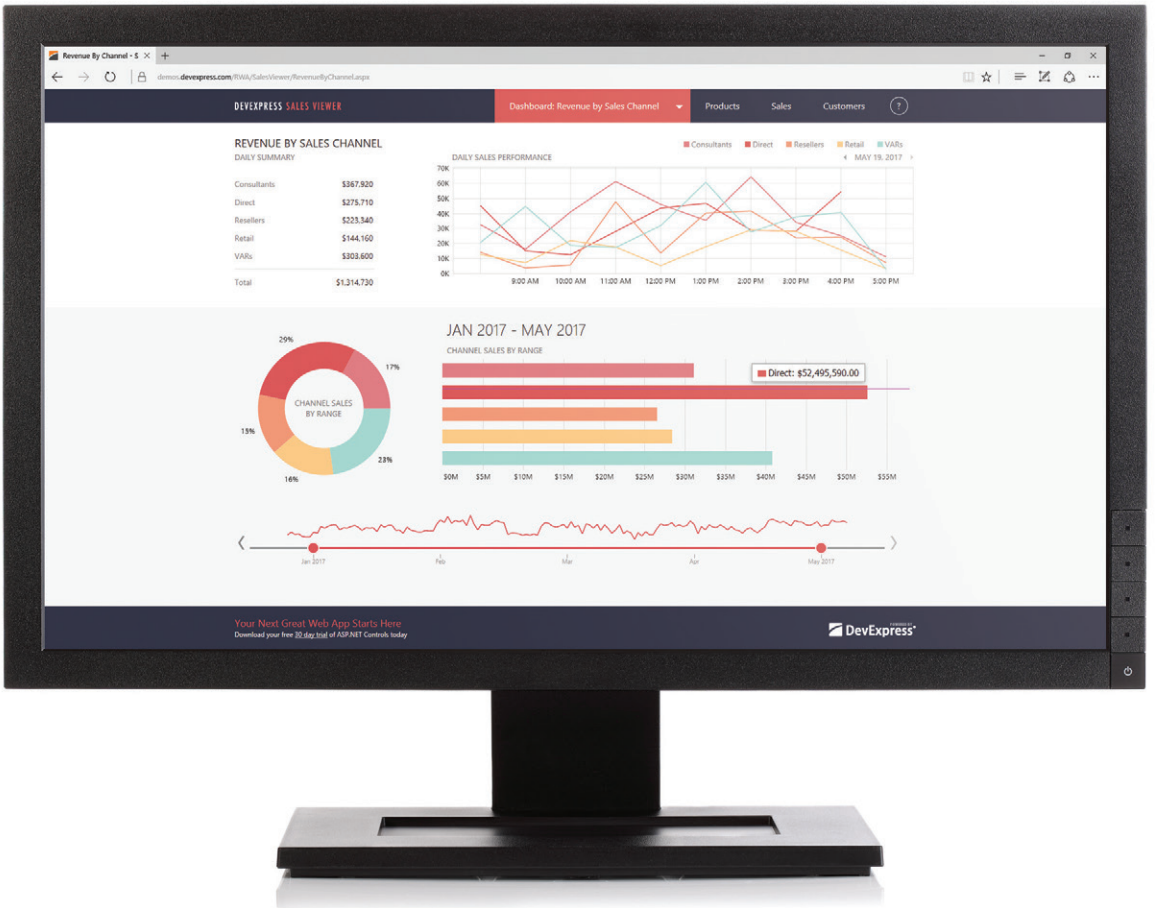
MARCH 2018 VOL 33 NO 3

msdn magazine

DevExpress®

Your Next Great Web App Starts Here

From apps that replicate the look and feel of Microsoft Office® 365, to high-impact decision support systems for your enterprise, DevExpress Web Controls for ASP.NET will help you build your best, without limits or compromise.



Download your free 30-day trial
and experience the DevExpress difference today.

devexpress.com/try

THE MICROSOFT JOURNAL FOR DEVELOPERS

MARCH 2018 VOL 33 NO 3

msdn magazine

COLUMNS

Microsoft

INFRAGISTICS

Infragistics Ultimate 17.2

Productivity Tools & Fast Performing UI Controls for Quickly Building Web, Desktop, & Mobile Apps

Includes 100+ beautifully styled, high-performance grids, charts & other UI controls, plus visual configuration tooling, rapid prototyping, and usability testing.

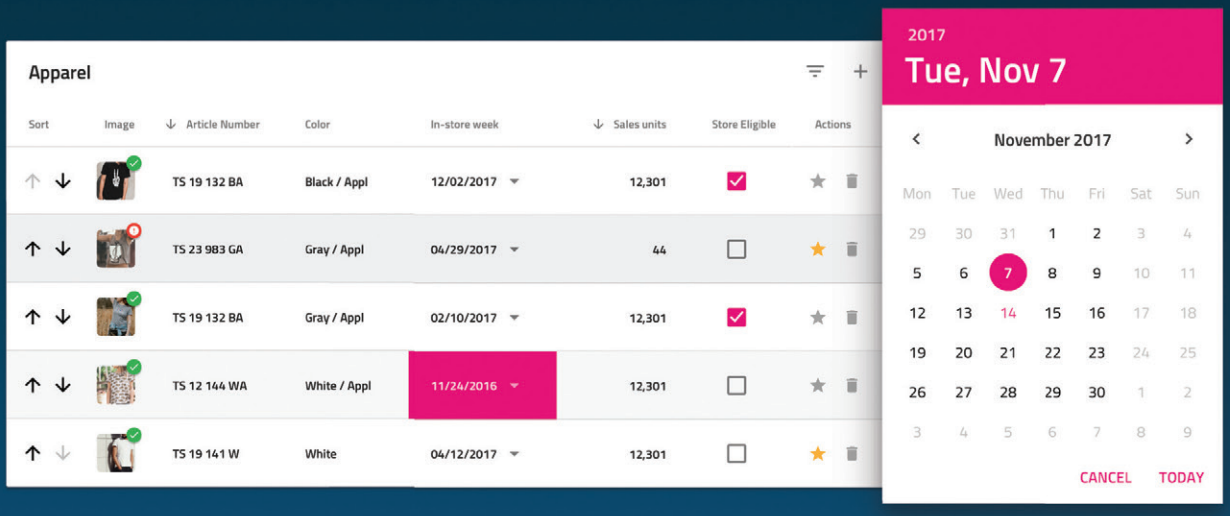
Angular | JavaScript / HTML5 | ASP.NET | Windows Forms | WPF | Xamarin

Download a free trial at
Infragistics.com/Ultimate

Featuring

Ignite UI

A complete UI component library for building high-performance, data rich web applications



- Create beautiful, touch-first, responsive desktop & mobile wep apps with over 100 JavaScript / HTML5, MVC & **Angular components**.
- Our easy to use Angular components have no 3rd party dependencies, a tiny footprint, and easy-to-use API.
- The Ignite UI **Angular Data Grid** enables you to quickly bind data with little coding - including features like sorting, filtering, paging, movable columns, templates and more!
- Speed up development time with responsive layout, powerful data binding, cross-browser compatibility, WYSIWYG page design, & built-in-themes.

Download a free trial of Ignite UI at: **Infragistics.com/Ignite-ui**

To speak with our sales team or request a product demo call: 1.800.321.8588

msdn magazine

MARCH 2018 VOLUME 33 NUMBER 3

ID STATEMENT MSDN Magazine (ISSN 1528-4859) is published 13 times a year, monthly with a special issue in November 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.

CORPORATE ADDRESS 1105 Media, 9201 Oakdale Ave. Ste 101, Chatsworth, CA 91311 1105media.com

MEDIA KITS Direct your Media Kit requests to Chief Revenue Officer Dan LaBianca, 972-687-6702 (phone), 972-687-6799 (fax), dlabianca@1105media.com

REPRINTS For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International
Phone: 212-221-9595
E-mail: 1105reprints@parsintl.com
Web: 1105Reprints.com

LIST RENTAL This publication's subscriber list is not available for rental. However, other lists from 1105 Media, Inc. can be rented.
For more information, please contact our list manager:
Jane Long, Merit Direct
Phone: 913-685-1301;
E-mail: jlong@meritdirect.com;
Web: meritdirect.com/1105

Reaching the Staff

Staff may be reached via e-mail, telephone, fax, or mail.
E-mail: To e-mail any member of the staff, please use the following form: FirstinitialLastname@1105media.com
Irvine Office (weekdays, 9:00 a.m. – 5:00 p.m. PT)
Telephone 949-265-1520; Fax 949-265-1528
4 Venture, Suite 150, Irvine, CA 92618
Corporate Office (weekdays, 8:30 a.m. – 5:30 p.m. PT)
Telephone 818-814-5200; Fax 818-734-1522
9201 Oakdale Avenue, Suite 101, Chatsworth, CA 91311
The opinions expressed within the articles and other contentsherein do not necessarily express those of the publisher.

Microsoft

BPA WORLDWIDE
1906 2006 100 AMERICAN BUSINESS MEDIA
WPA Western Publications Association

LEADTOOLS® V20

WEB MOBILE DESKTOP SERVER

WHAT DOES YOUR CODE LOOK LIKE?

SEARCH ocrdemo.cs

```
using System;
using Leadtools;
using Leadtools.Ocr;
using Leadtools.Document.Writer;

namespace LEADOcrSimpleDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         //Set the LEADTOOLS evaluation license
         RasterSupport.SetLicense(@"License\LEADTOOLS.LIC",
            System.IO.File.ReadAllText(@"License\LEADTOOLS.LIC.KEY"));
         //Specify input and output parameters
         string inputFile = @"C:\Users\Public\Documents\LEADTOOLS Images\OCR1.TIF";
         string outputFile = @"C:\Users\Public\Documents\LEADTOOLS Images\OCR1.PDF";

         //Call OCRImage method
         OCRImage(inputFile, outputFile);
      }

      static void OCRImage(string inputFile, string outputFile)
      {
         Console.WriteLine($"Loading and recognizing {inputFile}");

         //Initiate the LEADTOOLS OCR Engine
         using (IOcrEngine ocrEngine = OcrEngineManager.CreateEngine(OcrEngineType.LEAD, false))
         {
            //Startup the LEADTOOLS OCR Engine
            ocrEngine.Startup(null, null, null, null);
            //Run the AutoRecognizeManager and specify PDF format
            ocrEngine.AutoRecognizeManager.Run(inputFile, outputFile, DocumentFormat.Pdf, null, null);
            Console.WriteLine($"OCR output saved to {outputFile}");
         }
      }
   }
```

Above is an entire OCR application using LEADTOOLS

That's it. Clean and simple code that produces a fast and accurate OCR app backed by the most powerful OCR SDK. Download our free evaluation and see how easy it is to use LEADTOOLS for your document, medical, and multimedia projects.

Microsoft® .NET · X · Windows · Java · Linux · iOS · macOS · Android · HTML5

DOWNLOAD OUR FREE EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

Editor's Note

MICHAEL DESMOND

Chain of Demand

I've been alive long enough to enjoy a good, irrational market run-up. The stratospheric rise of Bitcoin could rank up there with the Dutch tulip mania of the 1630s, the dot-com bubble of the 1990s and Florida real estate circa 2007. All of which proves that people have an amazing ability to crave a thing, especially if they don't understand it.

Yet the hype and chicanery around cryptocurrencies can obscure the exciting technology that enables them: blockchains. As Jonathan Waldman explains in this month's lead feature, "Blockchain Fundamentals," the distributed technology rooted in cryptography and consensus algorithms has the potential to redefine transactional processes by making them secure, tamper-proof, permanent and even public.

Yet all the hype and chicanery around cryptocurrencies can obscure the exciting technology that enables them: blockchains.

"While still in their infancy, blockchain technologies are being used to power logistics and asset-tracking systems," Waldman explains over e-mail, before ticking off a list of scenarios that includes distributed cloud storage services, political voting systems, personal identification systems, digital notary services, and management of sensitive health care data. And blockchains can be used to document the provenance and thus authenticity of valuable goods. Waldman offers the example of digital IDs being laser-etched into diamonds by jewelers and into wine bottles by vintners.

Waldman says blockchain implementations have solved a fundamental challenge, by creating a "disintermediated digital infrastructure on which a digital asset can be openly and reliably transferred, rather than copied and shared." He describes blockchains as a data structure that tracks any asset of value or interest as it is transferred from owner to owner.

While blockchains are built on well-known concepts like hashing, cryptography and decentralized peer-based relationships, the number and diversity of blockchain solutions pose a challenge to developers. Blockchain implementations are often poorly documented beyond the source code, Waldman says, which can complicate decision making.

"One misconception I see is that blockchain technologies require 'no trust,'" says Waldman. "While a traditional, human-staffed, centralized trust authority isn't necessary, the technologies and algorithms that power blockchains must in fact be trusted. Due to the complexity and number of current blockchain implementations, it remains unclear which ones are truly trustworthy."

Waldman urges developers to consider a number of factors when deciding to adopt a blockchain implementation, including who's behind it, whether it's public or private, and the quality of its technical documentation and end-user apps. He also tells developers when vetting a particular blockchain to look into the kind of data it records, its age and adoption rate, and the number of discovered vulnerabilities. Ultimately, says Waldman, "the blockchain becomes the trust agent."

The best way to get up to speed, he says, is to pick an existing blockchain implementation and pore over its technical documentation. He also urges developers to explore the Microsoft Azure-hosted blockchain service at azure.microsoft.com/solutions/blockchain.

"I think that establishing trust is imperative if blockchain is going to become a widely deployed and effective tool," Waldman says. "I think it's fair to say that disintermediating untrustworthy agents and middlemen and replacing them with rock-solid, time-tested blockchain technologies is a natural, prudent, and inevitable step in the evolution of our digital lives."

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

DevExpress®

DevExpress Spreadsheet for WPF & WinForms

with Chart, Pivot Table and Cell Editor support

Your users already know Microsoft® Excel® and with the new capabilities we've introduced in our Spreadsheet Control, you can create solutions that utilize a "spreadsheet-first" UX for a broad range of use-case scenarios. Imagine sales entry forms that are an actual invoice or an inventory management document with built-in ordering and data entry options.

Give our Spreadsheet a try today and see how easy it is to build Windows® apps that amaze.

Free 30-day trial
devexpress.com/spreadsheet

#UseTheBest

JULIE LERMAN

Calling Azure Functions from the Universal Windows Platform

This is the final installment of my series on building a Universal Windows Platform (UWP) app that stores data locally and in the cloud. In the first installment, I built the UWP CookieBinge game, which uses Entity Framework Core 2 (EF Core 2) to store game scores onto the device on which the game is being played. In the next two installments, I showed you how to build Azure Functions in the cloud to store the game scores to and retrieve them from a Microsoft Azure Cosmos DB database. Finally, in this column, you'll see how to make requests from the UWP app to those Azure Functions to send scores, as well as receive and display top scores for the player across all of her devices in addition to top scores for all players around the globe.

This solution also enables users to register in the cloud and tie any of their devices to that registration. While I'll leverage that registration information to send and retrieve scores from the Azure Functions, I won't be describing that part of the application. However, you can see that code in the download, including code for the new Azure Functions I created for registration.

Figure 1 **Scores from the Azure Cosmos DB Database Retrieved from Azure Functions**

The app also has a feature that displays the top five scores from the local database.

There are three Azure Functions that I built in my last few columns. The first receives the game score, along with the player's UserId (assigned by the registration feature) and the name of the device on which the game was played. The function then stores this data into the Cosmos DB database. The other two functions respond to requests for data from the Cosmos DB database. One receives the player's UserId and returns their top five scores sent from all of the devices on which they play. The other simply returns the top five scores stored in the database regardless of the player.

So now the task is to integrate the Azure Functions into the game. First, at the time the game score is stored into the local database, the app should also send the score and other relevant data to the StoreScore function. Second, at the time the app reads the score history from the local database, it should also send requests to the functions that return the scores and display the results of those requests, as shown in **Figure 1**.

Where We Left Off

A little refresher will help reacquaint you with the UWP app and the Azure Functions I'll be connecting.

In the CookieBinge game, when a user completes a binge, they have two buttons to choose from to tell the app that they're done. One is the "Worth it" button to indicate that they're finished and are happy about all the cookies they scarfed down. The other is the "Not worth it" button. In response to these click events, the logic leads to the BingeServices.RecordBinge method, which uses EF Core 2 to store the data to the local database.

Code download available at msdn.com/magazine/0318magcode.

Communicating with the Web from the UWP

The UWP framework uses a special set of APIs for making Web requests and receiving responses. In fact, there are two namespaces to choose from and I recommend reading the MSDN blog post: "Demystifying HttpClient APIs in the Universal Windows Platform" at bit.ly/2rxZu3f. I'll be working with the APIs from the Windows.Web.Http namespace. These APIs have a very specific requirement for how data is sent along with any requests, and that means a little extra effort. For cases where I need to send some JSON along with my request, I've leveraged a helper class, HttpJsonContent, that will combine the JSON content together with the header content and perform some additional logic.

HttpJsonContent requires that I send my JSON in the format of a Windows.Data.Json.JsonValue. So you'll see where I've taken those steps. Alternatively, I could explicitly set the header content on the HttpRequest and then post as a String using the StringContent method, as demonstrated at bit.ly/2BBjFNE.

I encapsulated all of the logic for interacting with the Azure Functions into a class called CloudService.cs.

Once I figured out the pattern for using the UWP request methods and how to create JsonValue objects, I created a method called CallCookieBingeFunctionAsync that encapsulates most of that logic. The method takes two parameters: the name of the Azure Function to call and a JsonValue object. I also created an overload of this method that doesn't require the JsonValue object parameter.

Here's the signature for that method:

```
private async Task<T> CallCookieBingeFunctionAsync<T>(string apiMethod,
  JsonValue jsonValue)
```

As there are three different Azure Functions I need to call, let's start with the simplest—GetTop5GlobalUserScores. This function doesn't take any parameters or other content and returns results as JSON.

The GetTopGlobalScores method in the CloudService class calls my new CallCookieBingeFunctionAsync method, passing in the name of the function, and then returns the results contained in the function's response.

```
public async Task<List<ScoreViewModel>> GetTopGlobalScores()
{
  var results= await CallCookieBingeFunctionAsync<List<ScoreViewModel>>
    ("GetTop5GlobalUserScores");
  return results;
}
```

Notice that I'm not passing a second parameter to the method. That means the overload I created that doesn't require a JsonValue will be called:

```
private async Task<T> CallCookieBingeFunctionAsync<T>(string apiMethod)
{
  return await CallCookieBingeFunctionAsync<T>(apiMethod, null);
}
```

This in turn calls the other version of the method and simply passes a null where the JsonValue is expected. Here's the full listing for the CallCookieBingeFunctionAsync method (which definitely needs an explanation):

```
private async Task<T> CallCookieBingeFunctionAsync<T>(string apiMethod,
JsonValue jsonValue)
{
  var httpClient = new HttpClient();
  var uri = new Uri("https://cookiebinge.azurewebsites.net/api/" + apiMethod);
  var httpContent = jsonValue != null ? new HttpJsonContent(jsonValue): null;
  var cts = new CancellationTokenSource();
  HttpResponseMessage response = await httpClient.PostAsync(uri,
    httpContent).AsTask(cts.Token);
  string body = await response.Content.ReadAsStringAsync();
  T deserializedBody = JsonConvert.DeserializeObject<T>(body);
  return deserializedBody;
}
```

In the first step, the method creates an instance of Windows.Web.Http.HttpClient. Then the method constructs the information needed to make the request from the HttpClient, beginning with the Web address of the function to be called. All of my functions will begin with https://cookiebinge.azurewebsites.net/api/, so I've hardcoded that value into the method and then append the function name passed into the method.

Next, I have to define the headers and any content passed along to the function. As I explained earlier, I've chosen to use the helper class, HttpJsonContent, for this step. I copied this class from the JSON section of the official Windows Universal samples (bit.ly/2ry7mBP), which provided me with a means to transform a JsonValue object into an object that implements IHttpContent. (You can see the full class that I copied in the download.) If no JsonValue is passed into the method, as is the case for the GetTop5GlobalUserScores function, the httpContent variable will be null.

The next step in the method defines a CancellationTokenSource in the variable cts. While I won't be handling a cancellation in my code, I wanted to be sure you were aware of this pattern so I've included the token nevertheless.

With all of my pieces now constructed—the URI, the httpContent and the CancellationTokenSource, I can finally make the call to my Azure Function using the HttpClient.PostAsync method. The response comes back as JSON. My code reads that and uses JSON.Net's JsonConvert method to deserialize the response into whatever object was specified by the calling method.

With all of my pieces constructed now—the URI, the httpContent and the CancellationTokenSource, I can finally make the call to my Azure function using the HttpClient.PostAsync method.

If you look back at the code for GetTopGlobalScores, you'll see I specified that the results should be a List<ScoreViewModel>. ScoreViewModel is a type I created to match the schema of the score data returned by two of the Azure Functions. The class also has some additional properties that format the data based on how I want to display it in the UWP app. Given that the ScoreViewModel class is a long listing, I'll let you inspect its code in the download sample.

Calling an Azure Function That Takes a Parameter

There are still two more Azure Functions to explore. Let's look now at the other one that returns score data. This function needs the UserId passed in, whereas the previous function took no input. But in this case, I'm still not required to build up the HttpJsonContent because, if you recall the function as it was described in last month's article, it expects the UserId value to be passed in as part of the URI. The simple example from last month's article just uses the string 54321 as the UserId in this URI: https://cookiebinge.azurewebsites.net/api/GetUserScores/54321. With the addition of the identity management feature to the app, the UserId will now be a GUID.

I won't go deeply into the code for how a user's identity is managed, but here's a quick look. You will be able to see all of this code in the download. I created a new pair of Azure Functions for user management. When a user chooses to register to the cloud for score tracking, one of these functions creates a new GUID for the UserId, stores it in a separate collection in the CookieBinge Cosmos DB database and returns the GUID to the UWP app. The UWP app then uses EF Core 2 to store that UserId into a new table in the local database. Part of that GUID is displayed to users on their account page, as shown in **Figure 1**. When a user plays the CookieBinge game on another device, they can acquire the full GUID by sending the partial GUID that's available on any other device they've already registered to another Azure Function. That function returns the full GUID and the app will then store that UserId on the current device. In this way, the user can post scores from any of their devices to the cloud, always using the same UserId. Additionally, the application can use that same UserId to retrieve the scores from all of their devices from the cloud. An AccountService.cs class has functionality for the local interactions related to the UserId, including storing and retrieving the UserId from the local database. I came up with this pattern on my own and patted myself on the back, feeling so clever, even though I could probably have leveraged an existing framework.

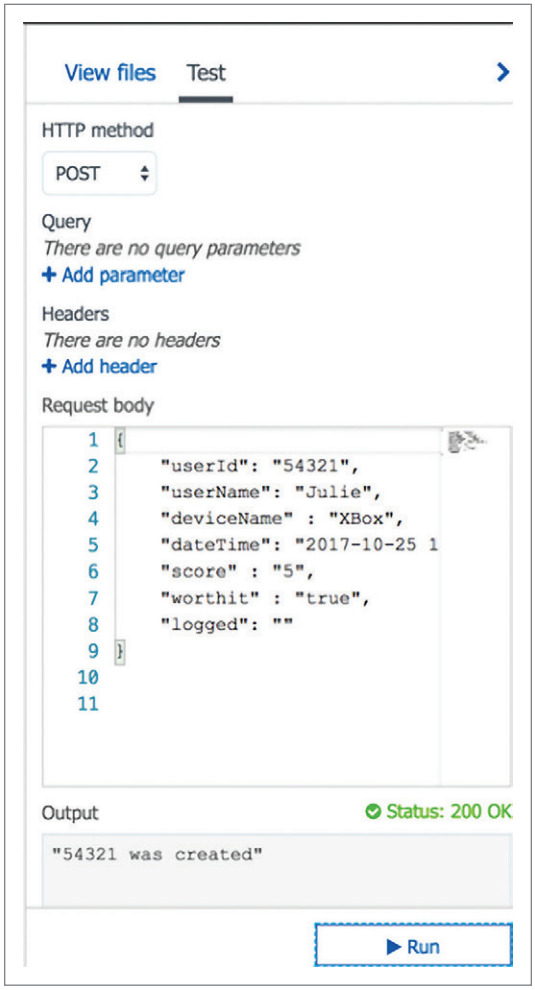


Figure 2 **The Azure Portal View of the StoreScores Function Being Tested with a JSON Request Body**

GetUserTopScores is the method in CloudServices that calls out to the GetUserScores function. Like the previous method, it calls the CallCookieBingeFunctionAsync method, again expecting the returned type to be a list of ScoreViewModel objects. I'm again passing in just a single parameter, which is not only the name of the function, but the full string that should get appended to the base URL. I'm using string interpolation to combine the function name with the results of the AccountService.AccountId property:

```
public async Task<List<ScoreViewModel>>
  GetUserTopScores()
{
  var results = await CallCookieBinge-
    FunctionAsync<List<ScoreViewModel>>
    ($"GetUserScores\\{AccountService.
      AccountId}");
  return results;
}
```

> An AccountService.cs class has functionality for the local interactions related to the user ID, including storing and retrieving the user ID from the local database.

Calling an Azure Function That Expects JSON Content in the Request

The final Azure Function, StoreScores, gives me the opportunity to show you how to append JSON to an HttpRequest. StoreScores takes a JSON object and stores its data into the Cosmos DB database. **Figure 2** serves as a reminder of how I tested the function in the Azure Portal by sending in a JSON object that follows the expected schema.

To match that schema in the UWP app I created a data transfer object (DTO) struct named StoreScoreDto, which helps me create the JSON body for the request. Here's the CloudService.SendBingeToCloudAsync method, which takes the Binge data resulting from game play and sends it to the Azure Function with the aid of the same CallCookieBingeFunctionAsync method I used to call the other two functions:

```
public async void SendBingeToCloudAsync(int count, bool worthIt,
  DateTime timeOccurred)
{
  var storeScore = new StoreScoreDto(AccountService.AccountId,
                                     "Julie", AccountService.DeviceName,
                                     timeOccurred, count, worthIt);
  var jsonScore = JsonConvert.SerializeObject(storeScore);
  var jsonValueScore = JsonValue.Parse(jsonScore);
  var results = await CallCookieBingeFunctionAsync<string>("StoreScores",
    jsonValueScore);
}
```

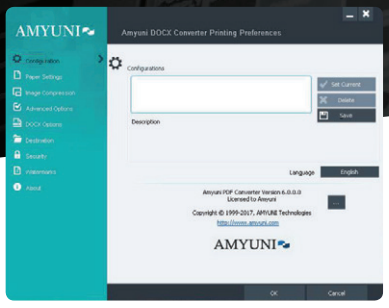
SendBingeToCloudAsync begins by taking in the relevant data about the Binge to be stored—the count of cookies consumed, whether or not the binge was worth it and when it occurred. I then create a StoreScoreDto object from that data and use JsonConvert again, this time to serialize the StoreScoreDto into a JSON object. The next step is to create a JsonValue, as I explained earlier, a special type in the Windows.Json.Data namespace. I do that using the JsonValue.Parse method, passing in the JSON object represented by jsonScore. The resulting JsonValue is the format required to send the JSON object along with the HTTP request. Now that I've got

DOCXCONVERTER
For Windows

Free Demo at DOCXConverter.com

Amyuni DOCX Converter for Windows

Convert any document, including PDF documents, into DOCX format.
Enable editing of documents using Microsoft Word or other Office products.



A standalone desktop version, a server product for automated processing or an SDK for integration into third party applications.

Create

Create naturally editable DOCX documents with paragraph formatting and reflow of text

Convert

Convert images and graphics of multiple formats into DOCX shapes

OCR

Use OCR technology to convert non-editable text into real text

Extract

Extract headers and footers from source document and save them as DOCX headers and footers

Open

Open PDF documents with the integrated PDF viewer and quickly resave them to DOCX format

Configure

Configure the way the fonts are embedded into the DOCX file for optimal formatting

A virtual printer driver available for Windows 7 to Windows 10 and Windows Server 2008 to 2016

Powered by Amyuni Technologies:

Developers of the Amyuni PDF Converter and Amyuni PDF Creator products integrated into hundreds of applications and installed on millions of desktops and servers worldwide.

www.docxconverter.com

the properly formatted JsonValue, I can send it to the CallCookeBingeFunctionAsync method along with the name of the function, StoreScores. Notice that the type I expect to be returned is a string, which will be a notification from the StoreScores Azure Function of the function's success or failure.

Wiring up the UI to the CloudService

With the CloudService methods in place, I can finally make sure the UI interacts with them. Recall that when a Binge is saved, the code in MainPage.xaml.cs calls a method in BingeService that stores that data to the local database. That same method, shown in **Figure 3**, now also sends the binge data to the CloudService to store it in the cloud via the StoreScores Azure Function.

> What began as an exercise to try out the latest version of EF Core 2 on Windows-based mobile devices turned into quite an adventure for me.

Both of the other methods that interact with the Azure Functions return lists of ScoreViewModel objects.

To display the scores stored in the cloud, as shown in **Figure 1**, I added a method to MainWindow.xaml.cs that calls the CloudService methods to retrieve the scores and then binds them to the relevant ListViews on the page. I named this method ReloadScores because it's also called by the refresh button on the same page:

```
private async Task ReloadScores()
{
  using (var cloudService = new BingeCloudService())
  {
    YourScoresList.ItemsSource = await cloudService.GetUserTopScores();
    GlobalScoresList.ItemsSource =
      await cloudService.GetTopGlobalScores();
  }
}
```

The UI then displays the score data based on the templates defined for each list on the page. For example, **Figure 4** shows XAML for displaying the GlobalScores in the UI.

Figure 3 **The Pre-Existing RecordBinge Method Now Sends the Binge to the Cloud**

```
public static  void RecordBinge(int count, bool worthIt)
{
  var binge = new CookieBinge{HowMany = count, WorthIt = worthIt,
                              TimeOccurred = DateTime.Now};
  using (var context = new BingeContext(options))
  {
    context.Binges.Add(binge);
    context.SaveChanges();
  }
  using (var cloudService = new BingeCloudService())
  {
    cloudService.SendBingeToCloudAsync(count, worthIt, binge.TimeOccurred);
  }
}
```

Figure 4 **XAML Data Binding the Score Data Returned from an Azure Function**

```
<ListView  x:Name="GlobalScoresList"    >
  <ListView.ItemTemplate>
    <DataTemplate >
      <StackPanel Orientation="Horizontal">
        <TextBlock FontSize="24" Text="{Binding score}"
                   VerticalAlignment="Center"/>
        <TextBlock FontSize="16" Text="{Binding displayGlobalScore}"
                   VerticalAlignment="Center" />
      </StackPanel>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Wrapping Up This Four-Part Series

What began as an exercise to try out the latest version of EF Core 2 on Windows-based mobile devices turned into quite an adventure for me, and I hope it was a fun, interesting and educational journey for you, as well. Working with the new .NET Standard 2.0-based UWP, especially in its early pre-release days, was certainly challenging for this back-end developer. But I loved the idea of being able to store data both locally and in the cloud and gaining new skills along the way.

The second and third column in the series were my very first experiences working with Azure Functions and I'm so happy I had an excuse to do so because I'm now a huge fan of this technology and have done much more with it since those first steps. I certainly hope you've been equally inspired!

As you saw in this article, interacting with those functions from the UWP app isn't as straightforward as my earlier experience making Web calls from other platforms. I personally got great satisfaction figuring out the workflow.

If you check out the download, you'll see the other addition I made to the application—all of the logic for registering to cloud storage, saving the Azure-generated UserId, as well as a name for the device, and registering additional devices and then accessing the UserId and device name for use in the StoreScores and GetUserScores methods. I've downloaded the entire Azure Function App into a .NET project so you can see and interact with all of the functions that support the app. I spent a surprising amount of time puzzling through the identity workflow and became somewhat obsessed with the entertainment of working it out. Perhaps I'll write about that someday, as well. ■

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

Thanks to the following technical expert for reviewing this article: *Ginny Caughey (Carolina Software Inc.)*

File Format APIs

Open, Create, Convert, Print and Save files from your applications!

Try risk free - 30 day trial

Aspose.Total

Enable your applications to manipulate Word, Excel, PDF, PowerPoint, Outlook and more than 100 other file formats for all major platforms.

Aspose.Words

Create, edit, convert or print Word documents (DOC, DOCX, RTF etc.) in your .NET, Java and Android applications.

Aspose.Cells

Develop high performance .NET, Java and Android applications to Create, Edit or Convert Excel worksheets (XLS, XLSX, ODS etc).

Aspose.Pdf

Manipulate PDF file formats (PDF, PDF/A, XPS etc.) using our native APIs for .NET, Java and Android platforms.

Aspose.Slides

Create, edit or convert PowerPoint presentations (PPT, PPTX, ODP etc.) in your .NET, Java and Android applications.

Aspose.Email

Create, Edit or Convert Outlook Email file formats (MSG, PST, EML etc.) and popular network protocols.

Aspose.BarCode

Generate or recognize barcodes (Code128, PDF417, Postnet etc.) using our native APIs for .NET and Java.

Aspose.Imaging | Aspose.Tasks | Aspose.OCR | Aspose.Diagram | Aspose.Note | Aspose.HTML

ASPOSE File Format APIs

Download a Free Trial at **https://downloads.aspose.com**

Americas: +1 903 306 1676 EMEA: +44 141 628 8900 Oceania: +61 2 8006 6987
sales@asposeptyltd.com

TED NEWARD

How To Be MEAN: Validating Angular

Welcome back again, MEANers.

In the previous column, I started looking at Angular's support for forms and input. Two-way binding took top billing, but any sense of how to validate input—to make sure that Speakers have both a first and a last name, for example, and not just empty strings—was left behind. In this month's column, we need to rectify that, because data input without validation is basically just asking users to pour garbage into your system, leaving it to you to sort out.

And that, as most of us know, would be bad. Like, trademarked levels of Really, Really Bad™.

SpeakerUI, Redux

In the last column, I ditched the SpeakerEdit component in favor of SpeakerUI, which wraps around a Speaker model instance and knows—based on what's passed into it—whether it's in a read-only state for viewing the Speaker, or an editable state. It uses two sections (one of which is hidden depending on the state the component is in) to keep the UI distinct (see **Figure 1**). The read-only section requires no validation, of course, because there's no user input; it's the editable section that concerns us here.

The first thing to note is that between last month's column and this, I moved the logic for determining whether or not the edit mode can be canceled (notice how the Cancel button's disabled property is bound) to a method on the component itself. This may be a bit of overkill in this particular case, but it does demonstrate an important aspect of Angular—that you do not have to do all of the UI logic directly inside the template itself. Should the cancellation logic get complicated, having that in the template is probably a bad idea, but we need to have the form object (the speakerForm object defined last month) available to use in the component code.

That requires the use of a new module, one that isn't already present in the component: NgForm.

NgForm

NgForm is a class defined in Angular specifically for working with forms. It's contained in a separate module from the rest of the Angular core, so it requires a standalone import to retrieve:

```
import { NgForm } from '@angular/forms';
```

When working with forms at runtime, Angular constructs a collection of objects that represents the various controls and the form itself, and uses it to do validation and other processing. This object collection is often hidden behind the scenes for convenience, but is always available to Angular developers for use.

Once passed into the cancellable method, you can use the form object to examine the state of the form via a number of properties. NgForm defines dirty, invalid, pristine, touched, untouched and valid properties to represent an entire spectrum of different user-interaction states with the form. For demonstration purposes, I'll add a few more diagnostic lines to the editable section of the form:

```
<br>Pristine: {{speakerForm.form.pristine}}
Dirty: {{speakerForm.form.dirty}}
Touched: {{speakerForm.form.touched}}
Untouched: {{speakerForm.form.untouched}}
Invalid: {{speakerForm.form.invalid}}
Valid: {{speakerForm.form.valid}}
```

These will simply display the state of each of these as the user interacts with the form, and help explain what each represents. For example, "untouched" means—quite literally—the user hasn't touched the form in any way. Simply clicking (or touching, on a mobile device) the edit field so that the cursor appears there is enough to render the form as being "touched." However, if no typing has taken place, even if the form is "touched," it's still "pristine." And so on.

Validity, as might be expected, suggests that the user has violated some kind of data-entry constraint that the developer has mandated. Angular looks to build off of standard HTML5 validity constraints, so, for example, if you decide that speakers must have both a first and a last name, you can simply use the "required" attribute on the edit fields:

```
FirstName: <input name="firstName" type="text"
  [(ngModel)]="model.firstName" required><br>
LastName:  <input name="lastName" type="text"
  [(ngModel)]="model.lastName" required><br>
```

Given this, if the user edits an existing Speaker and clears either the firstName or lastName edit field completely, the invalid state flips to true and the valid state to false, because Angular recognizes that the required flag is present. That said, though, Angular doesn't do

Figure 1 **The SpeakerUI Component**

```
<form #speakerForm="ngForm">
  <div [hidden]="readonly">
    FirstName: <input name="firstName" type="text"
      [(ngModel)]="model.firstName"><br>
    LastName:  <input name="lastName" type="text"
      [(ngModel)]="model.lastName"><br>
      Subjects: {{model.subjects}}<br>
    <button (click)="save()"
      [disabled]="!speakerForm.form.dirty">Save</button>
    <button (click)="cancel()"
      [disabled]="!cancellable(speakerForm)">Cancel</button>
    <br><span>{{diagnostic}}</span>
  </div>
  ...
</form>
```

Help & Manual Professional | from $586.04

ec software

Help and documentation for .NET and mobile applications.

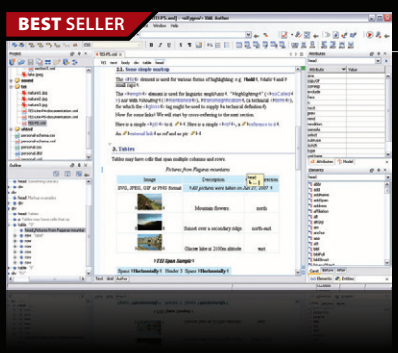
- Powerful features in an easy, accessible and intuitive user interface
- As easy to use as a word processor, but with all the power of a true WYSIWYG XML editor
- Single source, multi-channel publishing with conditional and customized output features
- Output to responsive HTML, CHM, PDF, MS Word, ePUB,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

DevExpress DXperience 17.2 | from $1,439.99

DevExpress

The complete range of DevExpress .NET controls and libraries for all major Microsoft platforms.

- WinForms - New TreeMap control, Chart series types and Unbound Data Source
- WPF - New Wizard control and Data Grid scrollbar annotations
- ASP.NET - New Vertical Grid control, additional Themes, Rich Editor Spell Checking and more
- Windows 10 Apps - New Hamburger Sub Menus, Splash Screen and Context Toolbar controls
- CodeRush - New debug visualizer expression map and code analysis diagnostics

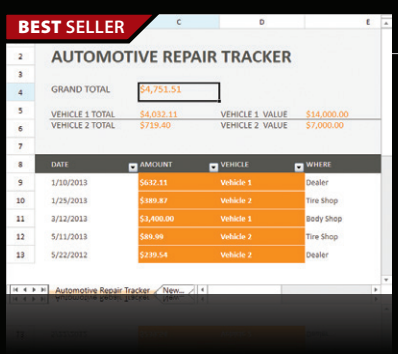


Oxygen XML Editor Professional | from $660.48

syncro soft

The complete XML editing solution, both for XML developers and content authors

- Produce output in PDF, ePUB, HTML, and many other formats from a single source
- Ready-to-use support for DITA, DocBook, XHTML, and TEI frameworks
- Extend the built-in XML publishing frameworks, or even create your own frameworks
- Interact with the majority of XML databases, content management systems, and WebDAV
- Make sure your XML documents are "well-formed" and valid, using as-you-type validation



Spread.NET | from $1,476.52

GrapeCity

Deliver multi-functional spreadsheets in less time with Visual Studio.

- Designed from the ground up for maximum performance and speed
- Gives you the features you expect for seamless Excel compatibility with NO Excel dependency
- Powerful calculation engine handles huge numbers of rows and columns with ease
- Use the extensive API to control every aspect of workbooks, worksheets, ranges, and cells
- Benefit from the integrated calculation engine with more than 462 Excel functions available

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

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

Europe (UK)
ComponentSource
The White Building
33 King's Road
Reading, Berkshire
RG1 3AR
United Kingdom

Europe (Ireland)
ComponentSource
Unit 1E & 1F
Core B, Block 71
The Plaza, Park West
Dublin 12
Ireland

Asia Pacific
ComponentSource
7F Kojimachi Ichihara Bldg
1-1-8 Hirakawa-cho
Chiyoda-ku
Tokyo, 102-0093
Japan

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

GSA Schedule Contract GS-35F-0188R

anything else—out of the box, Angular doesn't provide any built-in UI to indicate that the form is invalid. It's up to the developer to signal to the user in some way that the field requires attention. This can be done in a variety of ways, all dependent on what the developer is using for UI support. For example, it's common when using the Bootstrap CSS framework to flag the form field as requiring attention by coloring it (or some portion of it) red. Alternatively, it's not uncommon to have a hidden text span below or after the field that will display when the constraints are violated in some way, and tie the span's hidden attribute to the status of the form.

But that raises a subtle point—you would prefer to know which control within the form is invalid, so that you can tie the feedback directly to that control. Fortunately, the NgForm has a controls property, which is an array of NgControl objects, and each control defined within the form (such as firstName and lastName) will have an NgControl instance to represent it. Thus, you can reference those control objects directly within the hidden attribute's template expression:

```
FirstName: <input name="firstName" type="text"
  [(ngModel)]="model.firstName" required>
<span [hidden]="speakerForm.controls.firstName.valid">
  Speakers must have a first name</span><br>
LastName:  <input name="lastName" type="text"
  [(ngModel)]="model.lastName" required>
<span [hidden]="speakerForm.controls.firstName.valid">
  Speakers must have a first name</span><br>
```

Candor compels me to admit that this code has a subtle issue—when run, it will yield a few errors at runtime. That's because during the earliest stages of the component the NgForm hasn't constructed the collection of NgControl objects, and so the expression, speakerForm.controls.firstName, will be undefined.

The easy way to avoid this problem is to define a local template variable for the control, rather than go through the form's controls array, and use *ngIf directives to test to see if the form is touched or dirty, and if so, whether it's valid:

```
FirstName: <input name="firstName" type="text"
  [(ngModel)]="model.firstName" #firstName="ngModel"
  required>
<div *ngIf="firstName.invalid &&
            (firstName.dirty || firstName.touched)">
  <div *ngIf="firstName.errors.required">
    A first name is required.
  </div>
</div>
```

Essentially, this eliminates the need to work through the speakerForm, but it's useful to know that the speakerForm object is accessible to us at runtime, albeit with some caveats.

Custom Validation

In those situations where the HTML5 standard doesn't define a validation you want or need, Angular permits you to write a custom validator that can be invoked to test the field in question. For example, many years ago, let's assume I had a bad experience with a speaker named Josh. I don't like Josh. Never did. I don't care to let the guy be a part of our database, so I want a custom form validator that disallows particular input. (Obviously, this is a pretty pedantic example, but the concepts here hold for just about any kind of validator that could be imagined.)

Like the other validation, Angular wants to try and "tap in" to the HTML syntax as much as possible, which means that even custom validators will appear like HTML5 validators, so the forbiddenName validator should appear like any other HTML validation rule:

```
FirstName: <input name="firstName" type="text"
  [(ngModel)]="model.firstName" #firstName="ngModel"
  required forbiddenName="josh">
```

Within the template, you can simply add the necessary *ngIf directive to test whether the form contains the forbidden name specified, and if so, display a specific error message:

```
<div *ngIf="firstName.invalid &&
            (firstName.dirty || firstName.touched)">
  <div *ngIf="firstName.errors.required">
    A first name is required.
  </div>
  <div *ngIf="firstName.errors.forbiddenName">
    NO JOSH!
  </div>
</div>
```

In those situations where the HTML5 standard doesn't define a validation you want or need, Angular permits you to write a custom validator that can be invoked to test the field in question.

There. That should keep out any unwanted speakers (looking at you, Josh).

Custom Validator Directives

In order to make this work, Angular requires us to write the validator as an Angular directive, which is a means for hooking up some Angular code to an HTML-looking syntactic element, such as the forbiddenName directive in the input field, or the required or even *ngIf. Directives are quite powerful, and quite beyond the room I have here to explore fully. But I can at least explain how validators work, so let's start by creating a new directive using "ng generate directive ForbiddenValidator" at the command line, and have it scaffold out forbidden-validator.directive.ts:

```
import { Directive } from '@angular/core';

@Directive({
  selector: '[appForbiddenValidator]'
})
export class ForbiddenValidatorDirective {

  constructor() { }

}
```

The selector in the @Directive is the syntax that you want to use in the HTML templates and, frankly, appForbiddenValidator doesn't really get the heart racing. It should be something a little more clear in its usage, like forbiddenName. Additionally, the Directive needs to tap into the existing collection of validators—without going into too much detail, the providers parameter to the @Directive contains necessary boilerplate to make the ForbiddenValidatorDirective available to the larger collection of validators:

```
@Directive({
  selector: '[forbiddenName]',
  providers: [{provide: NG_VALIDATORS,
               useExisting: ForbiddenValidatorDirective,
               multi: true}]
})
```

Next, the directive needs to implement the Validator interface, which provides a single method, validate, which—as could be guessed—is invoked when validation needs to be done. However, the result from the validate function isn't some kind of pass/fail result of the validation, but the function to use to perform the validation itself:

```
export class ForbiddenValidatorDirective implements Validator {
  @Input() forbiddenName: string;

  validate(control: AbstractControl): {[key: string]: any} {
    if (this.forbiddenName) {
      const nameRe = new RegExp(this.forbiddenName, 'i');
      const forbidden = nameRe.test(control.value);
      return forbidden ? {'forbiddenName': {value: control.value}} : null;
    } else {
      return null;
    }
  }
}
```

Fundamentally, Angular walks through a list of validators, and if all of them return null, then everything is kosher and all the input is considered valid. If any of them return anything other than that, it's considered to be part of the set of validation errors, and added to the errors collection that the template referenced in the *ngIf statements earlier.

If the validator has a forbiddenName value, which is what's passed in from the directive's usage, then there's validation to be done—the component's input is used to construct a RegExp instance (using "i" for a case-insensitive match), and then the RegExp test method is used to check to see if the control's value matches the name-which-shall-not-be-accepted. If it does, then a set is constructed with the key forbiddenName (which is what the template was using earlier to determine whether to show the error message, remember), and the value being the control's input. Otherwise, the directive hands back null, and the rest of the form's validators are fired. If all of them hand back null, everything's legit.

Wrapping Up

Angular's validation support is, as you can tell, pretty extensive and full-featured. It builds off of the standard HTML validation support that's found in every HTML5 browser, but provides a degree of runtime control support that's extensible and powerful when needed. Most applications will find the built-in validators to be sufficient for most purposes, but having the ability to create custom validators means Angular can be as complex as necessary when working with user input. (Which is good, because users constantly find new ways to attempt to enter garbage into the system. I blame Josh for that.) There are still some more surprises hiding within the "@angular/forms" module before we're done here, so stay tuned.

Happy coding! ■

Ted Neward *is a Seattle-based polytechnology consultant, speaker, and mentor, currently working as the director of Developer Relations at Smartsheet.com. He has written a ton of articles, authored and co-authored a dozen books, and speaks all over the world. Reach him at ted@tedneward.com or read his blog at blogs.tedneward.com.*

Thanks to the following Microsoft technical expert: *Garvice Eakins*

FRANK LA VIGNE

Explore Deep Learning Toolkits with Jupyter Notebooks

In my last column, I explored the Jupyter Notebook, an open source, browser-based software solution that allows users to create and share documents that contain live code, visualizations and text. While not ideal for creating applications, Jupyter Notebooks are a great way to explore and experiment with data. Think of them as a kind of interactive "scratch pad" for data science. Jupyter Notebooks provide a common format for data scientists to share code, insights and documentation. Many of the popular machine learning libraries, such as CNTK and TensorFlow, provide Jupyter Notebooks as documentation to facilitate learning how to use them. Fortunately, there's an easy way to get access to numerous sample notebooks for all of the most popular libraries—all in one place and without installing software.

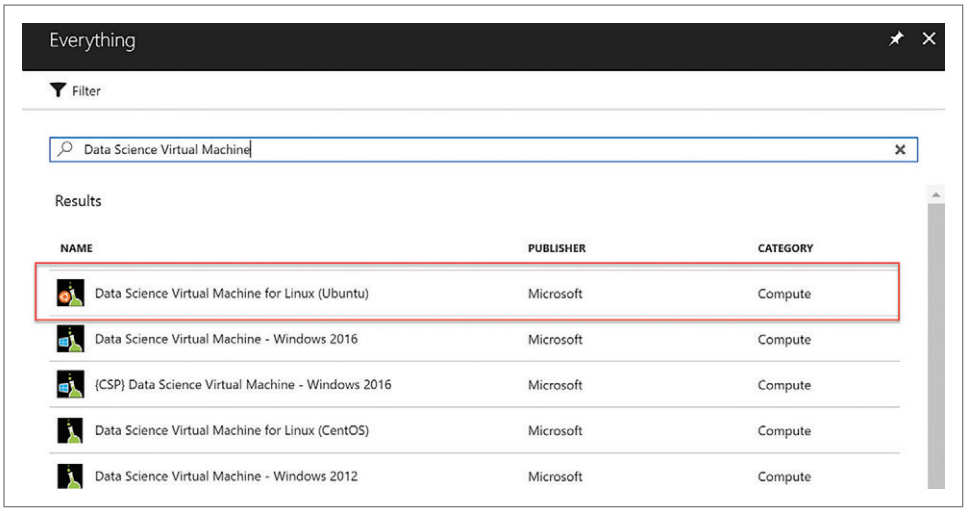


Figure 1 **Searching the Marketplace for the Data Science Virtual Machine Image**

The Data Science Virtual Machine

Data science and machine learning require new tools, many of which may be unfamiliar to .NET developers. Fortunately, Microsoft Azure provides a virtual machine (VM) image—called the Data Science Virtual Machine (DSVM)—that's pre-loaded with a number of tools and utilities for data science and machine learning work. The DSVM image makes it easy to get started with deep learning technologies, and includes the Microsoft Cognitive Toolkit (CNTK), TensorFlow, Keras, PyTorch and more, already built, installed and configured. Each toolkit is primed for use.

Best of all, and most relevant to this article, numerous sample Jupyter Notebooks are also included. In short, the DSVM provides an easy way to get started without having to install anything on your local machine. The DSVM image is available in Windows, Ubuntu and CentOS. In this article, I will focus on the Ubuntu version, as I find it to be the most thorough. You can find a complete list of tools installed on the DSVM image at bit.ly/2Dc5uA4.

Creating a DSVM is easy. To get started, log into the Azure Portal and click New, then type "Data Science Virtual Machine" in the search textbox and hit enter. You should see search results similar to **Figure 1**. Click on the item Data Science Virtual Machine for Linux (Ubuntu).

To create an instance of the DSVM, follow the instructions of the Create Virtual Machine blade. Provide a name for the VM, a username and choose "Password" for authentication type. Create a new resource group for the DSVM or use an existing one. Remember the Resource Group name in addition to the username and password. Leave the other options at their default values.

Click OK to move to the next step: picking a size for the VM. For now, size doesn't matter, so just go with a size that fits your budget. If you don't see an affordable option, click "View all" to see all VM configurations. Click Select to choose a size and configuration. On the third step, leave everything as defaults and click OK. The final step is the summary screen. It's worth noting that some Azure plans do not include the DSVM and your account will be charged separately. The summary screen will inform you if that's the case. Click Create to instantiate the VM to continue.

Connecting to the DSVM GUI

The best way to get the most out of the DSVM is to connect to it graphically. While the blade for the VM instance on the portal provides information on how to connect via an SSH service, it doesn't offer any guidance on connecting to it via a graphical shell. Fortunately, there's extensive documentation on how to do this using Remote Desktop for Windows (bit.ly/2Davn3j). The DSVM Ubuntu

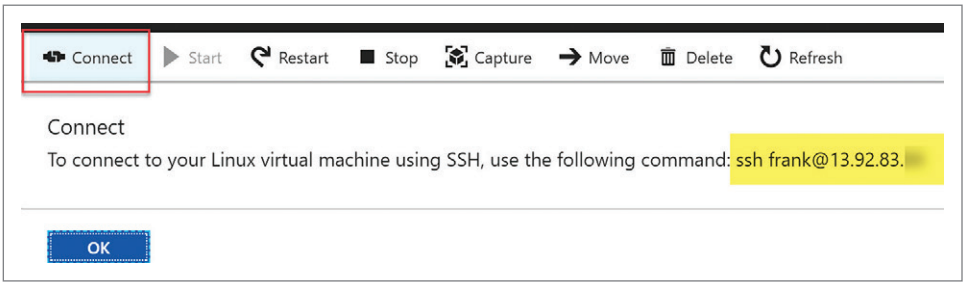


Figure 2 **Getting the SSH Connection Information for the DSVM**

image has the xcfe4 desktop environment already installed, and you created a local user account when provisioning the VM. All that's left to do now is configure the remote desktop service to listen for incoming Remote Desktop Protocol (RDP) connections. To do this, you'll need to enable xrdp, an open source RDP server that works with xfce.

All this configuration activity on the Ubuntu CLI may seem superfluous for a column about artificial intelligence, but bear in mind that many tools in this space assume a basic familiarity with Linux and the Bash command-line shell.

To install xrdp on the Ubuntu DSVM, you must connect to it via an SSH service, which on Windows often requires a terminal program like PuTTY (putty.org). If your PC is running Windows 10 Build 1709 or later, you can opt instead to install Ubuntu on Windows via the Store (bit.ly/2Dm8fSR). To get the connection information, look for the Connect button on the blade for the DSVM instance. Click on it to bring up the dialog shown in **Figure 2** to get the connection information.

In the terminal window, enter the connection information and, when prompted, enter "yes" to trust the connection, followed by the password for the username. To install and enable the xrdp service to work with xfce4, enter the following three lines:

```
sudo apt-get install xrdp
echo xfce4-session >~/.xsession
sudo service xrdp restart
```

If prompted, respond with "Y" to enable use of extra disk space. Once xfce is installed on the DSVM, you need to create a rule in the Network Security Group to allow RDP traffic through to the VM. This can be done via the Azure Portal (bit.ly/2mDQ1lt), or by issuing the follow statement into the command line through the Azure CLI:

```
az vm open-port --resource-group ResourceGroup --name DSVM-Name --port 3389
```

Be sure to replace "ResourceGroup" and "DSVM-Name" with the name of the Resource Group and VM name from earlier. Once that process completes, open the Remote Desktop Connect application, enter the IP address of the DSVM instance and click Connect. If prompted, choose to trust the machine and enter the credentials for the account created along with the DSVM.

All this configuration activity on the Ubuntu CLI may seem superfluous for a column about artificial intelligence, but bear in mind that many tools in this space assume a basic familiarity with Linux and the Bash command-line shell. The sooner you feel comfortable with Bash and Linux, the sooner you'll be productive.

If you wish to avoid the preceding steps to install and configure xrdp, X2Go (wiki.x2go.org) is an alternative for Mac and Windows users. X2Go communicates directly to the DSVM via the xfce protocol, so there's no need to install anything on the VM or to alter the Network Security Group. Simply install X2Go on your local desktop and connect with the IP address and username for the DSVM.

Once connected to the DSVM, click on the Applications menu in the upper-left corner of the GUI. Click on Development and click on JupyterHub on the resulting submenu. JupyterHub is a multi-user hub for managing multiple instances of the single-user Jupyter Notebook server. It can be used to serve notebooks to a group of users, such as students in a class, a research group or a team of data scientists.

A terminal window now opens and the system's default browser (FireFox) opens to http://localhost:8888/tree. Click on the CNTK folder to access the Microsoft Cognitive Toolkit samples. Next, click on the CNTK_101_LogisticRegression.ipynb file to access the Logistic Regression and ML Primer notebook, which contains a tutorial for those new to machine learning and to the CNTK. This tutorial notebook is written for Python. If you receive the error, "Can't execvp Jupyter: No such file or directory," you'll have to use X2Go to continue.

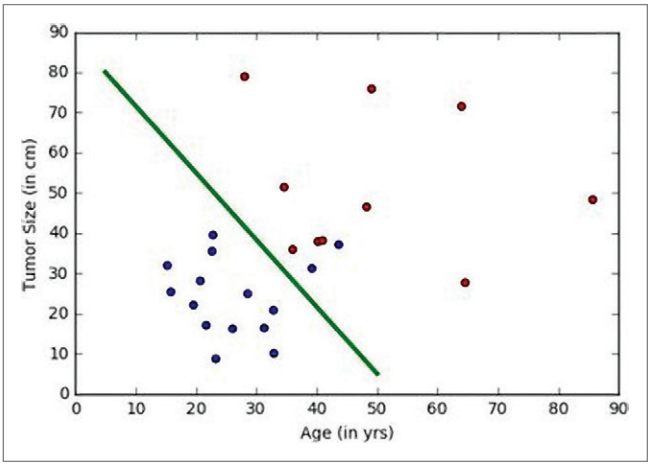


Figure 3 **Scatter Plot with Binary Classifier Indicated by Green Line**

Classifying Tumors

The problem posed in the CNTK_101_LogisticRegression notebook centers around classifying tumor growths as either malignant or

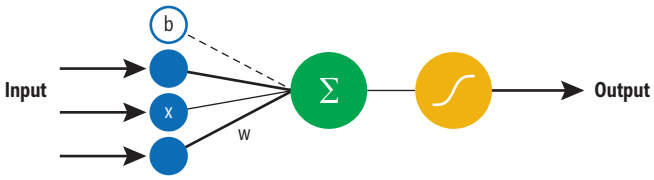


Figure 4 **Diagram of Logistic Regression Algorithm**

benign. This is a classification problem, specifically a binary classification, as there are only two output classifications. To classify the type of growth in each patient, the hospital has provided the patient's age and the size of his or her tumor. The working hypothesis is that that younger patients and patients with smaller tumors are less likely to have a malignant growth. **Figure 3** shows how each patient in the data set is represented as a dot in the plot in the notebook. Red dots indicate malignant growths and blue dots indicate benign. The goal is to create a binary classifier to separate the malignant from the benign, similar to the scatter plot in the Out[3] cell in the notebook and duplicated in **Figure 3**.

The notebook includes a caution stating that the data set used here is merely a sample for educational purposes. A production classification system for determining the status of growths would involve more data points, features, test results and input from medical personnel to make the final diagnosis.

The Five Stages of a Machine Learning Project

Machine learning projects generally break down into five stages: Reading the data, shaping the data, creating a model, learning the model's parameters and evaluating the model's performance. Reading the data entails loading the data set into a structure. For Python, this generally means a Pandas DataFrame (bit.ly/2EPC8rl). DataFrames are essentially a tabular data structure consisting of rows and columns.

The second step is shaping the data from the input format into a format that the machine learning algorithm accepts. Often, this process is referred to as "data cleansing" or "data munging." Very often this phase consumes the bulk of the time and effort in machine learning projects.

It's not until the third stage that the actual machine learning work begins. In this notebook, I'll create a model to separate the benign growths from the malignant ones. Logistic Regression is a simple linear model that takes input values (represented by the blue circles in **Figure 4**) of what I'm classifying and computes an output. Each of the input values has a different degree of weight on the output, depicted in **Figure 4** as line thickness.

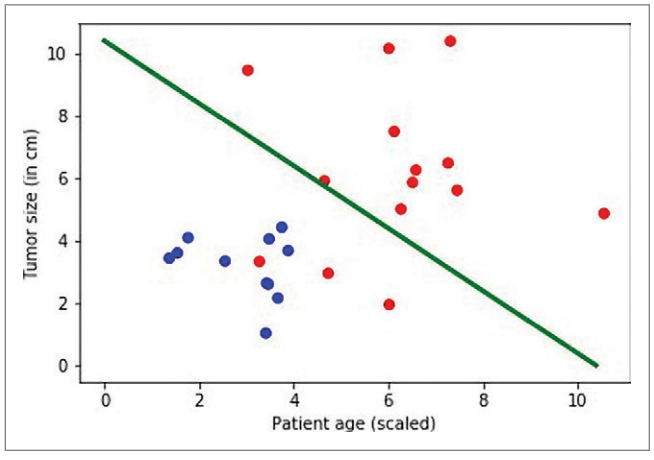


Figure 5 **Three Improperly Flagged Tumors**

The next step is to minimize the error, or loss, using an optimization technique. This notebook uses Stochastic Gradient Descent (SGD), a popular technique that usually begins by randomly initializing the model parameters. In this case, the model parameters are the weights and biases. For each row in the data set, the SGD optimizer can calculate the error between the predicted value and the corresponding true value. The algorithm will then apply gradient descent to create new model parameters after each observation. SGD is explained in further detail inside the notebook and in a YouTube video by Siraj Raval (bit.ly/2B8lHEz).

The final step is to evaluate the predictive model's performance against test data. There are essentially four outcomes for this binary classifier:

1. Correctly labeled the value as malignant.
2. Correctly labeled the value as benign.
3. Incorrectly labeled the value as malignant.
4. Incorrectly labeled the value as benign.

In this scenario, outcome No. 3 would be a false negative. The tumor is benign, but the algorithm flagged it incorrectly as malignant. This is also known as a Type II Error. Outcome No. 4 represents the reverse, a false positive. The tumor is malignant, but the algorithm marked it as benign, a Type I Error. More information about the Type I and II Errors, please refer to the Wikipedia article on erroneous outcomes of statistical tests at bit.ly/2DccUU8.

Using the plot in **Figure 5**, you can determine that the algorithm labeled three malignant tumors as benign, while not mislabeling any benign tumors as malignant.

Wrapping Up

The best way to quickly get started with exploring the various frameworks of data science, machine learning and artificial intelligence is through the DSVM image on Azure. It requires no installation or configuration, and it can be scaled up or down based on the problem to be solved. It's a great way for people interested in machine learning to start experimenting right away. Best of all, the DSVM includes numerous Jupyter Notebook tutorials on the most popular machine learning frameworks.

In this article, I set up a DSVM and took the first steps into exploring the CNTK. By creating a binary classifier with logistic regression and Stochastic Gradient Descent, I trained an algorithm to determine whether or not a tumor was benign or malignant. While this model relied on only two dimensions and may not be ready for production, you did get an understanding of how this problem would be tackled in the real world. ■

Frank La Vigne *leads the Data & Analytics practice at Wintellect and co-hosts the DataDriven podcast. He blogs regularly at FranksWorld.com and you can watch him on his YouTube channel, "Frank's World TV" (FranksWorld.TV).*

Thanks to the following technical expert for reviewing this article: *Andy Leonard*

Learn, Explore, Use

Your Destination for Data Cleansing & Enrichment APIs

Your centralized portal to discover our tools, code snippets and examples.

RAPID APPLICATION DEVELOPMENT

Convenient access to Melissa APIs to solve problems with ease and scalability.

REAL-TIME & BATCH PROCESSING

Ideal for web forms and call center applications, plus batch processing for database cleanup.

TRY OR BUY

Easy payment options to free funds for core business operations.

FLEXIBLE CLOUD APIS

Supports REST, JSON, XML and SOAP for easy integration into your application.

Turn Data into Success – Start Developing Today!

Melissa.com/developer

1-800-MELISSA

melissa

Blockchain Fundamentals

Jonathan Waldman

Back in 1999, the file-sharing network Napster made it easy to share audio files (usually containing music) on a hybrid peer-to-peer network ("hybrid" because it used a central directory server). That file-sharing network did more than just share music files: It allowed all users to retain copies of those shared files such that a single digital asset resulted in a limitless number of perfect copies across a global network. The casual ease with which technology was leveraged by anyone with a computer caught venerable Tower Records by such surprise that it was forced to close all of its 89 U.S.-based stores by 2006.

In 2008, the subprime debacle occurred, during which long-established, powerful U.S. financial institutions and insurance companies declared or teetered on the brink of bankruptcy. Those circumstances called for immediate federal government intervention in order to avoid a domestic and possibly global financial meltdown. This important event left a populace leery of centralized banks and exposed the danger of having financial ledgers closed to public scrutiny. In March 2008, the Heartland Payment Systems data breach exposed more than 130 million credit card numbers, many of which were later used to make fraudulent purchases.

These events illustrate the perils of living in a digital, connected world that depends on transaction-fee-generating middlemen and leaves people vulnerable to digital exploits, greed and crime. The academic challenge became how to create an available, disintermediated digital infrastructure on which a digital asset can be openly and reliably transferred (rather than copied and shared) from owner to owner, that has no corruptible or fallible central authority, is secure, and can be trusted.

This article discusses:

- Driving forces that led to the creation of blockchains
- How blockchains track and enforce ownership of digital assets
- Core underpinnings of blockchain technologies
- Leveraging core components to build a blockchain
- How blockchains use consensus to achieve decentralized automation

Technologies discussed:

Blockchain, Cryptographic Hash, Public Key Cryptography (PKC), Merkle Tree, Consensus Algorithm

Enter the Bitcoin Blockchain

Seemingly in response to the historical context in which it happened, on Jan. 3, 2009, a new kind of infrastructure mined 50 digital coins and recorded them on a tamper-proof public ledger that's replicated on a decentralized peer-to-peer network of Internet-connected computers. Those 50 cryptocurrency units, called bitcoins, were recorded as the genesis block—the first link on what would come to be known as the Bitcoin blockchain.

The remarkable thing about this blockchain-powered cryptocurrency is that it lacks any sort of trust authority or governance

(such as a bank or government) to validate each transaction. Furthermore, it disintermediates transactions, making it possible to transfer digital currency internationally using a global network with no middleman involvement, such as a broker or agency. Due to its reliance on modern cryptography, the data contained in the blockchain is tamper-proof and pseudonymous. And because a given blockchain is replicated on every node that comprises its peer-to-peer network, there's no single point of failure, making the technology available and reliable.

Since Bitcoin's launch, blockchain technologies continue to mature rapidly. Their implementation details vary dramatically, making the study of blockchains dynamic, vast and complex. Indeed, the term "blockchain" no longer applies only to Bitcoin in particular or to cryptocurrencies in general. Blockchains are being refined and perfected so they work faster and smarter. In fact, some blockchain technologies allow scripting to support smart contracts, which permits custom rules to be applied to transactions. In this way, blockchains have morphed into a new kind of programmable, hacker-proof storage technology, which is one reason why IT professionals, businesses, financial institutions, and others are clamoring to harness their true potential.

If you've been sitting on the blockchain sidelines, it's time to catch up. In this introductory article, it isn't possible to cover the finer technical details of any particular blockchain technology—each has its own rules, features and customizations—but the ideas I discuss here will help orient you to the core technical underpinnings on which many modern blockchain technologies are based.

How Blockchains Work

The Bitcoin blockchain is the world's first practical example of blockchain technology. And because of that distinction, "blockchain" is often misunderstood as being synonymous with Bitcoin. However, modern blockchain technology offerings track digital assets other than a digital currency, and those blockchains work quite differently from the way Bitcoin's blockchain works. Additionally, the Bitcoin blockchain popularized the notion that a blockchain is a data structure that virtualizes a bank ledger by tracking credits and debits while offering a creative, cryptographic solution that effectively bars the double-spending of cryptocurrency units. For this reason, the terms "digital ledger" and "double spend" became associated with cryptocurrency blockchains. Yet these terms, respectively, broadly apply to tracking ownership and enforcing a single transfer of digital assets. When you see these terms, don't assume they refer solely to cryptocurrency-oriented blockchain technologies.

At its essence, a blockchain is a tamper-proof data structure that tracks something of value or interest as it passes from owner to owner. That "something" can be any kind of digital asset—such as a digital coin, a Word document or the serial number of a Microsoft Surface tablet. In fact, any item that can be associated with a unique digital fingerprint can be tracked on a blockchain. Blockchains solve the so-called "double-spend" problem by requiring that ownership of a digital asset be transferred rather than copied or shared. But what makes a blockchain technology interesting is that it establishes a protocol, enforces transaction rules, and is able to let the nodes on its distributed network of computers self-police the entire operation. And it accomplishes this remarkable feat with no central server or trust authority, speedily and globally (that is, internationally). This promise excites those who see it as a way to eliminate middlemen and reduce or waive transaction fees, making commerce more efficient for businesses and consumers alike.

Blockchain Core Components

The Bitcoin blockchain network is public—anyone can participate anywhere in the world. Yet newer blockchain offerings, such as the Microsoft Azure-hosted blockchain, can be configured as public, private or permissioned networks. Blockchains are considered to be decentralized, but that term requires clarification: As Vitalik Buterin explains (bit.ly/2tEUYyT), "decentralized blockchains" means they're "politically decentralized (no one controls them) and architecturally decentralized (no infrastructural central point of failure) but they are logically centralized (there is one commonly agreed state and the system *behaves* like a single computer)." Decentralization offers fault tolerance, attack resistance and collusion resistance (the meaning of this will become clear when I discuss proof-of-work later).

At its essence, a blockchain is an effectively tamper-proof data structure that tracks something of value or interest as it passes from owner to owner.

Understanding how to engineer a public blockchain requires knowledge of cryptographic hashes, public key cryptography (PKC), binary hash chains (Merkle trees, in particular), and consensus algorithms. I'll briefly review these concepts, and then I'll show that a blockchain is a hash chain that contains a hash chain of transactions. Once you grasp this nested-hash-chain concept, you'll understand blockchain technology's fundamental design.

Cryptographic Hashes While there are many one-way cryptographic hash algorithm variants, a popular choice is to leverage SHA-256 (bit.ly/29kkpft), a one-way hash function that accepts a message of up to $(2^{64}-1)/8$ bytes and returns a 32-byte hash value (64 hexadecimal characters) in the decimal range of 0 to roughly 1.16×10^{77}. To put that number's magnitude in perspective, a drop of water has about 5×10^{12} atoms; the observable universe is estimated to have in the range of 10^{78} to 10^{82} atoms. Tweaking any character in the message and re-computing the SHA-256 hash value generates an entirely new hash value. (To experiment, visit onlinemd5.com and set the file or text checksum type to SHA-256.)

Given the same input, the SHA-256 algorithm always produces the same fixed-length output. With respect to blockchain technologies, the value of using SHA-256 cryptographic hashes

Figure 1 **Hashing Strings of Various Lengths Using the SHA-256 Algorithm**

Input String	SHA-256 Hash Value
m	62C66A7A5DD70C3146618063C344E531E6D4B59E379808443CE962B3ABD63C5A
M	08F271887CE94707DA822D5263BAE19D5519CB3614E0DAEDC4C7CE5DAB7473F1
M1	2D214CA69B86C255BE416D42CCA977A59B34A7492873105522C35015FAB806F0
M2	0892A10ECE1F933EE98F5D554601B28F8437801D1AA1B77799E4035DDCB6950C
March	9D95A2CF0D7180B5089691163B188A7203B0CDE179346B8CFAA8AB6C2C3E6414
March 1, 2018	767328E7367048FA9DB37354CFA43DBB1691E8330DB54D54F52C1A444CA2E680
March 2, 2018	CCF33BF1C08B74EDE6A7C15C56EEC16269D83967670032ACDA6EE395361B7595

Figure 2 **Double-Hashing the Values in Figure 1**

Input String	Double SHA-256 Hash Value
m	A5FCE7E78734EC317F83F9019C80FFAF2508689B06EFA02191495A7D21FECE9A
M	6F6DCF58526B0D29EE664A708A939B7CDAC124A6A8569FCACE46FEAD38868E2E
M1	6C5D08BE9FFBBABD24B5F19AFFE6590FD402D347A50B519A59D40E15DCC0A6CB
M2	B2307311CC5877D5A581EDC821F3BFD5F99EB4E3B1D1B4009D9545BCF07E2E1A
March	B5410E155022AE6EB22CA21FADEDE65F0F7296DE14CA1D7A720A4937BD23AA5D
March 1, 2018	345DD725FEE80F8C5953A66C1495605E4ED01C4CE5AEF6C0A6D238999266A1A6
March 2, 2018	3E85B3D910BA77F88ECD5E24D1396457C532C73B89C032DED9AD0CBB4D4D9794

is that they're unique enough to serve as a sort of digital fingerprint while also acting as a checksum. Furthermore, one-way hash functions can't (as a matter of practice) be decoded. Consider the SHA-256 value for my name: 8F12D83BA54AC0EA7687AD4AFDE5E258BBFF970AA8D60C6588381784C502CA8E. Given that hash value, there's no practical way to algorithmically reverse it back to my name. (One hacking technique leverages rainbow tables that list already-calculated hash values for common strings, such as "password"—but that's not algorithmically reversing the hash. To thwart such exploits, it's customary to embellish the string to be hashed by tacking on a random string, known as a "salt" value.)

If you don't have a SHA-256 generator handy, consider the table in **Figure 1**, which shows how strings of various lengths always produce a 64-digit hexadecimal hash value, and that a small change to any particular string produces a completely different result.

Sometimes a hash value is double-hashed, which means that the first hash is hashed again by applying a second round of the SHA-256 algorithm. If I double-hash the values in **Figure 1**, I end up with the results in **Figure 2**.

Public Key Cryptography Recall that one of the primary functions of a blockchain is to track ownership of a digital asset. The digital asset in question may be worth nothing or many millions of dollars, so the ownership test must ensure that the owner can't be spoofed. To conduct such a test in the digital realm, blockchains leverage PKC, which enables the owner to digitally sign their asset in order to prove ownership and authorize it to be transferred. Unlike symmetric key encryption, wherein a single private (secret) key is used to both encrypt and then decrypt a message, PKC uses asymmetric key encryption.

Because an accurate validation algorithm of digital asset ownership is critical for blockchains, they employ a high-strength public/private key-pair generation strategy that relies on the Elliptic Curve Digital Signature Algorithm, or ECDSA. The beauty of ECDSA is that it creates keys that are shorter in length but cryptographically stronger than same-length keys generated by the usual algorithm: Digital Signature Authorization (DSA). Whenever they're needed, users access a software application that uses ECDSA to generate the cryptographic key pair. The user must retain a backup of the private key because that key is required to transfer or harness the value held in a digital asset that's stored on a blockchain. If you have access only to the private key in a private/public key pair, you can regenerate the public key because there's a mathematical relationship between the two keys. But the private key can't be generated from the public key (which means if you back up only one key, be sure it's the private key!).

These keys typically are used in one of two ways. The first use case (see **Figure 3**) is when you want someone to send you an encrypted message that only you can open. To do that, give the other person your public key and ask them to use it to encrypt the document using software that applies an encryption algorithm and produces a ciphertext—the encrypted version of their message. They then send you only the ciphertext. Because they used your public key to encrypt the document, you must use the correctly paired private key to decrypt it.

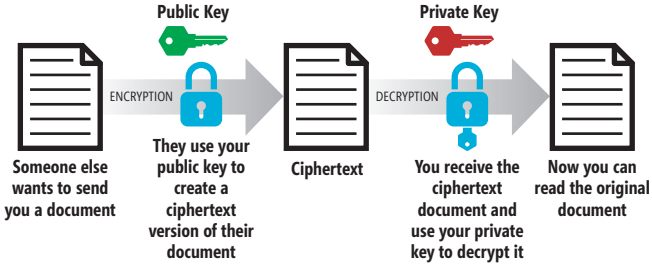


Figure 3 **Using PKC When You Want Someone to Send You an Encrypted Document/Message That Only You Can Open**

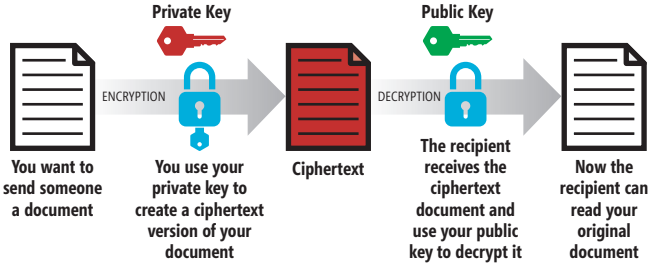


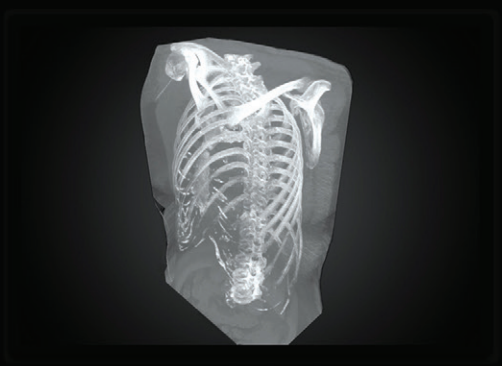
Figure 4 **Using PKC When You Want to Send Someone an Encrypted Document/Message to Assure Them That It Indeed Came from You**

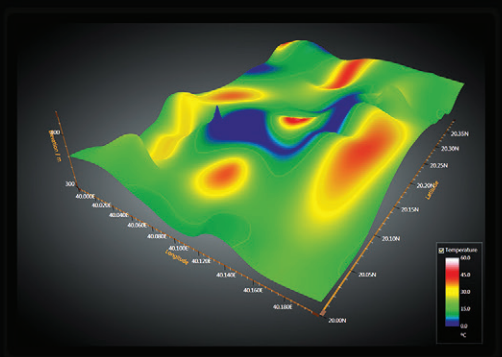
Arction

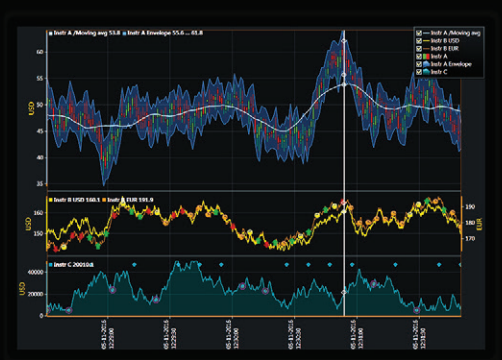
[WPF]
[Windows Forms]
[Free Gauges]
[Data Visualization]
[Volume Rendering]
[3D / 2D Charts] [Maps]

LightningChart®

The fastest and most advanced charting components







Create eye-catching and powerful charting applications for engineering, science and trading

- DirectX GPU-accelerated
- Optimized for real-time monitoring
- Supports gigantic datasets
- Full mouse-interaction
- Outstanding technical support
- Hundreds of code examples

NEW
- Now with Volume Rendering extension
- Flexible licensing options

Get free trial at
LightningChart.com/ms

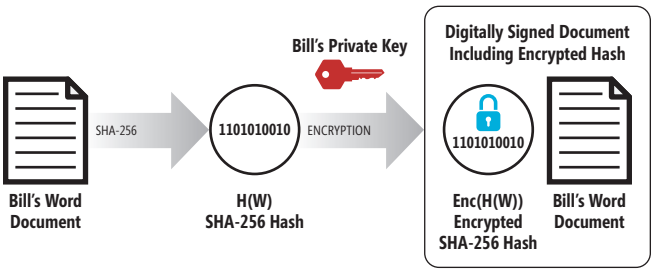


Figure 5 **Using PKC Along with a Cryptographic Hash to Digitally Sign a Document/Message**

The second use case (see **Figure 4**) is when you want to encrypt a message and demonstrate that it is indeed from you. To do that, you use your private key to create a ciphertext of your document. You then send that ciphertext to someone else. They use your public key to decrypt it. Because only your public key can decrypt that document, the recipient can assume that the document was encrypted by your private key—and unless your private key has been misappropriated, the document came from you.

A third use case employs PKC to prove ownership of a digital asset through a digital-signing process. In this use case (see **Figure 5**), imagine that Bill has composed a Word document of a legally binding document that he needs to e-mail to Susan. Susan wants to be sure that the copy of the document she receives from Bill is actually from Bill and hasn't been tampered with en route. Bill first creates a SHA-256 hash of the Word document and records the value as H(W). He next uses his private key to encrypt the document hash, resulting in Enc(H(W)), and then sends the Word document (optionally encrypted) and the Enc(H(W)) value (this is Bill's digital signature for document W) to Susan.

Susan re-computes the H(W) value from the copy of the Word document she received, then uses Bill's public key to decrypt the Enc(H(W)) value (see **Figure 6**). If the hash Susan computed is equal to the decrypted H(W) value, Susan can conclude that Bill signed the document and that the copy she received is exactly the same as the one Bill signed.

Using hashing and PKC, a blockchain maintains a history of digital asset ownership using transactions. Transaction data objects are linked to each other, forming a data structure called a hash chain. The way this works is that each transaction record constitutes the message (m) that's hashed using function (H) and then signed using the owner's private key (s). (It's customary to use "s," for "secret," to represent the private key so as not to confuse it with "p" for the public key.) This produces a signature (sig):

sig = signature(H(m), s)

When a digital asset transfers from one owner to another, its digital signature is examined, verified, and digitally signed by the new owner, and then registered as a new node on the hash chain. Although the details of the implementation vary dramatically across blockchain technologies and versions, the basic idea is the same for all of them. For example, as shown in **Figure 7**, Bill is the owner of a digital asset and uses his private key to initiate a transfer of that digital asset to Susan. Susan's transaction record uses Bill's public key to verify his signature. After this, Susan's public key is used to sign the digital asset, making Susan the new owner. This creates a new transaction record—a new link on the transaction hash chain.

This hash chain of transactions is cryptographically secure and tamper-proof. Any change to Transaction 0 would cause Sig0 to change, and that would require an update to the hash value stored in Transaction 1 and every subsequent transaction on the hash chain.

The transaction objects here are pictured with data. The data contained by each transaction varies for each blockchain implementation, so I've abstracted the underlying data for our purposes because the main point to understand is that the hash chain is a cryptographically linked chain of transactions—linked by the hash value of the previous owner's transaction record. (In cryptocurrency blockchains, each transaction object contains a list of digital-currency inputs and outputs, along with metadata, such as a timestamp and optional transaction fee. These cryptocurrency inputs and outputs provide the transaction detail needed to accurately model a financial ledger.)

Merkle Trees Some blockchains bundle up transactions using another kind of hash chain: the binary hash chain, or Merkle tree. A complete Merkle tree is referred to as a binary tree structure because it branches twice at each level starting at the root, as shown in **Figure 8**.

The work in setting up a Merkle tree is to create a series of leaf nodes by computing the SHA-256 hash for the data contained in each transaction object (the Bitcoin blockchain double-hashes each Merkle node; double-hashing can help strengthen the cryptographic value in the hash result should a vulnerability be discovered in the SHA-256 algorithm). The Merkle tree requires an even number of leaf nodes—it's customary to duplicate the last leaf node if starting with an odd number. Then each pair of leaf nodes is hashed together, producing a new hash value. In **Figure 8**, Leaf A shows the hash for Transaction A as H_A; Leaf B shows the hash for Transaction B as H_B and so on. This pattern continues at each tree level until you reach the final root node. The root node's hash value is the cryptographic hash sum of all of the other hash sums in the tree. Any change to the data in any of the leaf nodes causes the recomputed Merkle tree root hash value to change.

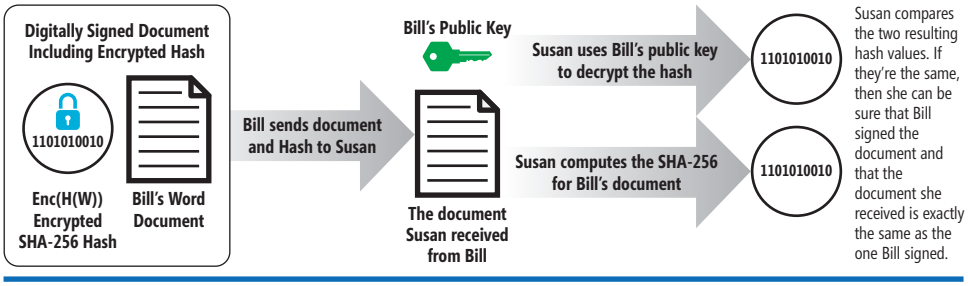


Figure 6 **Using PKC Along with a Cryptographic Hash to Verify That a Document/Message was Signed by the Expected Party**

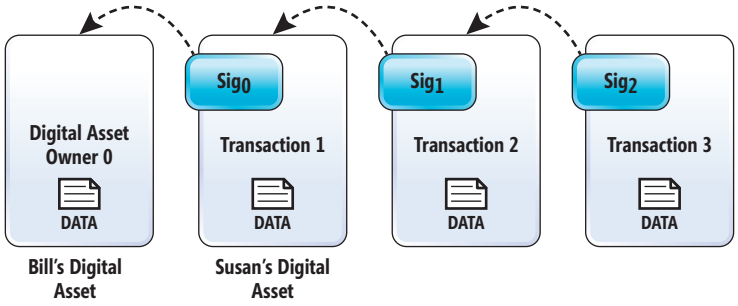


Figure 7 **The Transaction Hash Chain Uses Digital Signatures to Transfer Ownership of a Digital Asset; Each Transaction Record Maintains a Cryptographic Backlink to the Previous Transaction on the Hash Chain**

The Merkle binary hash tree structure offers some advantages. For example, it makes it easy to update data within a transaction and compute a new Merkle root hash without having to build the entire Merkle tree from scratch. For example, if Transaction E changes (it's highlighted in **Figure 8**), all you need to do is walk the tree efficiently back to the Merkle root, computing new hashes once for each level. Thus, you first compute the new Leaf hash H_E; then compute H_{EF} from H_E and H_F; then compute H_{EFGH} from H_{EF} and H_{GH}; then compute a new Merkle root hash from H_{ABCD} and H_{EFGH}. Updating the Merkle root hash required only four computations versus the 15 required to build the Merkle tree from scratch!

Building a Blockchain

To build a blockchain (see **Figure 9**), the binary hash chain data object containing transactions must somehow be committed to a tamper-proof data store that everyone can use (remember, this is a public blockchain—any node on the network can read from or write to it). The Merkle tree structure contains transactions and is tamper-proof, so it would seem it could serve as the blockchain. But there are several problems. In order for Bill to send his digital asset to Susan, Bill must trust the service or Web site that acts as an agent to process his digital-asset transfer request, and he must trust the server that persists the hash structure. Without a central node to process a new transaction or a central authority to delegate them for processing, any node could process Bill's pending transaction. A rogue or dominant node having superior processing power could allow invalid or fraudulent transactions to occur and those could propagate to honest nodes. To solve that, the network could try to randomly assign a node to process Bill's transaction, but that again centralizes control and requires trust that the random number generator is indeed enforcing randomness. To eliminate this issue, blockchains use consensus algorithms, described next.

Consensus Algorithms Blockchain technologies avoid centralized data store and trust-authority issues by honoring a protocol that dictates how blocks are added and maintained. They do so by enforcing a blockchain-building consensus algorithm. There are different kinds of consensus algorithms—I'll discuss how the proof-of-work (PoW) algorithm works.

PoW is based on the idea that a node on the network needs to prove its honorable intentions by incurring a cost and consuming the time required to solve a computationally challenging problem. To induce a node to participate in such a system and play by its rules, the network offers an incentive—often money—that is, node operators get paid when they add a block to the blockchain. To earn that money, nodes must validate all transactions (to ensure that they meet a blockchain's specific rules) and then solve a cryptographic puzzle.

Earlier, I mentioned that a central authority could randomly assign a node to process a batch of new transactions. That approach would require a central random-number generator, which could be flawed, hacked or disabled. However, giving nodes a difficult puzzle to solve yields the desired effect: The node that will first solve the puzzle can't be determined in advance, making for a sort of node-lottery on the network. No central authority is needed—and that's one of

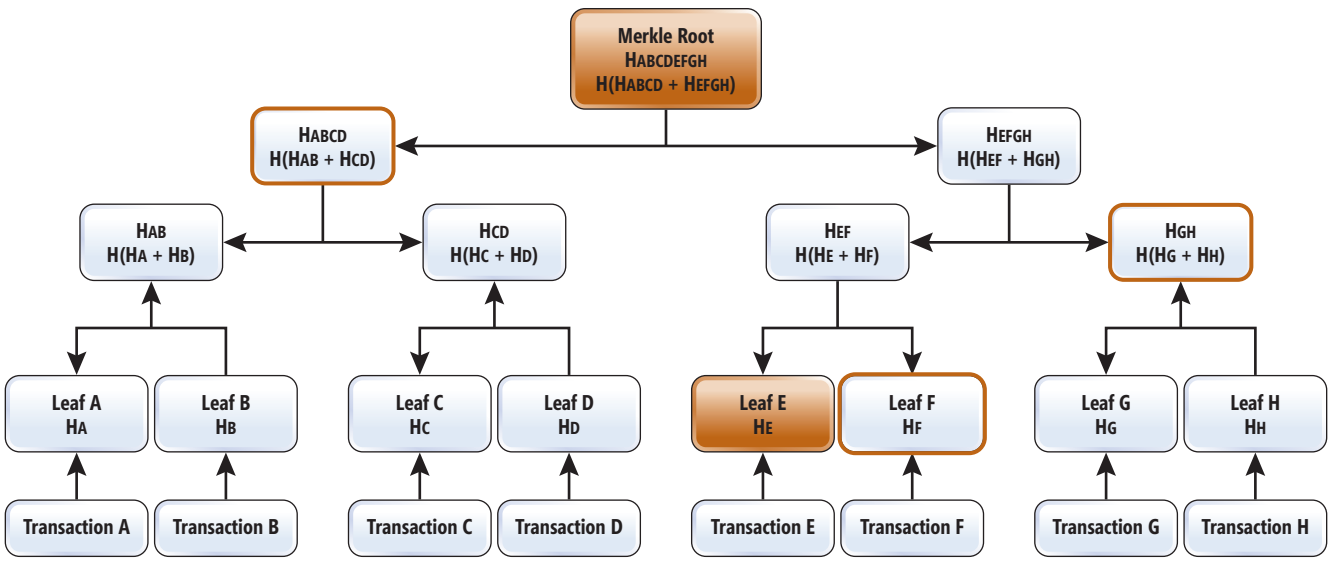


Figure 8 **A Merkle Tree is a Type of Binary Hash Tree That Produces a Merkle Root Hash; This Data Structure Can Efficiently Add Leaf Nodes and Calculate a New Merkle Root Without Requiring Full Re-Computation**

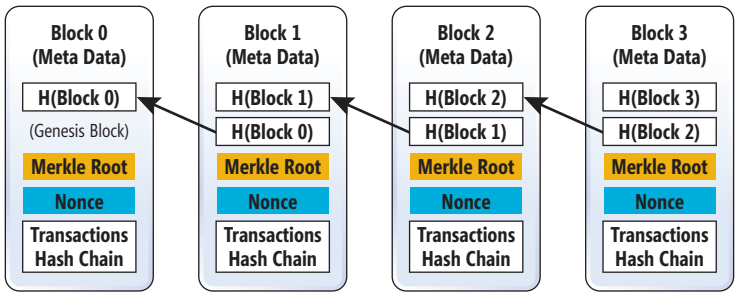


Figure 9 **The Blockchain Is Composed of Blocks That, in Turn, Include Transaction Hash Trees; Blocks on the Blockchain Are Back-Linked to Previous Blocks and Are Validated Using a Proof-of-Work Algorithm**

the key innovations of blockchain technologies. I also mentioned that blockchains are decentralized and therefore offer "collusion resistance." Because PoW takes time and costs money in computational power, it's practically impossible for any single node or group of nodes to collude on the network and have a blockchain-making advantage over other peer nodes. (There is a "51 percent attack" risk that suggests collusion can occur if a group of nodes ends up with 51 percent of the computational power, but employing a PoW consensus algorithm makes the possibility of such an attack infeasible.)

To construct a block of transactions, a node grabs unprocessed transactions stored on the network and builds a Merkle tree in order to compute the Merkle root hash. Therefore, a candidate block contains a list of transactions along with a block header that includes the Merkle root hash value, a current timestamp and PoW difficulty level (and sometimes additional header data). Then it must solve the PoW puzzle, which involves computing a double hash of the entire 256-bit block hash value, then finding a 32-bit number, called the nonce, which can be concatenated to it so that the hash of the resulting 288-bit number produces a result that has a certain number of leading zeroes. The 32-bit nonce has a range of 0 to 2^{32} (4,294,967,295), so instead of just trying to guess the nonce, it's typical to start with nonce 0, generate the SHA-256 hash, and determine if it has the target number of leading zeroes (that is, the resulting hash is below a target value); if it doesn't, the node increments the nonce value and tries again. If the node attempts all nonce values without solving the puzzle, its recalculates the block hash value. This guarantees the production of a different block hash value because a timestamp in the block header is included in the block-hash calculation. At any time, a node can select a different batch of pending transactions for inclusion in the new block (or add new pending transactions that might have appeared since it last checked), and this changes the Merkle root hash value, which, along with the timestamp, changes the newly computed block hash value. Each time the block hash is recomputed, the node iterates over all 4 billion-plus nonces again.

In time, some node on the network will solve its cryptographic puzzle. When it does so, it adds the new block to the end of its copy of the blockchain (every node maintains a copy of the blockchain) and then broadcasts the new block to every other node on the network so that they can update their blockchain copy. When a node broadcasts a new block, other nodes don't simply trust that a new block is valid—they prove it to themselves by validating the block. To validate, a node simply verifies the PoW puzzle solution by computing the block's SHA-256 hash concatenated with the nonce value and verifies that the answer produces a hash that has the number of leading zeroes specified by that block's PoW difficulty value.

Incidentally, on some blockchains, the PoW difficulty value is continually adjusted by the protocol so that new blocks are added to the blockchain at a prescribed time interval. This continual adjustment is necessary because nodes are constantly appearing and disappearing from the network and thus the average computational power of the nodes is always changing. Remember that in PoW, there is an incentive to add blocks to the blockchain, so node administrators often beef up their hardware in order to compete for an award. On the Bitcoin blockchain, the difficulty value is tweaked every 2016 blocks so that blocks continue to be added at an average rate of 10 minutes per block.

Sometimes branches occur. That's because on a large network, new-block propagation takes time. It's possible that during propagation, another node solves its PoW puzzle, adds a new block to its copy of the blockchain, then broadcasts that blockchain on the network. Receiving nodes will always add a valid block to their copy of the blockchain—and because each block is cryptographically tethered to the previous block, two new blocks published by two different nodes will result in a branch linked to the same block at the end of the chain. That's OK, however. Over time, nodes will add new blocks to the end of what the protocol deems to be the "longest chain." For example, given a branch, the longest chain could be defined as the one with the most-recent block timestamp. Over time, a single branch will prevail in length and blocks from abandoned (shorter) branches will be removed, returning their transactions to the pool of unprocessed transactions.

Wrapping Up

In this article, I've shown how to construct a public blockchain that's composed of cryptographically linked blocks—each of which contains its own hash chain of cryptographically linked transactions—on a decentralized peer-to-peer network of nodes. I covered the basics of blockchain technologies, trying not to focus on any single implementation but instead concentrating on some of the more typical technical features they all share. If you want to explore the subject further, I suggest choosing a specific blockchain technology, such as Bitcoin, Ethereum or Ripple, and trying to master the details of its particular implementation. If you want to experiment with blockchains on your own, take a look at Azure-hosted blockchain offerings at bit.ly/2Gj2zaC. ■

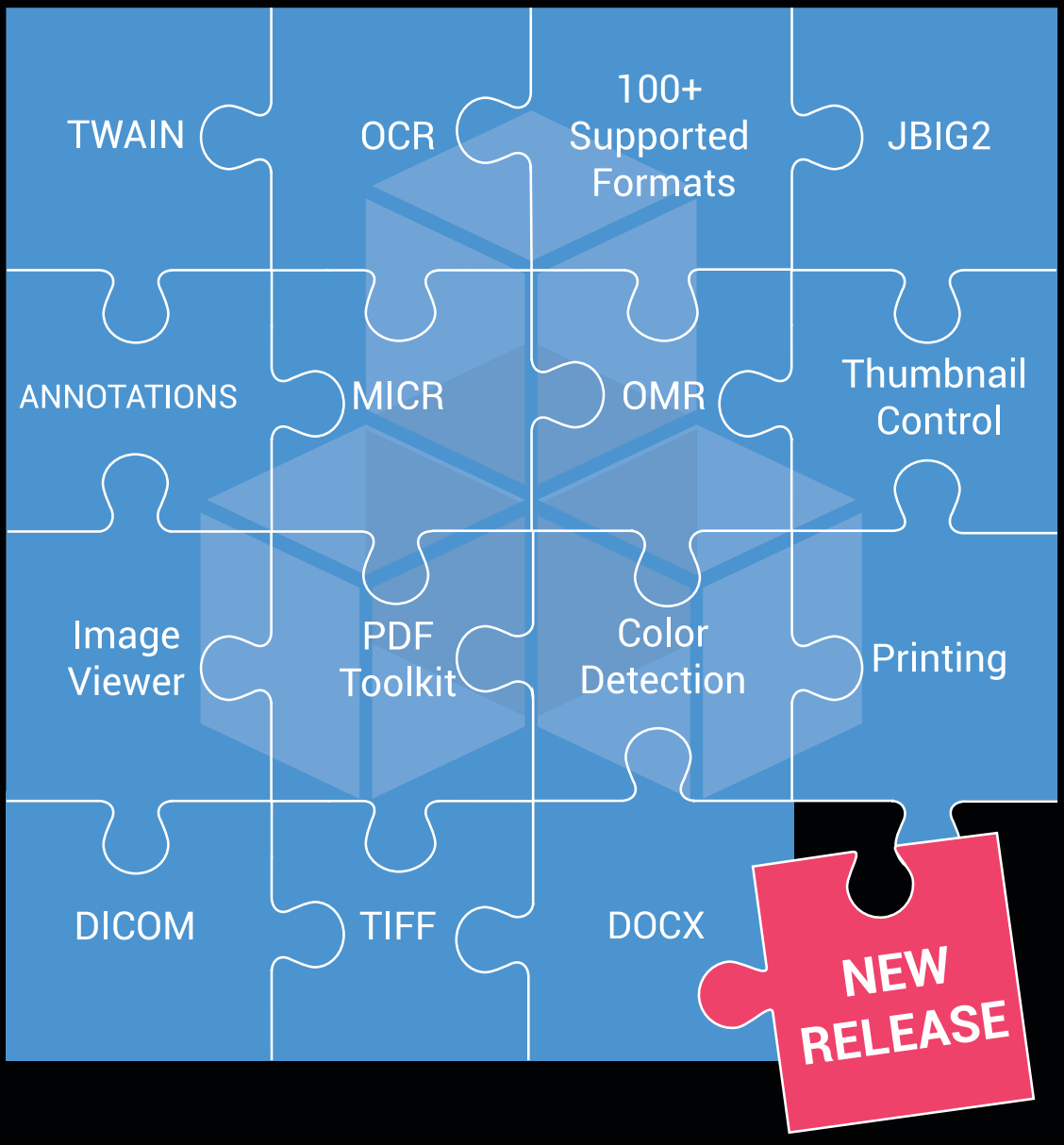
Jonathan Waldman *is a Microsoft Certified Professional who has worked with Microsoft technologies since their inception and who specializes in software ergonomics. Waldman is a member of the Pluralsight technical team and he currently leads institutional and private-sector software-development projects. He can be reached at jonathan.waldman@live.com.*

Thanks to the following Microsoft technical expert for reviewing this article: *James McCaffrey*

GdPicture.NET 14

100% ROYALTY FREE

Imaging SDK For **WinForms**, **WPF** And **Web** Development



Leverage your apps. with **GdPicture.NET** Imaging Toolkit

DOWNLOAD YOUR FREE TRIAL **www.gdpicture.com**

GdPicture.NET is an ORPALIS imaging technologies product

Use Razor to Generate HTML for Templates in a Single-Page App

Nick Harrison

Single-Page Application (SPA) apps are very popular and for good reason. Users expect Web apps to be fast, engaging and work on every device from smartphones to the widest-screen desktops. Beyond that, they need to be secure, visually engaging and do something useful. That's a lot to ask for from a Web app, but really this is only the starting point.

As users started expecting more from Web applications, there was an explosion of client-side frameworks to "help" by providing a client-side implementation of the Model-View-ViewModel (MVVM) pattern. It can seem like there's a new framework every week. Some have proven to be helpful, others not so much. But they all implement the design pattern differently, adding new features or solving recurring problems in their own ways. Each framework takes a different approach toward the design pattern, with a unique syntax for templates and introducing custom concepts like factories, modules and observables. The result is a sharp learning curve that can frustrate developers trying to stay current as new frameworks go in and out of vogue. Anything we can do to ease the learning curve is a good thing.

Having client-side frameworks implement the MVVM pattern and ASP.NET implement the MVC pattern on the server has led to some confusion. How do you blend server-side MVC with client-side MVVM? For many, the answer is simple: They don't blend. The common approach is to build static HTML pages or fragments and serve them up to the client with minimal server-side processing. In this scenario, Web API controllers may often replace the processing that would have been handled by the MVC Controller in the past.

As **Figure 1** shows, there are minimal requirements for the Web Server and a full-fledged server for the Web API, and both servers need to be accessible from outside the firewall.

This arrangement provides good separation of concerns, and many applications have been written following this pattern, but as a developer, you're leaving a lot on the table. ASP.NET has a lot more

This article discusses:

- Using scaffolding to automate generating views based on models shared between the Web application and the API
- Using T4 to customize the views generated to support working with AngularJS and KnockOut
- Using EditorTemplates to provide global editors in the Web application based on the data type of the property in the underlying model
- Using jQuery Validations to enable client-side validations mirroring validations automatically performed on the server

Technologies discussed:

Visual Studio 2017, T4 Templates, Scaffolding Templates, Editor Templates

Code download available at:

bit.ly/2nps4yz

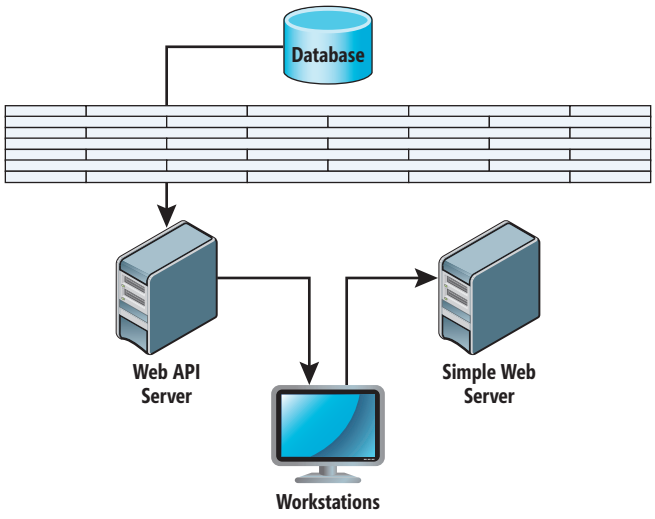


Figure 1 **Typical Application Layout**

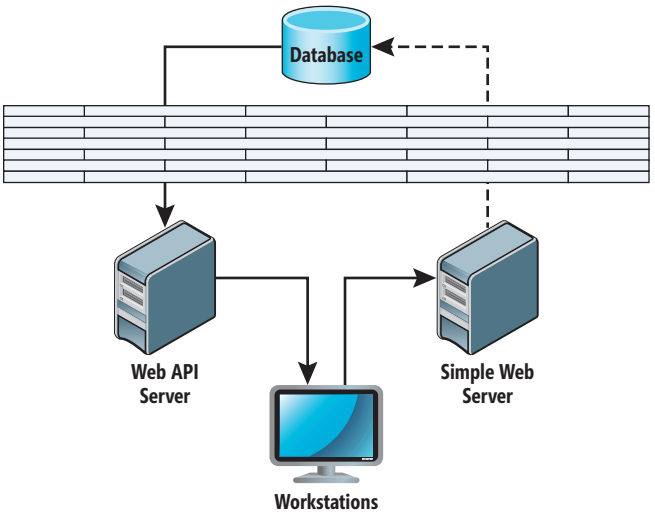


Figure 2 **Simple Flow Adding in Razor**

to offer and its implementation of MVC provides many features that are still relevant, even when one of the client-side frameworks does much of the heavy lifting. In this article, I'll focus on one of these features, the Razor View Engine.

Razor in the SPA

The Razor View Engine is a marvel when it comes to simplifying generation of HTML markup. With just a couple of tweaks, the generated markup is easily customized to fit the template expectations for whatever framework is used on the client. I'll highlight some examples with Angular and Knockout, but these techniques will work regardless of the framework you employ. If you make calls back to the server to provide templates for your application, you can now use Razor to do the heavy lifting generating the HTML.

There's a lot to like here. EditorTemplates and DisplayTemplates are still your friends, as are scaffolding templates. You can still inject partial views, and the fluid flow of the Razor syntax is still there to use to your advantage. Beyond the views you can also use the MVC controller, which can add processing power to speed things up or add an extra layer of security. If this isn't needed, the controllers could be a simple pass through to the view.

To demonstrate how this can be used to your benefit, I'll step through the creation of some data entry screens for a time-tracking application, showing how Razor can help expedite the creation of HTML suitable for use by either Angular or Knockout.

As **Figure 2** shows, not much changes from the typical layout. I've included an option showing that the MVC application could interact with this database. This isn't necessary at this stage, but if it simplifies your processing, it is available. In general, the flow will look like:

- Retrieve a full page from the MVC application, including all style sheet and JavaScript references and minimal content.
- Once the page has been fully loaded in the browser and the framework initialized, the framework can call the MVC server to request a template as a Partial View.
- This template is generated with Razor, typically without any associated data.
- At the same time, the framework makes a call to the API to get the data that will be bound to the template received from the MVC site.
- Once the user makes any required edits, the Framework makes calls to the Web API server to update the back-end database.

This workflow repeats as needed when a new template is requested. If the framework allows you to specify a URL for a template, MVC can serve up the template. While I will show examples generating views suitable for binding in Angular and Knockout, keep in mind that these are hardly the only options.

Setting up a Solution

From a setup perspective, you need at least three projects. One for the Web API, another for the MVC application and, finally, a common project to host code common between these two projects. For purposes of this article, the common project will host the ViewModels so they can be used in both the Web and Web API. The initial project will be structured as shown in **Figure 3**.

In the real world, you'll only support one client-side framework. For this article, it simplifies matters to have two MVC projects, one for each of the frameworks to be demonstrated. You may encounter something similar if you need to support a Web application, a customized mobile application, a SharePoint application, a desktop application or any other scenario that's not easily rendered from a common UI project. Regardless, only the logic embedded in the UI project will have to be repeated as you support multiple front ends.

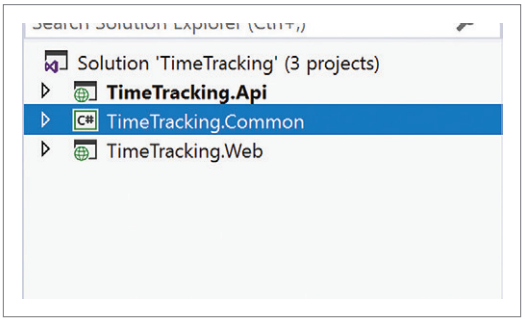


Figure 3 **Initial Project Structure**

Figure 4 **Simple TimeEntryViewModel**

```
namespace TimeTracking.Common.ViewModels
{
  public class TimeEntryViewModel
  {
    [Key]
    public int Id { get; set; }
    [Required (ErrorMessage ="All time entries must always be entered")]
    [DateRangeValidator (ErrorMessage ="Date is outside of expected range",
      MinimalDateOffset =-5, MaximumDateOffset =0)]
    public DateTime StartTime { get; set; }
    [DateRangeValidator(ErrorMessage = "Date is outside of expected range",
      MinimalDateOffset = -5, MaximumDateOffset = 0)]
    public DateTime EndTime { get; set; }
    [StringLength (128, ErrorMessage =
      "Task name should be less than 128 characters")]
    [Required (ErrorMessage ="All time entries should be associated with a task")]
    public string Task { get; set; }
    public string Comment { get; set; }
  }
}
```

Bringing in Data

In practice, your data would be stored in a database, probably using an object relational mapper (ORM) such as Entity Framework. Here, I'll sidestep issues of data persistence to focus on the front end. I'll rely on the Web API controllers to return hardcoded values in the Get actions and I'll run these samples in a perfect world where every API call returns successful. Adding appropriate error handling will be left as an exercise for you, the intrepid reader.

For this example I'll use a single View Model decorated with attributes from the System.ComponentModel.DataAnnotations namespace, as shown in **Figure 4**.

The DateRangeValidator attribute doesn't come from the DataAnnotations namespace. This isn't a standard validation attribute, but **Figure 5** shows how easily a new validator can be created. Once applied, it behaves just like the standard Validators.

Anytime the model is validated, all validators will run, including any custom validators. Views created with Razor can easily incorporate these validations client-side, and these validators are automatically evaluated on the server by the Model Binder. Validating user input is key to having a more secure system.

API Controllers

Now that I have a View Model, I'm ready to generate a controller. I'll use the built-in scaffolding to stub out the controller. This will create methods for the standard verb-based actions (Get, Post, Put, Delete). For purposes of this article, I'm not worried about the details for these Actions. I'm only interested in verifying that I have the endpoints that will be called from the client-side framework.

Creating a View

Next, I turn my attention to the View that the built-in scaffolding produces from the View Model.

In the TimeTracking.Web project, I'll add a new Controller and name it TimeEntryController and create it as an Empty Controller. In this Controller, I create the Edit action by adding this code:

```
public ActionResult Edit()
{
  return PartialView(new TimeEntryViewModel());
}
```

Figure 5 **Custom Validator**

```
public class DateRangeValidator : ValidationAttribute
{
  public int MinimalDateOffset { get; set; }
  public int MaximumDateOffset { get; set; }
  protected override ValidationResult IsValid(object value,
    ValidationContext validationContext)
  {
    if (!(value is DateTime))
      return new ValidationResult("Inputted value is not a date");
    var date = (DateTime)value;
    if ((date >= DateTime.Today.AddDays(MinimalDateOffset)) &&
      date <=  DateTime.Today.AddDays(MaximumDateOffset))
      return ValidationResult.Success;
    return new ValidationResult(ErrorMessage);
  }
}
```

From inside this method, I'll right-click and select "Add View." In the popup, I'll specify that I want an Edit template, selecting the TimeEntryViewModel template.

In addition to specifying the model, I make sure to specify creation of a partial view. I want the generated view to include only the markup defined in the generated view. This HTML fragment will be injected into the existing page on the client. A sampling of the markup generated by the scaffolding is shown in **Figure 6**.

There are a few key things to note with this generated view, which out of the box generates a responsive, bootstrap-based UI. These include:

- The form group is repeated for each property in the View Model.
- The Id property is hidden because it's marked with the Key attribute identifying it as a key that's not to be modified.
- Each input field has associated validation message placeholders that can be used if unobtrusive validation is enabled.
- All labels are added using the LabelFor helper, which interprets metadata on the property to determine the appropriate label. The DisplayAttribute can be used to give a better name, as well as handle localization.

Figure 6 **Razor Markup for the Edit View**

```
@model TimeTracking.Common.ViewModels.TimeEntryViewModel

@using (Html.BeginForm())
{
  @Html.AntiForgeryToken()

  <div class="form-horizontal">
    <h4>TimeEntryViewModel</h4>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    @Html.HiddenFor(model => model.Id)
    <div class="form-group">
      @Html.LabelFor(model => model.StartTime, htmlAttributes:
        new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.StartTime,
          new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.StartTime, "",
          new { @class = "text-danger" })
      </div>
    </div>
    ...
  </div>
}

<div>
  @Html.ActionLink("Back to List", "Index")
</div>
```

- Each input control is added using the EditorFor helper, which interprets the metadata on the property to determine the appropriate editor.

The metadata is evaluated at run time. This means that you can add attributes to your model after generating the view and these attributes will be used to determine the validators, labels and editors, as appropriate.

Because the scaffolding is a one-time code generation, it's OK to edit the generated markup, as that's generally expected.

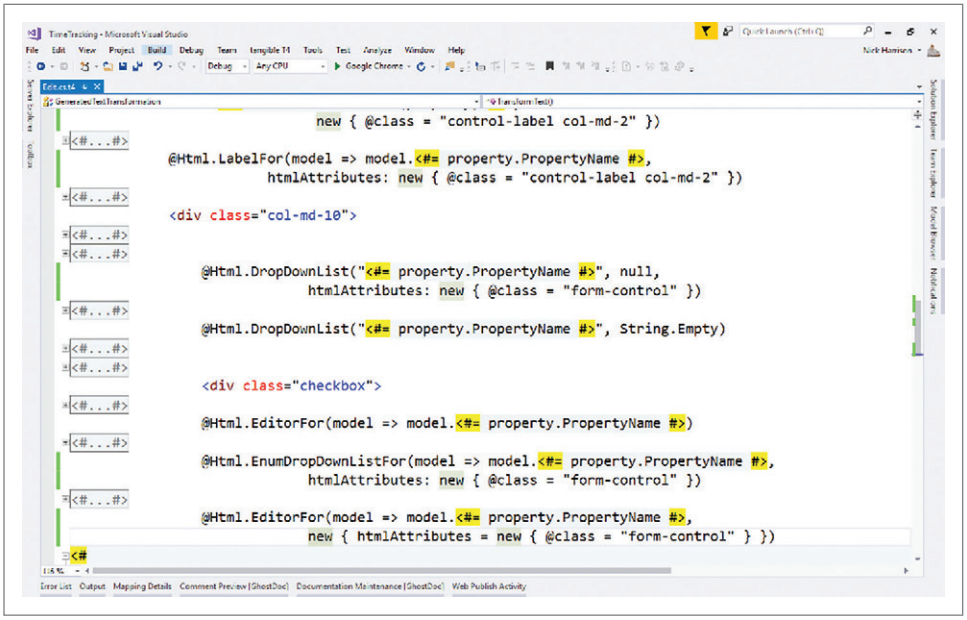


Figure 7 **The Visual Studio Edit T4 Template**

Binding to Knockout

To make the generated markup work with Knockout, I need to add a couple of attributes to the generated markup. I won't delve deeply into the internal workings of Knockout except to note that binding uses a data-bind attribute. The binding declaration specifies the type of binding and then the property to be used. I'll need to add a data-bind attribute to the input controls. Looking back at the generated markup, you see how the class attribute is added. Following the same process, I can modify the EditorFor function, as shown in the code here:

```
@Html.EditorFor(model => model.StartTime,
  new { htmlAttributes = new { @class     = "form-control",
                               data_bind = "text: StartTime" } })
```

Using the markup generated out of the box from the scaffolding, this is the only change needed to add Knockout binding.

Binding to Angular

Data binding with Angular is similar. I can add either an ng-model attribute or a data-ng-model attribute. The data-ng-model attribute will keep the markup HTML5-compliant, but the ng-bind is still commonly used. In either case, the value for the attribute is simply the name of the property to bind. To support binding to an Angular controller, I'll modify the EditorFor function, using this code:

```
@Html.EditorFor(model => model.StartTime,
  new { htmlAttributes = new { @class     = "form-control",
                               data_ng-model = "StartTime" } })
```

There are a couple of more minor tweaks that come into play for defining the application and controller. Please refer to the sample code for the full working example to see these changes in context.

You can adopt a similar technique to make the generated markup work with whatever MVVM framework you're using.

Changing the Scaffolding

Because scaffolding uses T4 to generate the output, I can change what gets generated to prevent having to edit every View generated. The templates used are stored under the installation for Visual Studio. For Visual Studio 2017, you'll find them here:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\
Extensions\Microsoft\Web\Mvc\Scaffolding\Templates\MvcView
```

You could edit these templates directly, but this would affect every project you work on from your computer, and anyone else working on the project would not benefit from the edits made to the templates. Instead, you should add the T4 templates to the project and have the local copy override the standard implementation.

Just copy the templates you want to a folder called "Code Templates" in the root folder of your project. You'll find templates for both C# and Visual Basic. The language is reflected in the file name. You only need the templates for one language. When you invoke scaffolding from Visual Studio, it will first look in the CodeTemplates folder for a template to use. If a template isn't found there, the scaffolding engine will then look under the Visual Studio installation.

T4 is a powerful tool for generating text in general, not just code. Learning it is a large topic on its own, but don't worry if you're not already familiar with T4. These are very minor tweaks to the templates. You'll have no need to delve deeply into the internals to understand how T4 works its magic, but you will need to download an extension to add support for editing T4 templates in Visual Studio. The Tangible T4 Editor (bit.ly/2Flqiqt) and Devart T4 Editor (bit.ly/2qT04GU) both offer excellent community versions of their T4 Editors for editing T4 templates that provide Syntax Highlighting, making it easier to separate code that drives the template from code being created by the template.

When you open the Edit.cs.t4 file, you'll find blocks of code to control the template and blocks of code that is the markup. Much

Figure 8 **Original Code for Generating the Editors**

```
@Html.DropDownList("<#= property.PropertyName #>", null,
  htmlAttributes: new { @class = "form-control" })
@Html.DropDownList("<#= property.PropertyName #>", String.Empty)
@Html.EditorFor(model => model.<#= property.PropertyName #>)
@Html.EnumDropDownListFor(model => model.<#= property.PropertyName #>,
  htmlAttributes: new { @class = "form-control" })
@Html.EditorFor(model => model.<#= property.PropertyName #>,
  new { htmlAttributes = new { @class = "form-control" } })
@Html.EditorFor(model => model.<#= property.PropertyName #>)
```

TEXT CONTROL

TX Text Control X15

Automate your reports and create beautiful documents in Windows Forms, WPF, ASP.NET and Cloud applications.

Text Control Reporting combines powerful reporting features with an easy-to-use, MS Word compatible word processor. Users create documents and templates using ordinary Microsoft Word skills.

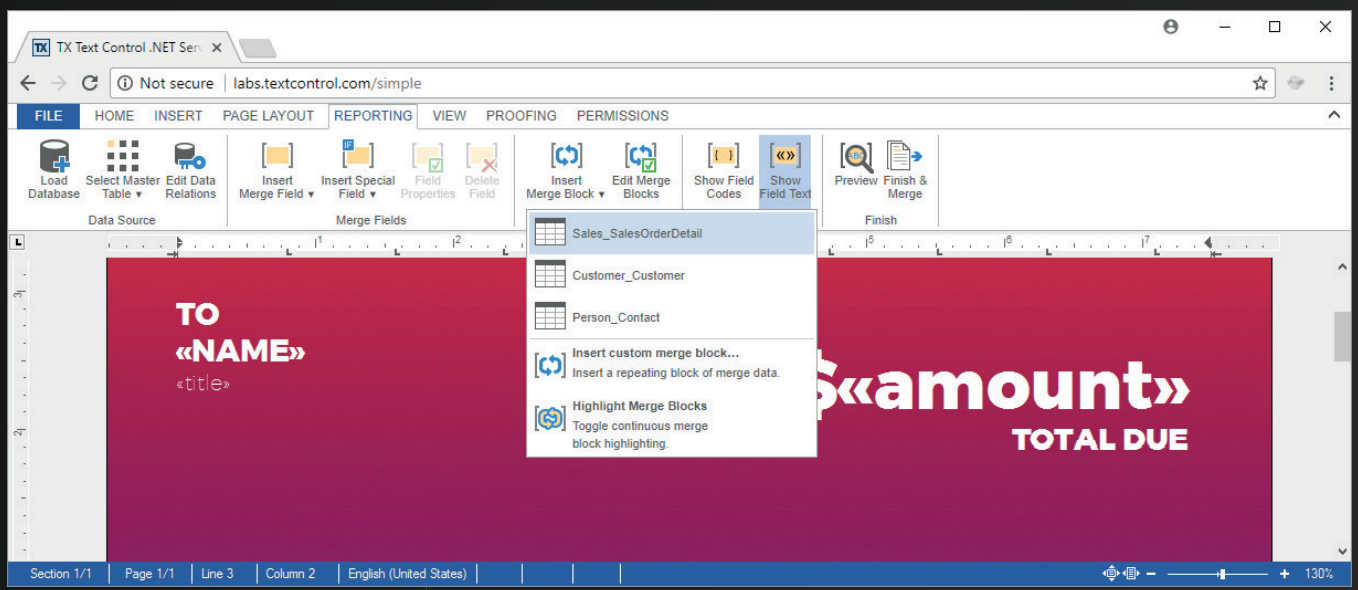
Download a free trial at
www.textcontrol.com

WE ARE CHANGING THE WAY YOU LOOK AT REPORTING

Visual Studio Partner

TX Text Control .NET Server for ASP.NET

Complete reporting and word processing for ASP.NET Web Forms and MVC



- Give your users a WYSIWYG, MS Word compatible, HTML5-based, cross-browser editor to create powerful reporting templates and documents anywhere.
- Text Control Reporting combines the power of a reporting tool and an easy-to-use WYSIWYG word processor - fully programmable and embeddable in your application.
- Replacing MS Office Automation is one of the most typical use cases. Automate, edit and create documents with Text Control UI and non-UI components.

WINDOWS FORMS | WPF | ASP.NET MVC | ASP.NET AJAX | CLOUD WEB API

TEXT CONTROL

Figure 9 **Markup for a DateTime Editor**

```
@model DateTime?
<div class="container">
  <div class="row">
    <div class='col-md-10'>
      <div class="form-group ">
        <div class='input-group date'>
         <span class="input-group-addon">
           <span class="glyphicon glyphicon-calendar"></span>
         </span>
         @Html.TextBox("", (Model.HasValue ?
           Model.Value.ToShortDateString() :
           string.Empty), new
           {
             @class = "form-control"
           })

        </div>
      </div>
    </div>
  </div>
</div>
```

of the code driving the template is conditional processing for handling special cases such as supporting partial pages and handling special types of properties such as foreign keys, enums and Booleans. **Figure 7** shows a sample of the template in Visual Studio when using an appropriate extension. The code driving the template is hidden in the collapsed sections, making it easier to see what the output will be.

Fortunately, you don't have to trace through these conditionals. Instead, look for the generated markup that you want to change. In this case, I care about six different statements. They're extracted for your reference in **Figure 8**.

Once you make the updates to support your specific client-side framework, all new views generated with the template will support your framework automatically.

Client-Side Validation

When I examined the view that was generated, I skipped over the ValidationMessageFor function calls that are made for each property. These calls produce placeholders to display any validation messages created when client-side validations are evaluated. These validations are based on the Validation attributes added to the model. All that's needed to enable these client-side validations is to add references to the jquery validation scripts:

```
@Scripts.Render("~/bundles/jqueryval")
```

The jqueryval bundle is defined in the BundleConfig class from the App_Start folder.

If you try to submit the form without entering any data, the required field validators will trigger to prevent the submission.

If you prefer a different validation strategy client-side, such as bootstrap form validation (bit.ly/2CZmRqR), you can easily modify the T4 template to not output the ValidationMessageFor calls. And if you don't use the native validation approach, you won't need to reference the jqueryval bundle, because it will no longer be needed.

Editor Templates

Because I specify the input control by calling the EditorFor html helper, I'm not explicitly specifying what the input control should be. Instead, I leave it to the MVC framework to use the most appropriate input control for the property specified, based on data type or attributes such as UIHint. I can also directly influence the editor selection by explicitly creating an EditorTemplate. This allows me to control at a global level how input of specific types will be treated.

The default editor for a DateTime property is a textbox. Not the most ideal editor, but I can change that.

I'll add a partial view called DateTime.cshtml to the folder \Views\Shared\EditorTemplates. The markup added to this file will be used as the editor for any property of type DateTime. I add the markup shown in **Figure 9** to the partial view and add the following code to the bottom of the Layout.cshtml:

```
<script>
  $(document).ready(function () {
    $(".date").datetimepicker();
  });
</script>
```

Once these code elements are added, I end up with a pretty nice Date and Time editor to help edit a DateTime property. **Figure 10** shows this new editor in action.

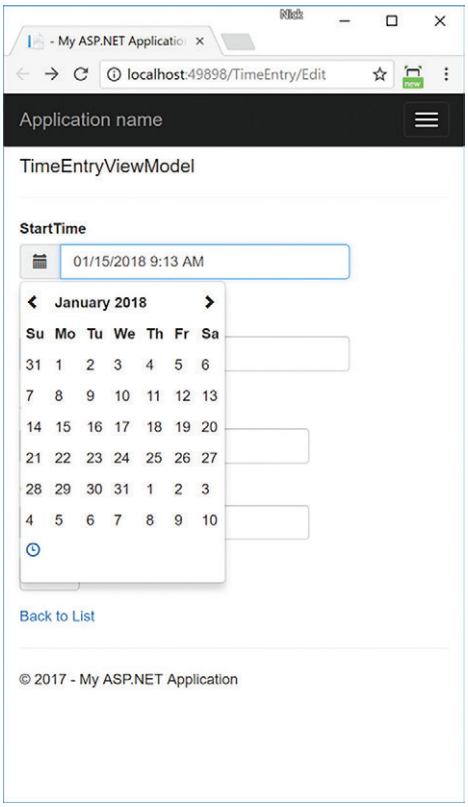


Figure 10 **An Activated Date Picker**

Wrapping Up

As you have seen, Razor can do a lot to simplify and streamline the creation of views in any client-side MVVM framework you want to use. You have the freedom to support application styles and conventions, and features such as scaffolding and EditorTemplates help ensure consistency across your application. They also enable built-in support for validations based on attributes added to your view model, making your application more secure.

Take a second look at ASP.NET MVC and you'll find many areas that are still relevant and useful, even as the landscape of Web applications continues to change. ■

Nick Harrison *is a software consultant living in Columbia, S.C., with his loving wife Tracy and daughter. He's been developing full stack using .NET to create business solutions since 2002. Contact him on Twitter: @Neh123us, where he also announces his blog posts, published works and speaking engagements.*

Thanks to the following technical expert for reviewing this article: *Lide Winburn (Softdocks Inc.)*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

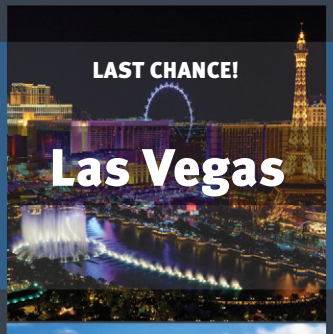
Visual Studio LIVE! 25 YEARS OF CODING INNOVATION

VISUAL STUDIO LIVE! (VSLive!™) is celebrating 25 years as one of the most respected, longest-standing, independent developer conferences, and we want you to be a part of it. Join us in 2018 for #VSLive25, as we highlight how far technology has come in 25 years, while looking toward the future with our unique brand of training on .NET, the Microsoft Platform and open source technologies in seven great cities across the US.

HELP US CELEBRATE #VSLIVE25. WHICH LOCATION WILL YOU ATTEND IN 2018?

MARCH 11 – 16
Bally's Hotel & Casino

Respect the Past. Code the Future.



APRIL 30 – MAY 4
Hyatt Regency Austin

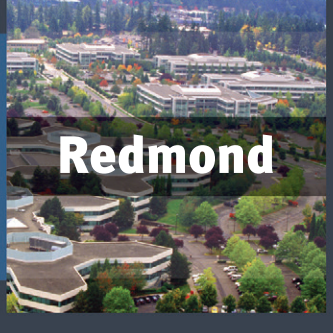
Code Like It's 2018!

JUNE 10 – 14
Hyatt Regency Cambridge

Developing Perspective.

AUGUST 13 – 17
Microsoft Headquarters

Yesterday's Knowledge; Tomorrow's Code!



SEPTEMBER 17 – 20
Renaissance Chicago

Look Back to Code Forward.

OCTOBER 7 – 11
Hilton San Diego Resort

Code Again for the First Time!

DECEMBER 2 – 7
Loews Royal Pacific Resort

Code Odyssey.

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

SUPPORTED BY

Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA YOUR GROWTH. OUR BUSINESS.

vslive.com

#VSLIVE25

Visual Studio LIVE!

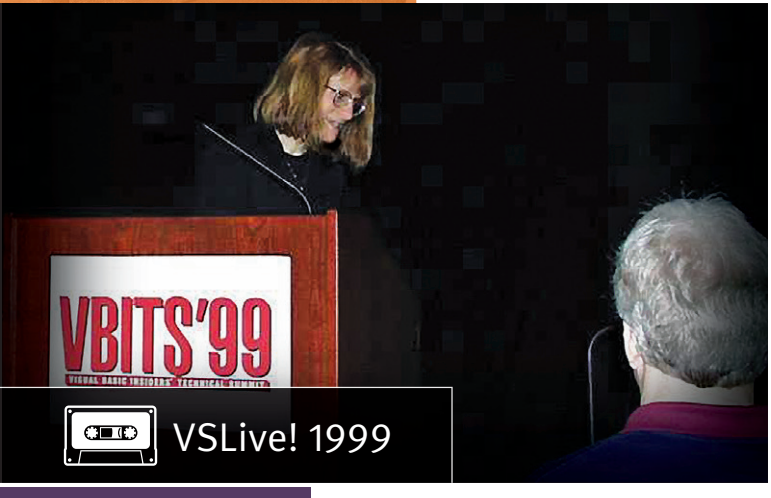
EXPERT SOLUTIONS FOR .NET DEVELOPERS

April 30 – May 4, 2018
Hyatt Regency Austin

Austin, TX

We're Gonna Code Like It's 2018!

Visual Studio LIVE! 25 YEARS OF CODING INNOVATION



VSLive! 1999

VSLive! 2017

INTENSE TRAINING FOR DEVELOPERS, ENGINEERS, PROGRAMMERS, ARCHITECTS AND MORE!

Development Topics Include:

- Visual Studio / .NET
- JavaScript / Angular
- Xamarin
- Software Practices
- Database and Analytics
- ASP.NET / Web Server
- ALM / DevOps
- Azure/Cloud
- UWP
- Hands-On Labs

REGISTER NOW

@ vslive.com/austinmsdn

Register to code with us today!

Register by March 28 and Save $200!

Use promo code AUONE

SILVER SPONSOR

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA — YOUR GROWTH, OUR BUSINESS.

AGENDA AT-A-GLANCE

Visual Studio LIVE! EXPERT SOLUTIONS FOR .NET DEVELOPERS **Austin, TX**

ALM / DevOps	Cloud Computing	Database and Analytics	Native Client	Software Practices	Visual Studio / .NET Framework	Web Client	Web Server

START TIME	END TIME						
Pre-Conference Workshops: Monday, April 30, 2018 *(Separate entry fee required)*							
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries					
9:00 AM	6:00 PM	**M01** Workshop: AI, Bots, ASP.NET with DevOps - *Brian Randell*		**M02** Workshop: Web Development in 2018 - *Chris Klug*		**M03** Workshop: Distributed Cross-Platform Application Architecture - *Rockford Lhotka and Jason Bock*	
6:45 PM	9:00 PM	Dine-A-Round					
START TIME	**END TIME**	**Day 1: Tuesday, May 1, 2018**					
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**T01** TypeScript: The Future of Front End Web Development. - *Ben Hoelting*	**T02** Flying High with Xamarin! - *Sam Basu*	**T03** Entity Framework Core 2 for Mere Mortals - *Philip Japikse*	**T04** - .NET, What Does That Mean? - *Chris Klug*		
9:30 AM	10:45 AM	**T05** Angular 101 - *Deborah Kurata*	**T06** Essential Tools for Xamarin Developers! - *Sam Basu*	**T07** Busy Developer's Guide to NoSQL - *Ted Neward*	**T08** - Get Started with Git - *Robert Green*		
11:00 AM	12:00 PM	**KEYNOTE: From Waterfall to Agile. Microsoft's Not-So-Easy Evolution into the World of DevOps** - *Abel Wang, Sr. Developer Technology Specialist, Microsoft*					
12:00 PM	1:00 PM	Lunch					
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors					
1:30 PM	2:45 PM	**T09** The Many Adventures of Routing - *Deborah Kurata*	**T10** Busy Developer's Guide to the Clouds - *Ted Neward*	**T11** Power BI for Developers - *Thomas LeBlanc*	**T12** Building Apps with Microsoft Graph and Visual Studio - *Robert Green*		
3:00 PM	4:15 PM	**T13** ASP.NET Core 2 For Mere Mortals - *Philip Japikse*	**T14** Cloud-Oriented Programming - *Vishwas Lele*	**T15** SQL Server 2017 Execution Plan Basics - *Thomas LeBlanc*	**T16** Advanced DevOps—Deep Dive into Feature Flags - *Abel Wang*		
4:15 PM	5:30 PM	Welcome Reception					
START TIME	**END TIME**	**Day 2: Wednesday, May 2, 2018**					
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**W01** Assembling the Web—A Tour of WebAssembly - *Jason Bock*	**W02** - .NET Core, Containers, and K8S in Azure - *Brian Randell*	**W03** New SQL Server Security Features for Developers - *Leonard Lobel*	**W04** Works On My Machine... Docker for Microsoft Developers - *Chris Klug*		
9:30 AM	10:45 AM	**W05** Tools for Modern Web Development - *Ben Hoelting*	**W06** Microservices with ACS (Managed Kubernetes) - *Vishwas Lele*	**W07** Introduction to Azure Cosmos DB - *Leonard Lobel*	**W08** Professional Scrum Development Using Visual Studio- *Richard Hundhausen*		
11:00 AM	12:00 PM	**GENERAL SESSION: To Be Announced**					
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch					
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:15pm (Must be present to win)					
1:30 PM	1:50 PM	**W09** Fast Focus: Living Happily Together: Visual Studio and Angular - *Brian Noyes*		**W10** Fast Focus: JSON Integration in SQL Server - *Leonard Lobel*		**W11** Fast Focus: Getting Git - *Jason Bock*	
2:00 PM	2:20 PM	**W12** Fast Focus: Getting Started with ASP.NET Core 2.0 Razor Pages - *Walt Ritscher*		**W13** Fast Focus: Xamarin.Forms Takes You Places! - *Sam Basu*		**W14** Fast Focus: Reuse Your .NET Code Across Platforms - *Rockford Lhotka*	
2:30 PM	3:45 PM	**W15** Introduction to HTML5 - *Paul Sheriff*	**W16** Practical Internet of Things for the Microsoft Developer - *Eric D. Boyd*	**W17** Writing Testable Code and Resolving Dependencies—DI Kills Two Birds with One Stone - *Miguel Castro*	**W18** DevOps for the SQL Server Database - *Brian Randell*		
4:00 PM	5:15 PM	**W19** Introduction to CSS 3 - *Paul Sheriff*	**W20** Lock the Doors, Secure the Valuables, and Set the Alarm - *Eric D. Boyd*	**W21** Dependencies Demystified - *Jason Bock*	**W22** Use Visual Studio to Scale Agile in Your Enterprise - *Richard Hundhausen*		
6:45 PM	9:00 PM	VSLive! Rollin' On the River Bat Cruise					
START TIME	**END TIME**	**Day 3: Thursday, May 3, 2018**					
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**TH01** Securing Angular Apps - *Brian Noyes*	**TH02** Use UWP to Modernize Your Existing WinForms and WPF Applications - *Walt Ritscher*	**TH03** Coaching Skills for Scrum Masters & The Self-Organizing Team - *Benjamin Day*	**TH04** Improving Code Quality with Roslyn Code Analyzers - *Jim Wooley*		
9:30 AM	10:45 AM	**TH05** Securing Web Apps and APIs with IdentityServer - *Brian Noyes*	**TH06** A Dozen Ways to Mess Up Your Transition From Windows Forms to XAML - *Billy Hollis*	**TH07** Demystifying Microservice Architecture - *Miguel Castro*	**TH08** Getting to the Core of the .NET Standard - *Adam Tuliper*		
11:00 AM	12:15 PM	**TH09** ASP Core Middleware & Filters—Why and How - *Miguel Castro*	**TH10** Building Cross Platform Business Apps with CSLA .NET - *Rockford Lhotka*	**TH11** Unit Testing & Test-Driven Development (TDD) for Mere Mortals - *Benjamin Day*	**TH12** C# 7.x Like a Boss! - *Adam Tuliper*		
12:15 PM	1:30 PM	Lunch					
1:30 PM	2:45 PM	**TH13** Getting Pushy with SignalR and Reactive Extensions - *Jim Wooley*	**TH14** UX Design for Developers: Basics of Principles and Process - *Billy Hollis*	**TH15** Modernize Your Legacy Applications with Continuous Integration and Continuous Delivery - *David Walker*	**TH16** Scale Agile without Friction: DevOps to the Rescue - *Clementino de Mendonca*		
3:00 PM	4:15 PM	**TH17** Migrating to ASP.NET Core—A True Story - *Adam Tuliper*	**TH18** Add a Conversational User Interface to Your App with the Microsoft Bot Framework - *Walt Ritscher*	**TH19** Hard-Core Integration Intoxication with .NET Core - *David Walker*	**TH20** VSTS as a Knowledge Management Hub: Secure the Future of Your IP - *Clementino de Mendonca*		
START TIME	**END TIME**	**Post-Conference Full Day Hands-On Labs: Friday, May 4, 2018** *(Separate entry fee required)*					
7:30 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries					
8:00 AM	5:00 PM	**HOL01** Full Day Hands-On Lab: Develop an ASP.NET Core2 and EF Core 2 App in a Day - *Philip Japikse*		**HOL02** Full Day Hands-On Lab: Developer Dive into SQL Server - *Leonard Lobel*			

Speakers and sessions subject to change

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

vslive.com/austinmsdn

Developing Perspective

Visual Studio Live! (VSLive!™) returns to Boston, June 10 – 14, 2018, with 5 days of practical, unbiased, Developer training, including NEW intense hands-on labs. Join us as we dig into the latest features of Visual Studio 2017, ASP.NET Core, Angular, Xamarin, UWP and more. Code with industry experts AND Microsoft insiders while developing perspective on the Microsoft platform. Plus, help us celebrate 25 years of coding innovation as we take a fun look back at technology and training since 1993. Come experience the education, knowledge-share and networking at #VSLive25.

Visual Studio LIVE! 25 YEARS OF CODING INNOVATION

SILVER SPONSOR

SUPPORTED BY

Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY

1105 MEDIA
YOUR GROWTH. OUR BUSINESS.

DEVELOPMENT TOPICS INCLUDE:

- VS2017/.NET
- JavaScript / Angular
- Xamarin
- Software Practices
- Database & Analytics
- ASP.NET / Web Server
- ALM / DevOps
- Azure / Cloud
- UWP (Windows)
- Hands-On Labs (NEW!)

"I have attended 12+ Visual Studio Live events, starting with the early days of VBITS and C# Live. I return every year because no one reflects and enhances what is currently happening in and with Microsoft's developer world like VSLive! I honestly trust and enjoy all the speakers I look to them as both mentors and old friends!"

– John Kasarda, Accuquote

"Last year, I got so much out of the SOLID principle sessions – it gave me great perspective on the subject. This year, I loved the HoloLens introduction. I work for an aerospace company, and I can see that the technology will be utilized for assembly, 3D design, 3D inspection and many other applications in the near future!"

– Mani Nazari, IT Factory Automation Systems

@ vslive.com/bostonmsdn

Register to code with us today!

Register Now and Save $300!

Use promo code VSLBO1

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

vslive.com/bostonmsdn

Enterprise Data Integration Patterns with Azure Service Bus

Stefano Tempesta

In the age of Big Data and machine learning, acquiring and managing information is critical, but doing so can be a complex undertaking because data is often more complex than you might realize. When considering how to make your application communicate with other IT systems, an effective design for data exchange is key to success. This article provides an overview and implementation of data integration processes between applications using Azure Service Bus.

Data Integration Design Patterns

Data flows in multiple directions across networks, applications, and volatile or persistent repositories. It may be exchanged record by record or in batches, through systems that talk to each other in real time or via scheduled synchronization jobs. Despite the variety of data integration "journeys," it's possible to identify common design patterns for how to address them in an enterprise context, where requirements include high availability, guaranteed delivery and security. Design patterns in software engineering are the most logical and proven sequences of steps to resolve a task. The four most common design patterns for data integration are broadcast, aggregation, bidirectional synchronization and correlation.

This article discusses:

- Data integration design patterns
- Publish/Subscribe pattern
- Service Bus Relay

Technologies discussed:

Azure Service Bus

Code download available at:

bit.ly/2s0FWow

In this article, I introduce each of these data integration design patterns and describe their application in the context of Azure Service Bus. An approach to data integration that leverages an enterprise service bus (ESB) facilitates the implementation of these patterns in a very effective way, by simply defining source and target systems, frequency of communication, and the data format for input and output. Along with the description of each pattern, I include code samples to illustrate communication with Azure Service Bus.

Context and Requirements

I'll start my journey to data integration by defining the enterprise context. In this case, let's assume I'm building an e-commerce platform, with the typical requirements for an online catalog and shopping cart, and that I'll publish my application in Azure. The e-commerce application is part of a larger ecosystem of IT systems, some exposed on a public cloud, some still hosted on a private datacenter. Thus, I'm operating in a truly hybrid context. The data integration requirements include the following capabilities:

1. Transfer product sales information from the e-commerce application to an invoicing system and to social media platforms.
2. Receive product availability information from an enterprise resource planning (ERP) application, and product descriptions from third-party vendor systems.
3. Add location tracking to shipped parcels.
4. Share customer data with partner organizations for co-marketing activities.

All of these integration requirements can be addressed using the design patterns mentioned earlier. Let's take a closer look.

The Broadcast Pattern

For the first capability, I need to extract data concerning product sales from the e-commerce application and transfer it to multiple

target systems—a financial system for issuing invoices and one or more social media platforms for promotion. The data flow direction is one-directional, from the application to external systems. I'm basically broadcasting the information to the world outside.

The broadcast integration pattern describes how to transfer data from an application to multiple target systems in a continuous real-time or near-real-time flow. This process is expected to be transactional: If a transaction completes successfully, data is committed at destination. If the transaction fails, the data transfer is aborted. It's immediately obvious that this broadcast integration channel must be highly available and reliable, in order to avoid losing critical data in transit. Adopting an ESB as the mechanism for queuing data packets and guaranteeing delivery at destination becomes crucial.

Implementing the broadcast pattern closely resembles implementing the publish/subscribe pattern in Azure Service Bus, based on topics and subscriptions (bit.ly/2oOmTtM). Topics represent queues of messages to which recipient applications (subscribers) subscribe to receive updates when a message is posted. My e-commerce application publishes a message into a topic. The ESB acts as a message broker and guarantees delivery of the message at destination by "pushing" the message to the destination, which consists only of subscribed recipients.

Broadcasting a data packet from the e-commerce application essentially means publishing a message in a topic, and having a target application listening on a specific subscription. The broadcast pattern adds a transactional attribute to the data flow, with the possibility to cancel the transaction in case of delivery failure. As transactions cross system boundaries, they benefit from a "state machine" that retains a snapshot of the brokered message being transferred, before it's read by all subscribed applications. If any of the subscribers fails to retrieve the message, the entire transaction is aborted, to ensure consistency across all systems involved.

The following code broadcasts a message to an Azure Service Bus topic and implements a state machine (bit.ly/29tKRT3) for tracking delivery of a message:

```
public class Broadcast
{
  public async Task Execute(Entity entity)
  {
    var client = TopicClient.CreateFromConnectionString(connectionString, topicName);
    var message = new BrokeredMessage(JsonConvert.SerializeObject(entity));

    await client.SendAsync(message);
  }
```

In case of error on delivery, the state machine moves the message to the "dead letter" queue in Azure Service Bus. The message at that point is no longer valid for data transfer and won't be processed further.

Sending a message to a topic in Azure Service Bus requires a TopicClient connection and a BrokeredMessage to wrap the original entity and send it, asynchronously, to the bus. All required objects for connecting to Azure Service Bus are distributed in the WindowsAzure.ServiceBus NuGet package and are available in the Microsoft.ServiceBus.Messaging namespace.

The state machine is a singleton asynchronous dictionary containing transaction counters by topic. The dictionary keeps count of the number of active transactions—subscribers—that are waiting for a message on a specific topic from the Service Bus. The dictionary is thread-safe to allow for concurrent requests:

```
  private static StateMachine _instance;
  public static StateMachine Current => _instance ?? (_instance = new StateMachine());

  protected ConcurrentDictionary<string, int> transactions =
    new ConcurrentDictionary<string, int>();
```

As shown in **Figure 1**, a subscriber application reads a message from the Service Bus topic by beginning a new transaction (using the BeginTransactionAsync method) for a specific topic on the state machine, and then handles the OnMessage event to obtain a copy of the entity. The entity is then processed internally; for example, it may be persisted by the recipient system. In case of error, the transaction is canceled.

Completing or aborting the transaction is managed by the state machine using either of two methods—SuccessAsync or CancelAsync. SuccessAsync invokes CompleteAsync on the brokered message, which indicates that the message should be marked as processed and eventually deleted from the topic. This takes place only when all concurrent active transactions are completed:

```
public async Task<bool> SuccessAsync(BrokeredMessage message, string topicName)
{
  bool done = await EndTransactionAsync(topicName);
  int count = Current.transactions[topicName];

  // All concurrent transactions are done
  if (done && count == 0)
  {
     await message.CompleteAsync();
  }

  return done;
}
```

CancelAsync, in contrast, aborts the message broadcast by resetting the transaction counter for a topic. By calling the DeadLetterAsync method, the brokered message is then moved to the "dead letter" queue, where unsuccessfully processed messages are stored:

```
public async Task<bool> CancelAsync(BrokeredMessage message, string topicName)
{
  // Cancel the message broadcast -> Remove all concurrent transactions
  int count = Current.transactions[topicName];
  bool done = Current.transactions.TryUpdate(topicName, 0, count);

  if (done)
  {
    await message.DeadLetterAsync();
  }

  return done;
}
```

Figure 1 **Reading a Message from the Service Bus Topic**

```
public async Task ReadMessageAsync()
{
  await StateMachine.Current.BeginTransactionAsync(topicName);

  var client = SubscriptionClient.CreateFromConnectionString(
    connectionString, topicName,
    subscriptionName);
  client.OnMessageAsync(async message =>
  {
    var entity = JsonConvert.DeserializeObject(message.GetBody<string>());

    try
    {
      Save(entity);
      await StateMachine.Current.SuccessAsync(message , topicName);
    }
    catch
    {
      await StateMachine.Current.CancelAsync(message , topicName);
    }
  });
}
```

Figure 2 **Reading Messages from a Topic**

```
public class Aggregation
{
  public void Execute()
  {
    var client = SubscriptionClient.CreateFromConnectionString(
      connectionString, topicName, subscriptionName);
    client.OnMessage(message => {
      ProductEntity product =
        EntityMap.Instance.MapToEntity<ProductEntity>(message);

      // Persist the product
      var exists = Find(product.Id) != null;
      if (exists)
        Update(product);
      else
        Create(product);
      });
  }
```

The Aggregation Pattern

The second requirement for my e-commerce platform is to share information about products from external systems and consolidate it into the Web portal. The direction of the data flow in this case is opposite to that of the broadcast pattern. The requirement now is to aggregate data from various sources into a single place. A simple approach would be to import data directly, point to point, from each source into the e-commerce application. But this isn't a scalable solution, as it implies building a different connection for each external system that's sending data to the target repository. Instead, by aggregating data in one process through an ESB, I eliminate the need for multiple one-way integrations, and mitigate concerns about data accuracy and consistency as data is processed in an atomic transaction.

What arises, though, is the challenge of merging data into a single entity without duplicating information or, worse, corrupting it. It's often necessary to implement custom merge logic as part of the integration process, in order to track aggregated records sourced from different systems and store them in one or more entities in the application. I need a mapping table to associate record IDs in the different source databases with the entity ID in the target database. This mapping table is typically persisted in a high-transaction database, and updated by the data merge custom logic during the aggregation process.

As with the Broadcast pattern, the implementation of this integration pattern also reflects publishing/subscribing to an Azure Service Bus topic, with the difference that, in this case, my e-commerce application is the target system receiving data (subscribing to a topic) from other source systems via the ESB. The overall solution also needs to use some data merge logic and data mapping to track source record IDs and the entity ID at destination.

As you can see in **Figure 2**, reading messages from a topic consists of creating a subscription connection (SubscriptionClient), and handling the OnMessage event that's raised by the subscription client when a new message is available in the topic. The received message contains an object sent by an external application, say a vendor sending product details. This object is then mapped to an entity in my system, using the entity map, and if the entity already exists in my database, I update it; otherwise I create a new one.

The mapping process, implemented in the EntityMap class, consists of two important steps:

1. It creates a map between the object in the external system and the entity in my database. This map identifies the external object by pairing the system name (such as "ERP," "Vendor1," "Vendor2") with the primary key. The mapped entity is identified by its type in my application (Product, Customer, Order) and its ID.
2. It builds an entity record using the properties of the external object. This is the custom merge logic, which can be as easy as using a library like AutoMapper (automapper.org) to map objects.

As shown in **Figure 3**, the object-entity map is a dictionary that associates a system name-primary key pair to an entity type-entity id counterpart. An object in an external system is identified uniquely by the combination of system name and primary key, whereas an entity in my application is identified by the combination entity type and entity id.

The map is populated by retrieving the system name and primary key from the properties of the brokered message, and then building an entity with AutoMapper, as shown in **Figure 4**.

A brokered message can be enriched with any additional property, which the publisher application should set before sending the message to the ESB:

```
public async Task SendMessageToServiceBus(object obj)
{
  var client = TopicClient.CreateFromConnectionString(connectionString, topicName);
  var message = new BrokeredMessage(JsonConvert.SerializeObject(obj));

  message.Properties["SystemName"] = "Publisher System Name";
  message.Properties["PrimaryKey"] = "Object Primary Key";

  await client.SendAsync(message);
}
```

The Bidirectional Synchronization Pattern

Let's look now at the third requirement: adding location tracking to shipped parcels. In more generic terms, I want to augment the attribute of an entity with additional attributes that are provided by a more specialized external application. For this reason, this pattern is sometimes referred to as the Augmentation pattern.

Systems involved in a bidirectional synchronization process with a third-party line-of-business application can extend their functionality beyond their boundaries. As an example, Dynamics 365 is a customer relationship management platform (and much more) that integrates natively with SharePoint for enterprise document management. Dynamics 365 still remains the master data source for all records, but documents are stored in SharePoint as an extension of entities in the CRM system. Basically, once two systems are in bidirectional sync, they behave as one system while still retaining

Figure 3 **The Object-Entity Map**

System Name	Primary Key	Entity Type	Entity Id
ERP	ABC012345	Product	FAE04EC0-301F-11D3-BF4B-00C04F79EFBC
Vendor 1	1000	Product	FAE04EC0-301F-11D3-BF4B-00C04F79EFBC
ERP	ABD987655	Product	2110F684-C277-47E7-B8B9-6F17A579D0CE
Vendor 2	1001	Product	2110F684-C277-47E7-B8B9-6F17A579D0CE

their own dataset and, obviously, functionality. Data is distributed across the two systems, but is seen as a single entity through this seamless integration.

Most of the time, and as with Dynamics 365 and SharePoint, this direct and real-time synchronization is not implemented with a service bus. A point-to-point connector typically exists that knows how to talk to either system with minimum configuration. But what happens when a native connector between applications doesn't exist? Besides building a custom connector, which may not be a trivial task at all (just think of the effort necessary to understand authentication and API calls, as well as provide the high availability and guarantee of delivery typical of an ESB), what you can do is implement a relay.

Azure Service Bus Relay (bit.ly/2BNTBih) is an extension of Service Bus that facilitates communication among systems in a hybrid configuration by enabling a secure connection to systems not accessible from the public cloud. Let's assume that the GIS system I'm connecting to is hosted within a private corporate network. Data transfer initiated by the on-premises service connects to the relay through an outbound port, creating a bidirectional socket for communication tied to a particular rendezvous address. The e-commerce application, hosted in Azure, can then communicate with the GIS service behind the firewall by sending messages to the relay service targeting the rendezvous address. The relay service then "relays" data to the on-premises service through a dedicated bidirectional socket. The e-commerce application doesn't need (and can't establish) a direct connection to the on-premises GIS service, and it doesn't even need to know where the service resides, as the entire communication is only to the relay.

Just to mention another advantage of implementing a solution based on Azure Relay, the relay capabilities differ from network-level integration technologies such as VPNs because they can be scoped to a single application endpoint on a single machine. VPN technology, in contrast, is far more intrusive as it relies on altering the network environment.

The NuGet package Microsoft.Azure.Relay contains the namesake namespace with the relevant objects for managing communications with Azure Service Bus Relay. But let's start by defining the GIS server first, which consists of:

- **GisObject:** An object for storing geo-coordinates (latitude and longitude) and a fully resolved location address.
- **GisProcess:** A process that maintains a bidirectional connection to the GIS server, via Azure Relay, and transfers instances of GisObject between the GIS server and the e-commerce application.
- **ServerListener:** An extension to the GIS server that acts as a bridge between the GIS server itself and Azure Relay.

The bidirectional connection is maintained in multiple steps:

First, I create a hybrid connection client to Azure Relay, using an access security key obtained from the Azure portal:

```
var tokenProvider =
  TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key);
var client = new HybridConnectionClient(
  new Uri($"sb://{relayNamespace}/{connectionName}"), tokenProvider);
var relayConnection = await client.CreateConnectionAsync();
```

Once the connection is established, I run two asynchronous tasks in sequence: The first task sends an instance of GisObject with latitude and longitude coordinates to the relay; the second task reads this object back from the relay. At the conclusion of both tasks, the hybrid connection is closed:

```
await new Task(
  () => SendToRelay(relayConnection, gisObject)
  .ContinueWith(async (t) =>
  {
    GisObject resolved = await ReadFromRelay(relayConnection);
    ShowAddress(resolved);
  })
  .ContinueWith(async (t) =>
    await relayConnection.CloseAsync(CancellationToken.None))
  .Start());
```

Sending an object to Azure Relay is a matter of writing a message to a stream. You can serialize the object in several formats; typically this is done in JSON:

```
private async Task SendToRelay(HybridConnectionStream relayConnection,
  GisObject gisObject)
{
  // Write the GIS object to the hybrid connection
  var writer = new StreamWriter(relayConnection) { AutoFlush = true };
  string message = JsonConvert.SerializeObject(gisObject);
  await writer.WriteAsync(message);
}
```

Similarly, reading an object from Azure Relay involves reading from a stream and de-serializing the obtained string of characters into the original object type:

```
private async Task<GisObject> ReadFromRelay(HybridConnectionStream relayConnection)
{
  // Read the GIS object from the hybrid connection
  var reader = new StreamReader(relayConnection);
  string message = await reader.ReadToEndAsync();
  GisObject gisObject = JsonConvert.DeserializeObject<GisObject>(message);
  return gisObject;
}
```

The GIS server also listens to traffic passing through the relay, reads incoming messages containing an instance of the serialized GisObject, and resolves a location address by invoking a specific GIS service (not described in the proposed solution):

```
private async Task Listen(HybridConnectionListener listener,
  CancellationTokenSource cts)
{
  // Accept the next available, pending connection request
  HybridConnectionStream relayConnection;
  do
  {
    relayConnection = await listener.AcceptConnectionAsync();
    if (relayConnection != null)
    {
      ProcessMessage(relayConnection, cts);
    }
  } while (relayConnection != null);
}
```

Figure 4 **Populating the Map**

```
public T MapToEntity<T>(BrokeredMessage message) where T : Entity, new()
{
  string systemName = message.Properties["SystemName"] as string;
  string primaryKey = message.Properties["PrimaryKey"] as string;

  T entity = BuildEntity<T>(message);
  map.Add((systemName, primaryKey), (entity.GetType(), entity.Id));

  return entity;
}

private T BuildEntity<T>(BrokeredMessage message) where T : Entity, new()
{
  var source = JsonConvert.DeserializeObject(message.GetBody<string>());
  T entity = Mapper.Map<T>(source);

  return entity;
}
```

The connection is a fully bidirectional stream. As **Figure 5** shows, I add a stream reader and a stream writer to it, which allows me to read the JSON-serialized GIS object and to write it back to the relay after resolving the provided geo-coordinates into a location address.

Figure 5 **Reading and Writing the JSON-Serialized GIS Object**

```
private async void ProcessMessage(HybridConnectionStream relayConnection,
  CancellationTokenSource cts)
{
  // Bidirectional streams for reading and writing to the relay
  var reader = new StreamReader(relayConnection);
  var writer = new StreamWriter(relayConnection) { AutoFlush = true };
  while (!cts.IsCancellationRequested)
  {
    // Read a message in input from the relay
    var message = await reader.ReadToEndAsync();

    // Resolve address by invoking a service on the GIS server
    GisObject gisObject =
      JsonConvert.DeserializeObject<GisObject>(message);
    await new GisServer().ResolveAddressAsync(gisObject);

    // Write the message back to the relay
    message = JsonConvert.SerializeObject(gisObject);
    await writer.WriteLineAsync(message);
  }

  await relayConnection.CloseAsync(cts.Token);
}
```

Figure 6 **The Correlation Class**

```
public class Correlation
{
  private async Task Execute(CustomerEntity customer)
  {
    // Map the Customer entity in the e-commerce application (source)
    // to Customer record in the partner application (destination)
    CustomerRecord customerRecord = PrepareCustomerRecord(customer);

    // Create a connection to an Azure Service Bus Topic
    // Serialize the customer record and send the message to the Topic
    var client = TopicClient.CreateFromConnectionString(
      connectionString, topicName);
    var message = new BrokeredMessage(
      JsonConvert.SerializeObject(customerRecord));

    // Register the customer record with the Correlation Service
    // and obtain a Correlation ID
    message.Properties["CorrelationId}"] =
      new CorrelationService().RegisterCustomer(customerRecord, subscriptionName);

    await client.SendAsync(message);
  }
```

Figure 7 **The Partner Class**

```
class Partner
{
  public void ReceiveCustomerRecord()
  {
    var client = SubscriptionClient.CreateFromConnectionString(
      connectionString, topicName, subscriptionName);
    client.OnMessageAsync(async message =>
    {
      CustomerRecord customerRecord =
        JsonConvert.DeserializeObject<CustomerRecord>(message.GetBody<string>());
      Guid correlationId = (Guid)message.Properties["CorrelationId"];

      if (CustomerRecordExists(correlationId))
      {
        await SaveAsync(customerRecord);
      }
    });
  }
```

The Correlation Pattern

There is one more requirement to meet: I need to share customer data with partner organizations. But I don't want to disclose information that partners aren't authorized to access. I need to implement a way to sync data among systems only if they're correlated with each other.

The Correlation pattern, focuses on the intersection of two datasets and performs a synchronization of that scoped dataset only if a record exists in both systems. While the relay communication with the GIS server would create a new record if the object couldn't be found in the system, implementing data integration based on the correlation pattern strictly requires that correlated records exist in both systems for the sync to happen. This applies perfectly to my case where I want to share data with marketing partners, but only if they already have this information in their own system. But there's a clear challenge here—how can I identify related records that represent the same entity (customer) across systems? This condition defines whether customer records can be synced with external partners.

As shown in **Figure 6**, the data correlation workflow in the e-commerce application sends a customer record with some marketing information to an Azure Service Bus topic. The customer record is an aggregation of data from multiple entities. It's not advisable to use the same object (a database entity) as a data transfer object (DTO), as this would create a dependency between the service and the data model in the source application. The brokered message is also decorated with a correlation ID that identifies the specific record in a topic subscription; this correlation ID will be useful later in the partner application for verifying whether a customer record already exists.

The correlation service simply exposes methods for matching customer records on a specific subscription, and registering a new customer and returning its correlation ID:

```
public class CorrelationService
{
  public Guid RegisterCustomer(CustomerRecord record, string subscription)
  {
    return store.ContainsKey((record, subscription)) ?
      GetCustomerCorrelationId(record, subscription) :
      AddCustomer(record, subscription);
  }

  public bool CustomerExists(Guid correlationId)
  {
    return store.ContainsValue(correlationId);
  }
```

Partner applications subscribe to that topic and retrieve the customer record and the correlation ID, as shown in **Figure 7**. If the customer record exists in their system, it can be saved eventually.

The entire solution is available free to download from my GitHub repository at bit.ly/2s0FWow. ■

Stefano Tempesta *is a Microsoft MVP and MCT, and chapter leader of CRMUG Switzerland. A regular speaker at international conferences, including Microsoft Ignite, Tech Summit and Developer Week, Stefano's interests extend to Office & Dynamics 365, Blockchain and AI-related technologies.*

Thanks to the following Microsoft technical expert for reviewing this article: *Massimo Bonanni*

Secure Your Sensitive Business Information with Azure Key Vault

Srikantan Sankaran

Azure Key Vault is a cloud-based service that lets organizations securely store sensitive business information. It lets you perform cryptographic operations on the data, and provides a framework for implementing policies to regulate access to applications, as well as an API model for applications to work with the keys, secrets and certificates stored in it. The SDKs that Azure Key Vault provides support a variety of device platforms and programming languages, allowing you to choose your preferred language and to deploy these applications to Azure App Service as managed Web applications. To expose these business applications securely to users both within an organization and outside, Azure Active Directory (Azure AD) and Azure Active Directory B2C (Azure AD B2C) provide turnkey implementations to enable authentication and authorization to the applications with minimal or no custom code. In this article I'll present a solution that demonstrates how Azure Key Vault can bring enhanced security to your organization.

The Managed Service Identity feature discussed in the article is in public preview. All information is subject to change.

This article discusses:

- Using Azure Key Vault to store sensitive business data as secrets
- Registering ASP.NET 2.0 Core applications with Azure AD and Azure AD B2C to provide turnkey authentication and authorization features
- Generating and reading QR codes in both Web and native apps

Technologies discussed:

Azure Key Vault, Azure App Service, Azure Active Directory, Azure Active Directory B2C, ASP.NET 2.0 Core, Azure SQL Database

Code download available at:

bit.ly/2DRvwdh

Use Case Scenario

A central agency is tasked with implementing a solution to issue, track and manage insurance policies for vehicles. The agency generates unique document serial numbers on receipt of orders and payment from insurance companies. These companies, either directly or through their brokers, assign insurance policies to the document serial numbers as they're sold to motorists. Document serial numbers are unique across all insurance companies.

The goal of the solution is to track the lifecycle of the document serial number. When created, a document serial number contains only its number and the name of the insurance company to which it's sold. Further into the business process, additional information, such as the vehicle registration, policy document number, identity of the customer and validity period of the insurance policy, will be added. All versions of this record must be tracked, including any changes made, the date and time of the changes, and the identity of the application that made the change.

Customers should be able to access the policy electronically and download the information securely for verification and easy reference.

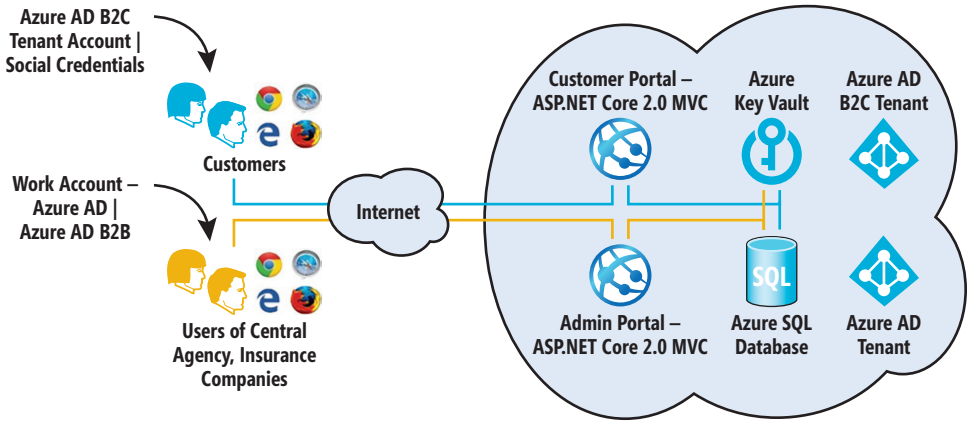


Figure 1 **Solution Architecture**

Architecture of the Solution

The solution uses Azure Key Vault to store the document serial number, along with the properties of the associated insurance policy, as a secret. For additional security, the data that's stored as a secret is encrypted beforehand using asymmetric keys generated in Azure Key Vault. While only the bare minimum data required to secure and validate each policy gets captured in the secret, additional supporting information is stored in an Azure SQL Database. The database also implements constraints on the data, to ensure, for example, that a registered vehicle has a single active policy number, that the same policy number hasn't been used for multiple records, and so forth. **Figure 1** represents the architecture used in the Solution.

> Separate service principals in Azure AD are created for the admin and customer portals, and separate policies are set that lock down their access to operations in Azure Key Vault.

I've implemented two portal applications in this solution, one that's used by the central agency and insurance companies, and the other by customers who buy insurance policies and by regulatory authorities who need to ascertain the validity of the policies.

The admin portal and the customer portal are ASP.NET 2.0 Core MVC applications that use Entity Framework to store policy data in an Azure SQL Database, after it has first been stored in Azure Key Vault. The .NET SDK for Azure Key Vault is used to perform the cryptographic operations on the data, such as the creation of secrets and their versions and the encryption and decryption of the secrets using keys. Users of the admin portal are authenticated with Azure AD, while customers, who are external users, use Azure AD B2C to self-register and sign in to the customer portal.

Separate service principals in Azure AD are created for the admin and customer portals, and separate policies are set that lock down their access to operations in Azure Key Vault. The admin portal policy permits the creation of keys and secrets, as well as the performance of operations like the encryption and decryption of data. The customer portal, in contrast, is assigned a policy that permits only the "get" operation on a secret and the "decrypt" operation on the secret retrieved. This ensures that individual applications don't have more access than needed to Azure Key Vault.

The policy data stored as a secret in Azure Key Vault is first encrypted, for additional security. Every time the secret is updated, a new version gets created, and previous versions of the data are preserved. An audit trail is also maintained for all operations performed in the Key Vault, which is archived to meet statutory compliance requirements.

The attribute bundle of the secrets stored in Azure Key Vault captures the policy start and end dates, which are used to ascertain the validity of the policy. Tags and content-type parameters of secrets are used to store additional information pertaining to the insurance policy.

The following code snippet shows how the attributes, tags and content types are added to the policy data stored as a secret:

```
SecretAttributes attribs = new SecretAttributes
  {
    Enabled = true,
    Expires = DateTime.UtcNow.AddYears(1),
    NotBefore = DateTime.UtcNow.AddDays(1)
  };

IDictionary<string, string> alltags = new Dictionary<string, string>();
alltags.Add("InsuranceCompany", policydata.Inscompany);
string contentType = "DigitalInsurance";
SecretBundle bundle= await _keyVaultClient.SetSecretAsync(keyVaultUri,
policydata.Uidname,encrypteddata,alltags,contentType,attribs);
```

Implementing the Use Case Scenario

Let's take a closer look at the use case scenario that's implemented by the solution. Here are the basic steps:

Purchase of Unique Codes by Insurance Companies: On receipt of orders from the insurance companies, the central agency uses the admin portal to generate an inventory of document serial numbers and store them as secrets in Azure Key Vault. The admin portal creates the first version of a secret in Azure Key Vault and then creates a record in Azure SQL Database.

Policy Generation: When a customer purchases a vehicle policy, an unassigned secret from the previous step is chosen and additional information, such as the vehicle registration number, the identity of the customer, the policy document number generated and the validity period of the policy, is added. A new version of the original secret containing this additional information is

created in the process, and the corresponding record in Azure SQL Database is also updated.

Activation of Policy by Customer: Once all the details of a policy are captured in the secret, a notification is sent to the customer (outside the scope of this article) with instructions for activating the policy. Users can self-register on the customer portal either using their social credentials or the credentials stored in Azure AD B2C. When the customer is signed into the portal, the policy details are displayed along with an option to activate it. On activation, the user downloads a QR code from the portal and affixes its image to the insured vehicle.

Policy Validation: Customers or regulatory authorities can validate the policy anytime using a native Xamarin app that reads the QR code on the vehicle and displays the policy details in it, to be tallied with that on the vehicle. This validation doesn't require an Internet connection and can be done offline. When connected to the Internet, additional validation can be performed. The native app invokes a REST API that's exposed in the customer portal MVC application, passing the data from the QR code. The API first matches this data with the data in Azure SQL Database, and additionally with the data stored in the secret in Azure Key Vault.

Technical Aspects of the Solution

Now let's delve into the source code and automation scripts used in the solution. Keep in mind that the code and scripts shared in this article are by no means intended to be a complete solution, nor do they necessarily handle all validations, exceptions or best practices required in a production-ready application. They are meant rather to illustrate specific aspects of a technology area or to provide guidance toward developing a full-fledged solution.

Figure 2 **Dependency Framework in ASP.NET Core 2.0 MVC Application**

```
// This method gets called by the runtime. Use this method to add
services to the container.
public void ConfigureServices(IServiceCollection services)
  {
    //Adding the Azure AD integration for User authentication
    services.AddAuthentication(sharedOptions =>
    {
      sharedOptions.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
      sharedOptions.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddAzureAd(options => Configuration.Bind("AzureAd", options))
    .AddCookie();
    services.AddMvc();

// Add the Key Vault Service Client Connection to the Context Object
    AKVServiceClient servClient =
      new AKVServiceClient(Configuration["AzureKeyVault:ClientIdWeb"],
      Configuration["AzureKeyVault:AuthCertThumbprint"],
      Configuration["AzureKeyVault:VaultName"],
      Configuration["AzureKeyVault:KeyName"]);
    services.AddSingleton<AKVServiceClient>(servClient);

// Get the Connection string to Azure SQL Database
// from the secret in Azure Key Vault
    string connection = servClient.GetDbConnectionString();
// Add the Azure SQL Database Connection to the Context Object
    services.AddDbContext<ContosoinsauthdbContext>(options =>
      options.UseSqlServer(connection));

    services.AddOptions();
  }
```

Create and Configure Azure Key Vault The PowerShell script files PrepareContosoAKV.ps1 and PrepareContosousersAKV.ps1, included with the accompanying download, are used to provision and configure the key vault used in this article. Here's what they accomplish:

- Creation of self-signed certificates (to be used only in dev scenarios) for the admin and custmer portal ASP.NET MVC applications, which are used when creating the service principals in Azure AD.
- Creation of a service principal in Azure AD that's assigned to the admin portal. The access policy that's set for this service principal permits creation and update of keys and secrets, and the performance of operations like encryption and decryption:

```
# Specify privileges to the vault for the Admin Portal application
Set-AzureRmKeyVaultAccessPolicy -VaultName $vaultName `
  -ObjectId $servicePrincipal.Id `
  -PermissionsToKeys all `
  -PermissionsToSecrets all
```

- Creation of a service principal in Azure AD that's assigned to the customer portal. The access policy that's set for this service principal permits Get operations on keys and secrets, and the decryption of data:

```
# Specify privileges to the vault for the Customer Portal application
Set-AzureRmKeyVaultAccessPolicy -VaultName $vaultName `
  -ObjectId $servicePrincipal.Id `
  -PermissionsToKeys get,list,decrypt `
  -PermissionsToSecrets get,list
```

- Note that there's an alternative to creating these service principals with PowerShell, which is to use the Managed Service Identity feature in Azure App Service. This is actually recommended. Refer to the guidance at bit.ly/2BgB6mu for more details.
- Creation of a key used for the encryption and decryption of a secret.
- Creation of a secret that stores the connection string to the Azure SQL Database. (This can be done directly on the Azure Portal, as can the other steps.)

For simplicity, this solution uses a single key vault, and a single set of keys for all insurance companies and brokers to encrypt and decrypt the data. In the real world, for added isolation and security, separate Azure Key Vault instances should be created for each insurance company, and grouped by region, for example, or any other criteria. This ensures that the data maintained by one insurance company can't be read by others because they wouldn't share the same encryption key.

Keep in mind that secrets stored in Azure Key Vault must be no more than 25KB in size. Hence, to avoid bloat, only certain properties of the policy data are stored in it, such as ID (document serial number), secret name, user ID, policy number and insurance company ID. The Entity Insdata.cs file in the Visual Studio 2017 solution ContosoInsAuthorityAdminPortal.sln contains these properties. Other properties, like the effective start and end dates, content type, and so on, are stored as attributes of the secret in the key vault.

Build the Admin and Customer Portal Applications Refer to the Visual Studio 2017 solution ContosoInsAuthorityAdminPortal.sln in the download for the admin portal source code, and to ContosoinsExtPortal.sln for the customer portal code.

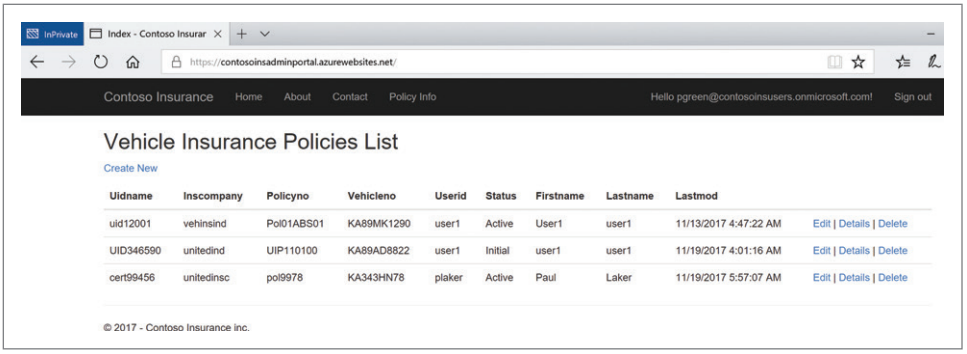


Figure 3 **Insurance Policies List on the Contoso Insurance Admin Portal Landing Page**

Both the admin and customer portals are built using ASP.NET Core 2.0, which supports dependency injection to add framework services—like Entity Framework integration and custom service modules to access the Azure Key Vault APIs—to the application in the start-up class.

Visual Studio 2017 project templates provide turnkey integration with Azure AD and Azure AD B2C for user sign-in and sign-up experiences to secure access to the portal applications.

The connection string for the Azure SQL Database is stored in Azure Key Vault and is retrieved by the portal Web application at start up.

The code snippet in **Figure 2** shows how dependency injection in ASP.NET 2.0 Core is added for Entity Framework Azure AD authentication and for the access provider, Azure Key Vault Service API, access and to read application configuration data from the appsettings.json file.

The Azure Key Vault configuration provider for ASP.NET Core 2.0 (bit.ly/2DfOXeq), available as a NuGet package, provides a turnkey implementation to retrieve all secrets from Azure Key Vault at application start. However, this feature is not used in the solution to avoid loading all the business data unnecessarily on application start, that is, the insurance policy secrets, along with other secrets that application requires, such as the connection string to access the Azure SQL Database. This feature could be used when one key vault instance is used to store all connection strings required by the application, and a separate key vault instance is used for the business data.

App Service Creation The admin and customer portal applications are deployed as Azure App Service Web apps using the built-in tools in Visual Studio 2017. Both Web apps are hosted inside a single App Service Plan.

During development, the ASP.NET Core MVC applications can be deployed locally using Visual Studio 2017. When the PowerShell scripts I discussed earlier are run, two digital certificates are generated, one for each portal application. The .pfx files are added to the current user's certificate store. These are embedded in the requests that the ASP.NET MVC applications make to access the Azure Key Vault APIs. The thumbprints of these certificates are added to the appsettings.json file in the Visual Studio 2017 solution of the respective ASP.NET MVC applications.

When deploying the application to Azure App Service, you must:

- Upload both.pfx files to the Azure App Service instance from the Azure Portal.
- Create the entry "WEBSITE_LOAD_CERTIFICATES" in the App Settings blades of both the admin and customer portal Web apps in the Azure Portal, and add the thumbprint of the respective .pfx file.

Refer to the documentation at bit.ly/2mVEKOq for more details on the steps to be performed.

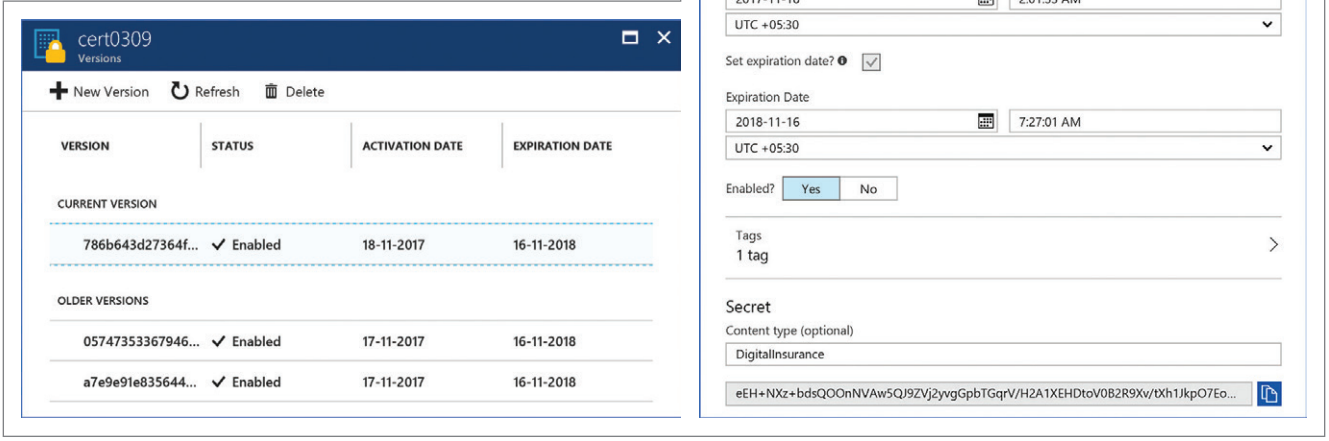


Figure 4 **Different Versions of the Insurance Policy Data Stored as a Secret, as Seen on the Azure Portal**

Database Creation The script file used to create the database for the solution is available for download along with the other artifacts of the solution. Transparent data encryption (TDE) and audit is enabled on the database.

Azure AD and Azure AD B2C Tenant Creation The default Azure AD tenant in the Azure subscription is used as the identity provider for the internal users of the central agency accessing the admin portal. A separate Azure AD tenant is created in this subscription where the users representing the insurance companies are registered. These users are added as guest users to the default Azure AD tenant to access the portal applications. If the insurance companies have their own Azure AD tenant, Azure AD B2B can be used to federate that tenant with the default Azure AD tenant in this subscription.

From the Azure Portal, an Azure AD B2C tenant is created in the Azure subscription and policies are defined to permit customers to self-signup for access to the customer portal. In the identity providers section of the Azure AD B2C configuration, local accounts in this tenant are set to "user name" as opposed to e-mail address, and e-mail verification on user signup is disabled, for simplicity. Refer to the documentation of Azure AD B2C for guidance on creating and configuring policies for the sign-in and sign-up experiences (bit.ly/2n7Vro9).

Running the Application

To allow you to run this solution, I've provided sample credentials for signing in to the admin and customer portals in the GitHub repository associated with this article.

To keep the solution simple for this article, I haven't implemented the step in which insurance companies purchase unique codes. Instead, users of the admin portal would directly execute the next step, where the document serial number, customer information, policy and vehicle details are captured all in one shot.

Figure 3 shows the landing page of the admin portal—Contoso Insurance. You can sign in to the admin portal using the credentials of an insurance company user and select Create New to enter the details for a new document. The document serial number is auto-generated by the application and can be viewed only in the New or Edit Item Form.

Figure 4 shows different versions of the insurance policy data stored as a secret. You can also view additional information, such as content type, tags and attributes. Notice that the Secret is encrypted before it's stored.

> Offline validation can be performed using any QR code reader app on a mobile device, or by using a native Xamarin app.

If you sign in to the customer portal, you can view all the insurance policies that were purchased and are ready for activation. The Edit policy page provides an option to activate the policy, which then updates the status of the policy in Azure SQL Database to Active. Once the policy is activated, you can use the Download Policy option, which generates a QR code for the policy data.

Figure 5 shows the user experience on the customer portal for downloading the QR code. Also shown is the JSON data that is read from the QR code using an app on a mobile device. A native app scans the QR code and displays formatted policy details on the screen, for easy offline verification.

Additional security could be implemented by having the customer portal sign the JSON data using a private key in Azure Key Vault and generating a QR code on the signed data. The native mobile app could be shipped with the public key required to verify the signed data before displaying the JSON data on the device for verification.

The customer portal uses the QR code generator for JavaScript, available on GitHub at bit.ly/2sa3TWy, to generate and display the QR code on the portal.

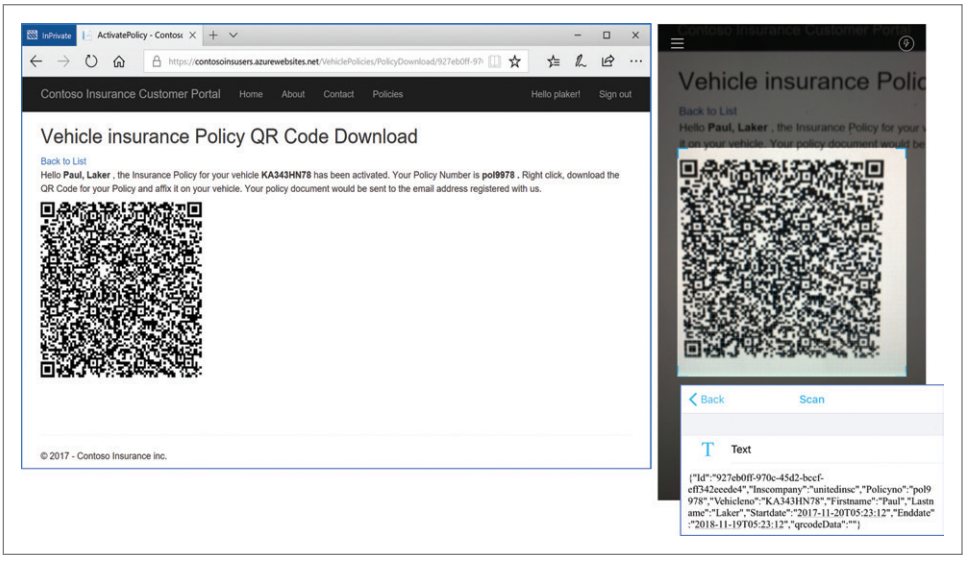


Figure 5 **QR Code Generation**

Policy Validation

Policies can be validated either offline or online. Offline validation can be performed using any QR code reader app on a mobile device, or by using a native Xamarin app. After the QR code is scanned, the results are displayed in a user-friendly way for verification.

In contrast, **Figure 6** shows a request for validation sent using the Postman tool (getpostman.com) to the API in the MVC application, which returns the outcome of the validation as a Boolean. In this case, the policy start date is later than the current date so the outcome of the validation is "false." A

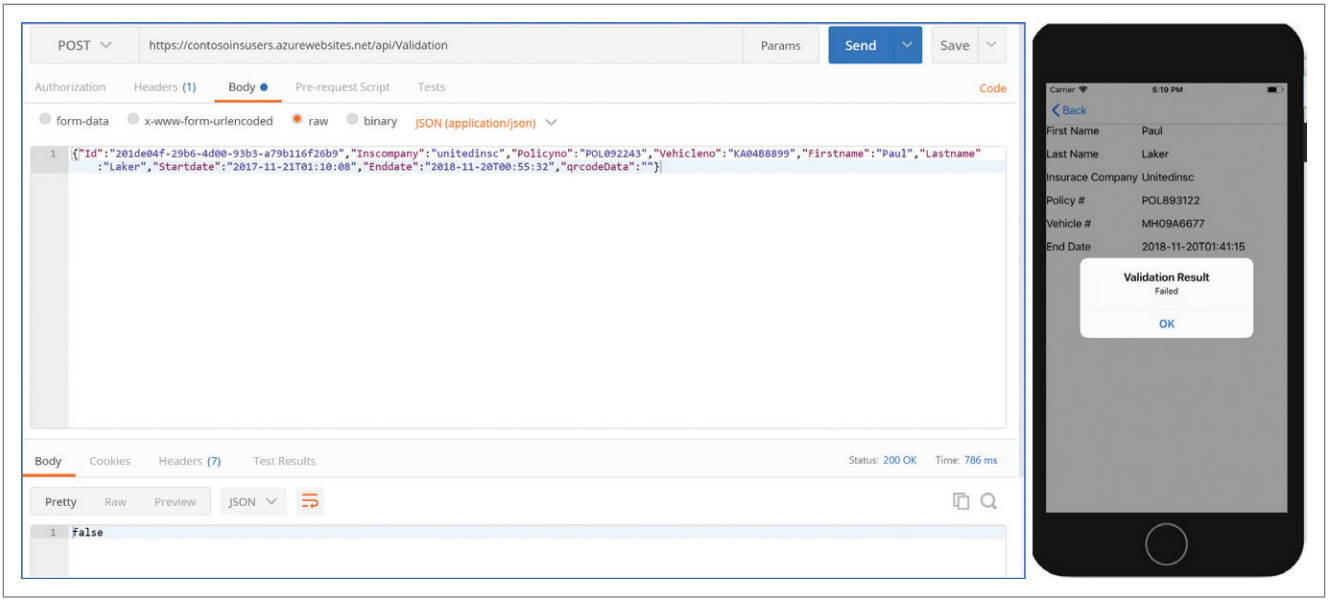


Figure 6 **REST API Call to Validate a Policy**

Xamarin app is used to sign the user in, scan and view the QR code data and make a request to this API to perform online validation.

All operations on Azure Key Vault can be audited and the logs archived for compliance with statutory regulations. You can enable auditing from the Settings blade in the Azure Portal for the Azure Key Vault service instance. To view the logs, navigate to the Azure storage resource configured for the logs. You could use role-based access control (RBAC) in the Azure Portal to ensure only designated users can access this information.

> Additional security could be implemented by having the customer portal sign the JSON data using a private key in Azure Key Vault and generating a QR code on the signed data.

All operations on the data in Azure SQL Database can also be enabled for audit, from the Settings blade of the database instance in the Azure Portal. Transparent data encryption of the data at rest in Azure SQL Database is by enabled by default.

Deploying the Solution

If you'd like to try out this solution on your own, you can download the source files and scripts from the GitHub repository at bit.ly/2DRvwdh. You'll need the following software to implement this solution:

- Visual Studio 2017 Preview, Community or Enterprise Edition with update 3
- An Azure subscription
- A Windows PowerShell script editor
- Postman
- A QR code generator for JavaScript

To deploy the solution in your own subscription, you'll need to update the config entries in the appsettings.json file after the PowerShell Scripts are executed and the other resources are provisioned in the Azure subscription. The steps to do this have been provided in the GitHub repository along with the source code and solution files described in this article. Many thanks to Bindu Chinnasamy of the Microsoft CSE team for help with building the solution accompanying the article.

Wrapping Up

Azure Key Vault provides a helpful, efficient platform for businesses to securely manage their sensitive information, using industry standard algorithms and techniques to perform cryptographic operations. It lets developers use the SDKs for the platforms and languages to which they're accustomed. This, coupled with the rich set of additional services in Azure, such as Azure App Service, Azure AD and Azure B2C, and the elaborate tooling support these services provide, lets developers focus on building core business features, thereby significantly reducing the time required to develop and deploy an end-to-end solution.

In the sequel to this article, I will be demonstrating how the same application could, without significant changes, be deployed to Azure Container Service using Docker Containers and Kubernetes. ■

Srikantan Sankaran *is a principal technical evangelist from the One Commercial Partner team in India, based out of Bangalore. He works with numerous ISVs in India and helps them architect and deploy their solutions on Microsoft Azure. Reach him at sansri@microsoft.com.*

Thanks to the following Microsoft technical expert for reviewing this article: *Frank Hokhold*

Respect the Past. Code the Future.

Visual Studio LIVE! 25 YEARS OF CODING INNOVATION

BACK BY POPULAR DEMAND:
Sunday Hands-On Labs!
Choose From:

- ASP.NET Core Architecture Security
- Xamarin and Xamarin.Forms
- Angular

Starting at **$695** through March 11
SPACE IS LIMITED

Register to code with us today!

Register by March 11 and Save $300!
Must use discount code LVEB01

SUPPORTED BY
Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY
1105 MEDIA
YOUR GROWTH. OUR BUSINESS.

vslive.com/lasvegas

AGENDA AT-A-GLANCE

March 11 – 16, 2018
Bally's Hotel & Casino

LasVegas

ALM / DevOps	Cloud Computing	Database and Analytics	Native Client	Software Practices	Visual Studio / .NET Framework	Web Client	Web Server

START TIME	END TIME						
Full Day Hands-On Labs: Sunday, March 11, 2018 *(Separate entry fee required)*							
8:00 AM	9:00 AM	Pre-Conference Hands-On Lab Registration - Coffee and Morning Pastries					
9:00 AM	6:00 PM	**HOL01** Full Day Hands-On Lab: Modern Security Architecture for ASP.NET Core (Part 1) - *Brock Allen*		**HOL02** Full Day Hands-On Lab: From 0-60 in a Day with Xamarin and Xamarin.Forms - *Roy Cornelissen & Marcel de Vries*		**HOL03** Full Day Hands-On Lab: Busy Developer's HOL on Angular - *Ted Neward*	
4:00 PM	6:00 PM	Conference Registration Open					
Pre-Conference Workshops: Monday, March 12, 2018 *(Separate entry fee required)*							
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries					
9:00 AM	6:00 PM	**HOL01** Full Day Hands-On Lab: Modern Security Architecture for ASP.NET Core (Part 2) - *Brock Allen*		**M02** Workshop: Developer Dive into SQL Server 2016 - *Leonard Lobel*		**M03** Workshop: Add Intelligence to Your Solutions with AI, Bots, and More - *Brian Randell*	
7:00 PM	9:00 PM	Dine-A-Round					
Day 1: Tuesday, March 13, 2018							
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**T01** Go Serverless with Azure Functions - *Eric D. Boyd*		**T02** Getting Ready to Write Mobile Applications with Xamarin - *Kevin Ford*	**T03** What's New in Visual Studio 2017 for C# Developers - *Kasey Uhlenhuth*		**T04** Database Development with SQL Server Data Tools - *Leonard Lobel*
9:30 AM	10:45 AM	**T05** Angular 101 - *Deborah Kurata*		**T06** Lessons Learned from Real World Xamarin.Forms Projects - *Nick Landry*	**T07** Using Visual Studio Mobile Center to Accelerate Mobile Development - *Kevin Ford*		**T08** Introduction to Azure Cosmos DB - *Leonard Lobel*
11:00 AM	12:00 PM	**KEYNOTE: .NET Everywhere and for Everyone** - *James Montemagno, Principal Program Manager - Mobile Developer Tools, Microsoft*					
12:00 PM	1:00 PM	Lunch					
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors					
1:30 PM	2:45 PM	**T09** MVVM and ASP.NET Core Razor Pages - *Ben Hoelting*		**T10** Works On My Machine... Docker for Developers - *Chris Klug*	**T11** DevOps for the SQL Server Database - *Brian Randell*		**T12** To Be Announced
3:00 PM	4:15 PM	**T13** Angular Component Communication - *Deborah Kurata*		**T14** Customizing Your UI for Mobile Devices: Techniques to Create a Great User Experience - *Laurent Bugnion*	**T15** PowerShell for Developers - *Brian Randell*		**T16** To Be Announced
4:15 PM	5:30 PM	Welcome Reception					
Day 2: Wednesday, March 14, 2018							
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**W01** The Whirlwind Tour of Authentication and Authorization with ASP.NET Core - *Chris Klug*		**W02** Building Mixed Reality Experiences for HoloLens & Immersive Headsets in Unity - *Nick Landry*	**W03** Using Feature Toggles to Separate Releases from Deployments - *Marcel de Vries*		**W04** Lock the Doors, Secure the Valuables, and Set the Alarm - *Eric D. Boyd*
9:30 AM	10:45 AM	**W05** TypeScript: The Future of Front End Web Development - *Ben Hoelting*		**W06** A Dozen Ways to Mess Up Your Transition From Windows Forms to XAML - *Billy Hollis*	**W07** Overcoming the Challenges of Mobile Development in the Enterprise - *Roy Cornelissen*		**W08** Computer, Make It So! - *Veronika Kolesnikova & Willy Ci*
11:00 AM	12:00 PM	**General Session: The Act of Creation—How Dev Tooling Makes Successful Developers** - *Kasey Uhlenhuth, Program Manager - .NET Managed Languages, Microsoft*					
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch					
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:15pm (Must be present to win)					
1:30 PM	1:50 PM	**W09** Fast Focus: 0-60 for Small Projects in Visual Studio Team Services - *Alex Mullans*		**W10** Fast Focus: Cross Platform Device Testing with xUnit - *Oren Novotny*		**W11** Fast Focus: Understanding .NET Standard - *Jason Bock*	
2:00 PM	2:20 PM	**W12** Fast Focus: HTTP/2: What You Need to Know - *Robert Boedigheimer*		**W13** Fast Focus: Serverless Computing: Azure Functions and Xamarin in 20 Minutes - *Laurent Bugnion*		**W14** Fast Focus: Why You Should Love SQL Server 2017 - *Scott Klein*	
2:30 PM	3:45 PM	**W15** Advanced Fiddler Techniques - *Robert Boedigheimer*		**W16** Building Cross-Platforms Business Apps with C# and CSLA .NET - *Rockford Lhotka*	**W17** Versioning NuGet and npm Packages - *Alex Mullans*		**W18** Getting to the Core of .NET Core - *Adam Tuliper*
4:00 PM	5:15 PM	**W19** Assembling the Web - A Tour of WebAssembly - *Jason Bock*		**W20** Radically Advanced XAML: Dashboards, Timelines, Animation, and More - *Billy Hollis*	**W21** Encrypting the Web - *Robert Boedigheimer*		**W22** Porting MVVM Light to .NET Standard: Lessons Learned - *Laurent Bugnion*
7:00 PM	8:30 PM	VSLive! High Roller Evening Out					
Day 3: Thursday, March 15, 2018							
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	**TH01** ASP.NET Core 2 For Mere Mortals - *Philip Japikse*		**TH02** Performance in 60 Seconds – SQL Tricks Everybody MUST Know - *Pinal Dave*	**TH03** Demystifying Microservice Architecture - *Miguel Castro*		**TH04** Cognitive Services in Xamarin Applications - *Veronika Kolesnikova*
9:30 AM	10:45 AM	**TH05** Getting to the Core of ASP.NET Core Security - *Adam Tuliper*		**TH06** Secrets of SQL Server - Database Worst Practices - *Pinal Dave*	**TH07** Unit Testing Makes Me Faster: Convincing Your Boss, Your Co-Workers, and Yourself - *Jeremy Clark*		**TH08** Publish Your Angular App to Azure App Services - *Brian Noyes*
11:00 AM	12:00 PM	**Panel Discussion: Security in Modern App Development** - *Rockford Lhotka (moderator), Damian Brady, Pinal Dave, Veronika Kolesnikova, James McCaffrey, and Oren Novotny*					
12:00 PM	1:00 PM	Lunch					
1:00 PM	2:15 PM	**TH09** Entity Framework Core 2 For Mere Mortals - *Philip Japikse*		**TH10** SQL Server 2017 - Intelligence Built-in - *Scott Klein*	**TH11** Writing Testable Code and Resolving Dependencies - DI Kills Two Birds with One Stone - *Miguel Castro*		**TH12** Signing Your Code the Easy Way - *Oren Novotny*
2:30 PM	3:45 PM	**TH13** Analyzing Code in .NET - *Jason Bock*		**TH14** Introduction to Azure Machine Learning - *James McCaffrey*	**TH15** "Doing DevOps" as a Politically Powerless Developer - *Damian Brady*		**TH16** Busy Developer's Guide to Chrome Development - *Ted Neward*
4:00 PM	5:15 PM	**TH17** Securing Web Apps and APIs with IdentityServer - *Brian Noyes*		**TH18** Introduction to the CNTK v2 Machine Learning Library - *James McCaffrey*	**TH19** I'll Get Back to You: Task, Await, and Asynchronous Methods - *Jeremy Clark*		**TH20** Multi-targeting the World: A Single Project to Rule Them All - *Oren Novotny*
Post-Conference Workshops: Friday, March 16, 2018 *(Separate entry fee required)*							
7:30 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries					
8:00 AM	5:00 PM	**F01** Workshop: Creating Mixed Reality Experiences for HoloLens & Immersive Headsets with Unity - *Nick Landry & Adam Tuliper*		**F02** Workshop: Distributed Cross-Platform Application Architecture - *Jason Bock & Rockford Lhotka*		**F03** Workshop: UX Design for Developers: Basics of Principles and Process - *Billy Hollis*	

Speakers and sessions subject to change

DINO ESPOSITO

REST and Web API in ASP.NET Core

I've never been a fan of ASP.NET Web API as a standalone framework and I can't hardly think of a project where I used it. Not that the framework in itself is out of place or unnecessary. I just find that the business value it actually delivers is, most of the time, minimal. On the other hand, I recognize in it some clear signs of the underlying effort Microsoft is making to renew the ASP.NET runtime pipeline. Overall, I like to think of ASP.NET Web API as a proof of concept for what today has become ASP.NET Core and, specifically, the new runtime environment of ASP.NET Core.

Web API was primarily introduced as a way to make building a RESTful API easy and comfortable in ASP.NET. This article is about how to achieve the same result—building a RESTful API—in ASP.NET Core.

The Extra Costs of Web API in Classic ASP.NET

ASP.NET Web API was built around the principles sustaining the Open Web Interface for .NET (OWIN) specification, which is meant to decouple the Web server from hosted Web applications. In the .NET space, the introduction of OWIN marked a turning point, where the tight integration of IIS and ASP.NET was questioned. That tight coupling was fully abandoned in ASP.NET Core.

Any Web façade built using the ASP.NET Web API framework relies on a completely rewritten pipeline that uses the standard OWIN interface to dialog with the underlying host Web server. Yet, an ASP.NET Web API is not a standalone application. To be available for callers it needs a host environment that takes care of listening to some configured port, captures incoming requests and dispatches them down the Web API pipeline.

A Web API application can be hosted in a Windows service or in a custom console application that implements the appropriate OWIN interfaces. It can also be hosted by a classic ASP.NET application, whether targeting Web Forms or ASP.NET MVC. Over the past few years, hosting Web API within a classic ASP.NET MVC application proved to be a very common scenario, yet one of the least effective in terms of raw performance and memory footprint.

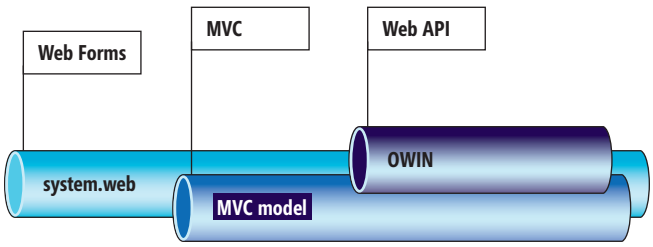


Figure 1 **Frameworks Involved in a Classic ASP.NET Web API Application**

As **Figure 1** shows, whenever you arrange a Web API façade within an ASP.NET MVC application, three frameworks end up living side-by-side, processing every single Web API request. The host ASP.NET MVC application is encapsulated in an HTTP handler living on top of system.web—the original ASP.NET runtime environment. On top of that—taking up additional memory—you have the OWIN-based pipeline of Web API.

> Web API is the most high-profile example of the OWIN principles in action.

The vision of introducing a server-independent Web framework is, in this case, significantly weakened by the constraints of staying compatible with the existing ASP.NET pipeline. Therefore, the clean and REST-friendly design of Web API doesn't unleash its full potential because of the legacy system.web assembly. From a pure performance perspective, only some edge use cases really justify the use of Web API.

Effective Use Cases for Web API

Web API is the most high-profile example of the OWIN principles in action. A Web API library runs behind a server application that captures and forwards incoming requests. This host can be a classic Web application on the Microsoft stack (Web Forms, ASP.NET MVC) or it can be a console application or a Windows service.

In any case, it has to be an application endowed with a thin layer of code capable of dialoging with the Web API listener.

Hosting a Web API outside of the Web environment removes at the root any dependency on the system.web assembly, thus magically making the request pipeline as lean and mean as desired.

This is the crucial point that led the ASP.NET Core team to build the ASP.NET Core pipeline. The ideal hosting conditions for Web API have been reworked to be the ideal hosting conditions for just about any ASP.NET Core application. This enabled a completely new pipeline devoid of dependencies on the system.web assembly and hostable behind an embedded HTTP server exposing a contracted interface—the IServer interface.

The OWIN specification and Katana, the implementation of it for the IIS/ASP.NET environment, play no role in ASP.NET Core. But the experience with these platforms matured the technical vision (especially with Web API edge cases), which shines through the dazzling new pipeline of ASP.NET Core.

The funny thing is that once the entire ASP.NET pipeline was redesigned—deeply inspired by the ideal hosting environment for Web API—that same Web API as a separate framework ceased to be relevant. In the new ASP.NET Core pipeline there's the need for just one application model—the MVC application model—based on controllers, and controller classes are a bit richer than in classic ASP.NET MVC, thus incorporating the functions of old ASP.NET controllers and Web API controllers.

Extended ASP.NET Core Controllers

In ASP.NET Core, you work with controller classes whether you intend to serve HTML or any other type of response, such as JSON or PDF. A bunch of new action result types have been added to make building RESTful interfaces easy and convenient. Content negotiation is fully supported for any controller classes, and formatting helpers have been baked into the action invoker infrastructure. If you want to build a Web API that exposes HTTP endpoints, all you do is build a plain controller class, as shown here:

```
public class ApiController : Controller
{
  // Your methods here
}
```

The name of the controller class is arbitrary. While having /api somewhere in the URL is desirable for clarity, it's in no way required. You can have /api in the URL being invoked both if you use conventional routing (an ApiController class) to map URLs to action methods, or if you use attribute routing. In my personal opinion, attribute routing is probably preferable because it allows you to expose multiple endpoints with the same /api item in the URL, while being defined in distinct, arbitrarily named controller classes.

In ASP.NET Core, you work with controller classes whether you intend to serve HTML or any other type of response, such as JSON or PDF.

The Controller class in ASP.NET Core has a lot more features than the class in classic ASP.NET MVC, and most of the extensions relate to building a RESTful Web API. First and foremost, all ASP.NET Core controllers support content negotiation. Content negotiation refers to a silent negotiation taking place between the caller and the API regarding the actual format of returned data.

Content negotiation doesn't happen all the time and for just every request. It takes place only if the incoming request contains an Accept HTTP header that advertises the MIME types the caller is able to understand. In this case, the ASP.NET Core infrastructure goes through the types listed in the header content until it finds one for which a formatter exists in the current configuration of the application. If no matching formatter is found in the list of types, then the default JSON formatter is used, like so:

```
[HttpGet]
public ObjectResult Get(Guid id)
{
  // Do something here to retrieve the resource data
  var data = FindResourceDataInSomeWay(id);

  return Ok(data);
}
```

The Controller class in ASP.NET Core has a lot more features than the class in classic ASP.NET MVC, and most of the extensions relate to building a RESTful Web API.

Another remarkable aspect of content negotiation is that while it won't produce any change in the serialization process without an Accept HTTP header, it's technically triggered only if the response being sent back by the controller is of type ObjectResult. The most common way to return an ObjectResult action result type is by serializing the response via the Ok method. It's important to note that if you serialize the controller response via, say, the Json method, no negotiation will ever take place regardless of the headers sent. Support for output formatters can be added programmatically through the options of the AddMvc method. Here's an example:

```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new PdfFormatter());
});
```

In this example, the demo class PdfFormatter contains internally the list of supported MIME types it can handle.

Note that by using the Produces attribute you override the content negotiation, as shown here:

```
[Produces("application/json")]
public class ApiController : Controller
{
  // Action methods here
}
```

The Produces attribute, which you can apply at the controller or method level, forces the output of type ObjectResult to be always serialized in the format specified by the attribute, regardless of the Accept HTTP header.

For more information on how to format the response of a controller method, you might want to check out the content at bit.ly/2kIDgdY.

REST-Oriented Action Result Types

Whether a Web API is better off with a REST design is a highly debatable point. In general, it's safe enough to say that the REST approach is based on a known set of rules and, in this regard, it is more standard. For this reason, it's generally recommended for a public API

Figure 2 **Web API-Related Action Result Types**

Type	Description
AcceptedResult	Returns a 202 status code. In addition, it returns the URI to check on the ongoing status of the request. The URI is stored in the Location header.
BadRequestResult	Returns a 400 status code.
CreatedResult	Returns a 201 status code. In addition, it returns the URI of the resource created, stored in the Location header.
NoContentResult	Returns a 204 status code and null content.
OkResult	Returns a 200 status code.
UnsupportedMediaTypeResult	Returns a 415 status code.

that's part of the enterprise business. If the API exists only to serve a limited number of clients—mostly under the same control of the API creators—then no real business difference exists between using REST design route or a looser remote-procedure call (RPC) approach.

In ASP.NET Core, there's nothing like a distinct and dedicated Web API framework. There are just controllers with their set of action results and helper methods. If you want to build a Web API whatsoever, you just return JSON, XML or whatever else. If you want to build a RESTful API, you just get familiar with another set of action results and helper methods. **Figure 2** presents the new action result types that ASP.NET Core controllers can return. In ASP.NET Core, an action result type is a type that implements the IActionResult interface.

Figure 3 **Common RESTful Skeleton of Code**

```
[HttpGet]
public ObjectResult Get(Guid id)
{
  // Do something here to retrieve the resource
  var res = FindResourceInSomeWay(id);

  return Ok(res);
}

[HttpPut]
public AcceptedResult UpdateResource(Guid id, string content)
{
  // Do something here to update the resource
  var res = UpdateResourceInSomeWay(id, content);
  var path = String.Format("/api/resource/{0}", res.Id);
  return Accepted(new Uri(path));
}

[HttpPost]
public CreatedResult AddNews(MyResource res)
{
  // Do something here to create the resource
  var resId = CreateResourceInSomeWay(res);

  // Returns HTTP 201 and sets the URI to the Location header
  var path = String.Format("/api/resource/{0}", resId);
  return Created(path, res);
}

[HttpDelete]
public NoContentResult DeleteResource(Guid id)
{
  // Do something here to delete the resource
  // ...

  return NoContent();
}
```

Note that some of the types in **Figure 2** come with buddy types that provide the same core function but with some slight differences. For example, in addition to AcceptedResult and CreatedResult, you find xxxAtActionResult and xxxAtRouteResult types. The difference is in how the types express the URI to monitor the status of the accepted operation and the location of the resource just created. The xxxAtActionResult type expresses the URI as a pair of controller and action strings whereas the xxxAtRouteResult type uses a route name.

OkObjectResult and BadRequestObjectResult, instead, have an xxxObjectResult variation. The difference is that object result types also let you append an object to the response. So OkResult just sets a 200 status code, but OkObjectResult sets a 200 status code and appends an object of your choice. A common way to use this feature is to return a ModelState dictionary updated with the detected error when a bad request is handled.

Another interesting distinction is between NoContentResult and EmptyResult. Both return an empty response, but NoContentResult sets a status code of 204, whereas EmptyResult sets a 200 status code. All this said, building a RESTful API is a matter of defining the resource being acted on and arranging a set of calls using the HTTP verb to perform common manipulation operations. You use GET to read, PUT to update, POST to create a new resource and DELETE to remove an existing one. **Figure 3** shows the skeleton of a RESTful interface around a sample resource type as it results from ASP.NET Core classes.

> A Web API is a programmatic interface comprising a number of publicly exposed HTTP endpoints.

If you're interested in further exploring the implementation of ASP.NET Core controllers for building a Web API, have a look at the GitHub folder at bit.ly/2j4nyUe.

Wrapping Up

A Web API is a common element in most applications today. It's used to provide data to an Angular or MVC front end, as well as to provide services to mobile or desktop applications. In the context of ASP.NET Core, the term "Web API" finally achieves its real meaning without ambiguity or need to further explain its contours. A Web API is a programmatic interface comprising a number of publicly exposed HTTP endpoints that typically (but not necessarily) return JSON or XML data to callers. The controller infrastructure in ASP.NET Core fully supports this vision with a revamped implementation and new action result types. Building a RESTful API in ASP.NET has never been easier! ■

Dino Esposito *is the author of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2014) and "Programming ASP.NET Core" (Microsoft Press, 2018). A Pluralsight author and developer advocate at JetBrains, Esposito shares his vision of software on Twitter: @despos.*

Thanks to the following Microsoft technical expert for reviewing this article: *Ugo Lattanzi*

Spreadsheets Everywhere.

SpreadsheetGear 2017 Released

SpreadsheetGear 2017 adds a new SpreadsheetGear for .NET Standard product, official support for Excel 2013 and Excel 2016, 51 new Excel functions for a total of 449 fully supported functions, full conditional formatting support, enhanced workbook protection and encryption, cell gradient rendering and more.

Support for iOS, Android, Linux, macOS, UWP and more

SpreadsheetGear for .NET Standard enables cross-platform developers to enjoy the same high performance Excel-compatible reporting, charting, calculations and more relied on by thousands of Windows developers for 10+ years.

Scalable Reporting

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

WIN Windows Forms | Silverlight | WPF

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.

Fastest Calculations

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

Download your free fully functional evaluation at SpreadsheetGear.com

SpreadsheetGear

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

AUGUST 6 – 10, 2018 • Microsoft Headquarters, Redmond, WA

Change is ~~Coming.~~ Are You Ready? HERE

Join us for TechMentor, August 6 – 8, 2018, as we return to Microsoft Headquarters in Redmond, WA. In today's IT world, more things change than stay the same. As we celebrate the 20th year of TechMentor, we are more committed than ever to providing immediately usable IT education, with the tools you need today, while preparing you for tomorrow – **keep up, stay ahead and avoid Winter, ahem, Change.**

Plus you'll be at the source, Microsoft HQ, where you can have lunch with Blue Badges, visit the company store, and experience life on campus for a week!

You owe it to yourself, your company and your career to be at TechMentor Redmond 2018!

SAVE $400!
REGISTER NOW
Use Promo Code TMJAN2

EVENT PARTNER Microsoft

SUPPORTED BY Redmond | Redmond Channel Partner | Virtualization & Cloud Review

PRODUCED BY 1105 MEDIA — YOUR GROWTH, OUR BUSINESS.

AGENDA AT-A-GLANCE

Client and Endpoint Management	PowerShell and DevOps	Infrastructure	Soft Skills for ITPros	Security	Cloud (Public/ Hybrid/Private)

START TIME	END TIME				
TechMentor Pre-Conference Workshops: Monday, August 6, 2018 *(Separate entry fee required)*					
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Light Breakfast			
9:00 AM	12:00 PM	**M01** Workshop: How to Prevent all Ransomware / Malware in 2018 - *Sami Laiho*	**M02** Workshop: Building Office 365 Federated Identity from Scratch Using AD FS - *Nestori Syynimaa*	**M03** Workshop: Managing Windows Server with Project Honolulu - *Dave Kawula*	
12:00 PM	2:00 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
2:00 PM	5:00 PM	**M01** Workshop: How to Prevent all Ransomware / Malware in 2018 (Continues) - *Sami Laiho*	**M04** Workshop: Master PowerShell Tricks for Windows Server 2016 and Windows 10 - *Will Anderson & Thomas Rayner*	**M05** Workshop: Dave Kawula's Notes from the Field on Microsoft Storage Spaces Direct - *Dave Kawula*	
6:30 PM	8:30 PM	Dine-A-Round Dinner - Suite in Hyatt Regency Lobby			
START TIME	**END TIME**	**TechMentor Day 1: Tuesday, August 7, 2018**			
7:00 AM	8:00 AM	Registration - Coffee and Light Breakfast			
8:00 AM	9:15 AM	**T01** Enterprise Client Management in a Modern World - *Kent Agerlund*	**T02** How to Write (PowerShell) Code that Doesn't Suck - *Thomas Rayner*	**T03** The Easy Peasy of Troubleshooting Azure - *Mike Nelson*	**T04** Nine O365 Security Issues Microsoft Probably Hasn't Told You (and You Probably Don't Want to Know) - *Nestori Syynimaa*
9:30 AM	10:45 AM	**T05** Managing Client Health—Getting Close to the Famous 100% - *Kent Agerlund*	**T06** The Network is Slow! Or is it? Network Troubleshooting for Windows Administrators - *Richard Hicks*	**T07** Getting Started with PowerShell 6.0 for IT Pro's - *Sven van Rijen*	**T08** The Weakest Link of Office 365 Security - *Nestori Syynimaa*
11:00 AM	12:00 PM	**KEYNOTE: To Be Announced** - *Stephen L. Rose, Sr. PMM, One Drive For Business, Microsoft*			
12:00 PM	1:00 PM	Lunch - *McKinley* / Visit Exhibitors - *Foyer*			
1:00 PM	2:15 PM	**T09** How to Get Started with Microsoft EMS Right Now - *Peter Daalmans*	**T10** Back to the Future! Access Anywhere with Windows 10 Always on VPN - *Richard Hicks*	**T11** Using Desired State Configuration in Azure - *Will Anderson*	**T11** To Be Announced
2:15 PM	2:45 PM	Sponsored Break - Visit Exhibitors - *Foyer*			
2:45 PM	4:00 PM	**T12** Conceptualizing Azure Resource Manager Templates - *Will Anderson*	**T13** How to Use PowerShell to Become a Windows Management SuperHero - *Petri Paavola*	**T14** Making the Most Out of the Azure Dev/Test Labs - *Mike Nelson*	**T15** To Be Announced
4:00 PM	5:30 PM	Exhibitor Reception – Attend Exhibitor Demo - *Foyer*			
START TIME	**END TIME**	**TechMentor Day 2: Wednesday, August 8, 2018**			
7:30 AM	8:00 AM	Registration - Coffee and Light Breakfast			
8:00 AM	9:15 AM	**W01** Automated Troubleshooting Techniques in Enterprise Domains (Part 1) - *Petri Paavola*	**W02** Troubleshooting Sysinternals Tools 2018 Edition - *Sami Laiho*	**W03** In-Depth Introduction to Docker - *Neil Peterson*	**W04** How Microsoft Cloud Can Support Your GDPR Journey - *Milad Aslaner*
9:30 AM	10:45 AM	**W05** Automated Troubleshooting Techniques in Enterprise Domains (Part 2) - *Petri Paavola*	**W06** What's New in Windows Server 1803 - *Dave Kawula*	**W07** Simplify and Streamline Office 365 Deployments the Easy Way - *John O'Neill, Sr.*	**W08** How to Administer Microsoft Teams Like a Boss - *Ståle Hansen*
11:00 AM	12:00 PM	**TECHMENTOR PANEL: The Future of Windows** - *Peter De Tender, Dave Kawula, Sami Laiho, & Petri Paavola*			
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch - *McKinley* / Visit Exhibitors - *Foyer*			
1:00 PM	1:30 PM	Networking Break - Exhibitor Raffle @ 1:10 pm (Must be present to win) - *Foyer in front of Business Center*			
1:30 PM	2:45 PM	**W09** Putting the Windows Assessment and Deployment Kit to Work - *John O'Neill, Sr.*	**W10** Deploying Application Whitelisting on Windows Pro or Enterprise - *Sami Laiho*	**W11** Azure is 100% High-Available... Or Is It? - *Peter De Tender*	**W12** What the NinjaCat Learned from Fighting Cybercrime - *Milad Aslaner*
3:00 PM	4:15 PM	**W13** The Evolution of a Geek—Becoming an IT Architect - *Mike Nelson*	**W14** Advanced DNS, DHCP and IPAM Administration on Windows Server 2016 - *Orin Thomas*	**W15** Managing Tesla Vehicles from the Cloud - *Marcel Zehner*	**W16** Nano Server—Containers in the Cloud - *David O'Brien*
6:15 PM	9:00 PM	Set Sail! TechMentor's Seattle Sunset Cruise - *Busses depart the Hyatt Regency at 6:15pm to travel to Kirkland City Dock*			
START TIME	**END TIME**	**TechMentor Day 3: Thursday, August 9, 2018**			
7:30 AM	8:00 AM	Registration - Coffee and Light Breakfast			
8:00 AM	9:15 AM	**TH01** Manage Your Apple Investments with Microsoft EMS - *Peter Daalmans*	**TH02** Tips and Tricks for Managing and Running Ubuntu/Bash/Windows Subsystem for Linux - *Orin Thomas*	**TH03** The OMS Solutions Bakery - *Marcel Zehner*	**TH04** Getting Started with PowerShell for Office 365 - *Vlad Catrinescu*
9:30 AM	10:45 AM	**TH05** HoloLens, Augmented Reality, and IT - *John O'Neill, Sr.*	**TH06** A Real-world Social Engineering Attack and Response - *Milad Aslaner*	**TH07** 30 Terrible Habits of Server and Cloud Administrators - *Orin Thomas*	**TH08** Advanced PowerShell for Office 365 - *Vlad Catrinescu*
11:00 AM	12:15 PM	**TH09** 10 Tips to Control Access to Corporate Resources with Enterprise Mobility + Security - *Peter Daalmans*	**TH10** What's New and Trending with Microsoft Enterprise Client Management - *Kent Agerlund*	**TH11** OneNote LifeHack: 5 Steps for Succeeding with Personal Productivity - *Ståle Hansen*	**TH12** Managing Virtual Machines on AWS—Like in Real Life! - *David O'Brien*
12:15 PM	2:15 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
2:15 PM	3:30 PM	**TH13** Security Implications of Virtualizing Active Directory Domain Controllers - *Sander Berkouwer*	**TH14** Building a New Career in 5 Hours a Week - *Michael Bender*	**TH15** Azure CLI 2.0 Deep Dive - *Neil Peterson*	**TH16** Open SSH for Windows - *Anthony Nocentino*
3:45 PM	5:00 PM	**TH17** Running Hyper-V in Production for 10 years - Notes from the Field - *Dave Kawula*	**TH18** Network Sustainability and Cyber Security Measures - *Omar Valerio*	**TH19** Azure AD Connect Inside and Out - *Sander Berkouwer*	**TH20** I Needed to Install 80 SQL Servers...Fast. Here's How I Did It! - *Anthony Nocentino*
START TIME	**END TIME**	**TechMentor Post-Conference Workshops: Friday, August 10, 2018** *(Separate entry fee required)*			
8:30 AM	9:00 AM	Post-Conference Workshop Registration - Coffee and Light Breakfast			
9:00 AM	12:00 PM	**F01** Workshop: Hardening Your Windows Server Environment - *Orin Thomas*		**F02** Workshop: Learn the Latest and Greatest Updates to the Azure Platform IaaS and PaaS Services v2.0 - *Peter De Tender*	
12:00 PM	1:00 PM	Lunch - *McKinley*			
1:00 PM	4:00 PM	**F01** Workshop: Hardening Your Windows Server Environment (Continues) - *Orin Thomas*		**F02** Workshop: Learn the Latest and Greatest Updates to the Azure Platform IaaS and PaaS Services v2.0 (Continues) - *Peter De Tender*	

Speakers and sessions subject to change

JAMES MCCAFFREY

Neural Binary Classification Using CNTK

The goal of a binary classification problem is to make a prediction where the value to predict can take one of just two possible values. For example, you might want to predict if a hospital patient has heart disease or not, based on predictor variables such as age, blood pressure, sex and so on. There are many techniques that can be used to tackle a binary classification problem. In this article I'll explain how to use the Microsoft Cognitive Toolkit (CNTK) library to create a neural network binary classification model.

Take a look at **Figure 1** to see where this article is headed. The demo program creates a prediction model for the Cleveland Heart Disease dataset. The dataset has 297 items. Each item has 13 predictor variables: age, sex, pain type, blood pressure, cholesterol, blood sugar, ECG, heart rate, angina, ST depression, ST slope, number of vessels and thallium. The value to predict is the presence or absence of heart disease.

Behind the scenes, the raw data was normalized and encoded, resulting in 18 predictor variables. The demo creates a neural network with 18 input nodes, 20 hidden processing nodes and two output nodes. The neural network model is trained using stochastic gradient descent with a learning rate set to 0.005 and a mini-batch size of 10.

```
C:\WINDOWS\system32\cmd.exe

C:\NeuralBinaryClassification>python cleveland_bnn.py

Begin Cleveland Heart Disease binary classification (two-node technique)

Using CNTK version = 2.3

Creating a 18-20-2 tanh-softmax NN for Cleveland Heart dataset
Selected CPU as the process wide default device.
Creating an ordinary cross entropy batch=10 SGD LR=0.005 Trainer

Starting training

batch    0: mean loss = 0.6932, accuracy = 40.00%
batch  500: mean loss = 0.7084, accuracy = 30.00%
batch 1000: mean loss = 0.5456, accuracy = 90.00%
batch 1500: mean loss = 0.4894, accuracy = 80.00%
batch 2000: mean loss = 0.4164, accuracy = 80.00%
batch 2500: mean loss = 0.7000, accuracy = 80.00%
batch 3000: mean loss = 0.2024, accuracy = 100.00%
batch 3500: mean loss = 0.1493, accuracy = 100.00%
batch 4000: mean loss = 0.2583, accuracy = 90.00%
batch 4500: mean loss = 0.3860, accuracy = 90.00%

Training complete

Evaluating accuracy using built-in test_minibatch()

Classification accuracy on the 297 data items = 84.18%

End Cleveland Heart Disease classification

C:\NeuralBinaryClassification>_
```

Figure 1 **Binary Classification Using a CNTK Neural Network**

During training, the average loss/error and the average classification accuracy on the current 10 items is displayed every 500 iterations. You can see that, in general, loss/error gradually decreased and accuracy increased over the 5,000 iterations. After training, the classification accuracy of the model on all 297 data items was computed to be 84.18% (250 correct, 47 incorrect).

This article assumes you have intermediate or better programming skill, but doesn't assume you know much about CNTK or neural networks. The demo is coded using Python, but even if you don't know Python, you should be able to follow along without too much difficulty. The code for the demo program is presented in its entirety in this article. The data file used is available in the accompanying download.

Code download available at msdn.com/magazine/0318magcode.

Understanding the Data

There are several versions of the Cleveland Heart Disease dataset at bit.ly/2EL9Leo. The demo uses the processed version, which has 13 of the original 76 predictor variables. The raw data has 303 items and looks like:

```
[001] 63.0,1.0,1.0,145.0,233.0,1.0,2.0,150.0,0.0,2.3,3.0,0.0,6.0,0
[002] 67.0,1.0,4.0,160.0,286.0,0.0,2.0,108.0,1.0,1.5,2.0,3.0,3.0,2
[003] 67.0,1.0,4.0,120.0,229.0,0.0,2.0,129.0,1.0,2.6,2.0,2.0,7.0,1
...
[302] 57.0,0.0,2.0,130.0,236.0,0.0,2.0,174.0,0.0,0.0,2.0,1.0,3.0,1
[303] 38.0,1.0,3.0,138.0,175.0,0.0,0.0,173.0,0.0,0.0,1.0,?,3.0,0
```

The first 13 values in each line are predictors. The last item in each line is a value between 0 and 4 where 0 means absence of heart disease and 1, 2, 3, or 4 means presence of heart disease. In general, the most time-consuming part of most machine learning scenarios is preparing your data. Because there are more than two predictor variables, it's not possible to graph the raw data. But you can get a rough idea of the problem by looking at just age and blood pressure, as shown in **Figure 2**.

The first step is to deal with missing data—notice the "?" in item [303]. Because there are only six items with missing values, those six items were just tossed out, leaving 297 items.

The next step is to normalize the numeric predictor values, such as age in the first column. The demo used min-max normalization where the value in a column is replaced by (value - min) / (max - min). For example, the minimum age value is 29 and the maximum is 77, so the first age value, 63, is normalized to (63 - 29) / (77 - 29) = 34 / 48 = 0.70833.

The next step is to encode the categorical predictor values, such as sex (0 = female, 1 = male) in the second column and pain type (1, 2, 3, 4) in the third column. The demo used 1-of-(N-1) encoding so sex is encoded as female = -1, male = +1. Pain type is encoded as 1 = (1, 0, 0), 2 = (0, 1, 0), 3 = (0, 0, 1), 4 = (-1, -1, -1).

The last step is to encode the value to predict. When using a neural network for binary classification, you can encode the value to predict using just one node with a value of 0 or 1, or you can use two nodes with values of (0, 1) or (1, 0). For a reason I'll explain shortly, when using CNTK, it's much better to use the two-node technique. So, 0 (no heart disease) was encoded as (0, 1) and values 1 through 4 (heart disease) were encoded as (1, 0).

The final normalized and encoded data was tab-delimited and looks like:

```
|symptoms  0.70833  1  1  0  0  0.48113 ... |disease  0  1
|symptoms  0.79167  1 -1 -1 -1  0.62264 ... |disease  1  0
...
```

Tags "|symptoms" and "|disease" were inserted so the data could be easily read by a CNTK data reader object.

The Demo Program

The complete demo program, with a few minor edits to save space, is presented in **Figure 3**. All normal error checking has been removed. I indent with two space characters instead of the usual four as a matter of personal preference and to save space. The "\" character is used by Python for line continuation.

The cleveland_bnn.py demo has one helper function, create_reader. All control logic is in a single main function. Because CNTK is young and under vigorous development, it's a good idea to add a comment detailing which version is being used (2.3 in this case).

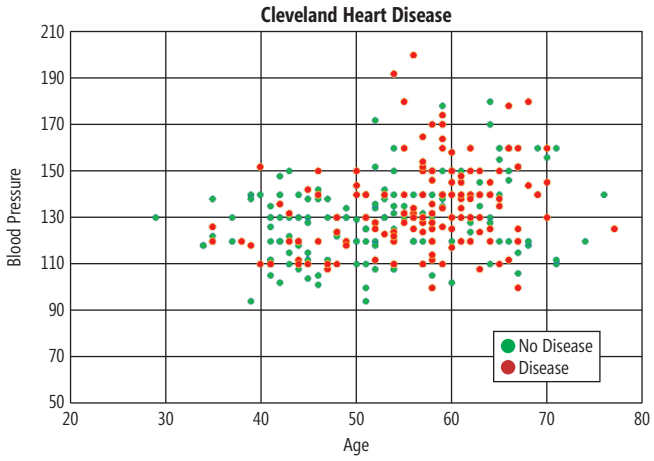


Figure 2 **Cleveland Heart Disease Partial Raw Data**

Installing CNTK can be a bit tricky. First, you install the Anaconda distribution of Python, which contains the required Python interpreter, required packages such as NumPy and SciPy, and useful utilities such as pip. I used Anaconda3 4.1.1 64-bit, which includes Python 3.5. After installing Anaconda, you install CNTK as a Python package, not as a standalone system, using the pip utility. From an ordinary shell, the command I used was:

```
>pip install https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
```

Almost all CNTK installation failures I've seen have been due to Anaconda-CNTK version incompatibilities.

The demo begins by preparing to create the neural network:

```
input_dim = 18
hidden_dim = 20
output_dim = 2
train_file = ".\\Data\\cleveland_cntk_twonode.txt"

X = C.ops.input_variable(input_dim, np.float32)
Y = C.ops.input_variable(output_dim, np.float32)
```

The number of input and output nodes is determined by your data, but the number of hidden processing nodes is a free parameter and must be determined by trial and error. Using 32-bit variables is typical for neural networks because the precision gained by using 64 bits isn't worth the performance penalty incurred.

The network is created like so:

```
with C.layers.default_options(init=C.initializer.uniform(scale=0.01,\
  seed=1)):
  hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
    name='hidLayer')(X)
  oLayer = C.layers.Dense(output_dim, activation=None,
   name='outLayer')(hLayer)
nnet = oLayer
model = C.ops.softmax(nnet)
```

The Python with statement is a syntactic shortcut to apply a set of common arguments to multiple functions. The demo uses tanh activation on the hidden layer nodes; a common alternative is the sigmoid function. Notice that there's no activation applied to the output nodes. This is a quirk of CNTK because the CNTK training function expects raw, un-activated values. The nnet object is just a convenience alias. The model object has softmax activation so it can be used after training to make predictions. Because Python assigns by reference, training the nnet object also trains the mode object.

Training the Neural Network

The neural network is prepared for training with:

```
tr_loss = C.cross_entropy_with_softmax(nnet, Y)
tr_clas = C.classification_error(nnet, Y)
max_iter = 5000
batch_size = 10
learn_rate = 0.005
learner = C.sgd(nnet.parameters, learn_rate)
trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])
```

The tr_loss ("training loss") object tells CNTK how to measure error when training. An alternative to cross entropy with softmax is squared error. The tr_clas ("training classification error") object can be used to automatically compute the percentage of incorrect predictions during or after training.

The values for the maximum number of training iterations, the number of items in a batch to train at a time, and the learning rate, are all free parameters that must be determined by trial and error. You can think of the learner object as an algorithm, and the trainer object as the object that uses the learner to find good values for the neural network's weights and biases.

Figure 3 **Demo Program Structure**

```
# cleveland_bnn.py
# CNTK 2.3 with Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

import numpy as np
import cntk as C

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  x_strm = C.io.StreamDef(field='symptoms', shape=input_dim,
   is_sparse=False)
  y_strm = C.io.StreamDef(field='disease', shape=output_dim,
    is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order, \
    max_sweeps=sweeps)
  return mb_src

# =================================================================

def main():
  print("\nBegin binary classification (two-node technique) \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  input_dim = 18
  hidden_dim = 20
  output_dim = 2

  train_file = ".\\Data\\cleveland_cntk_twonode.txt"

  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 18-20-2 tanh-softmax NN ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,\
    seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  nnet = oLayer
  model = C.ops.softmax(nnet)

  # 2. create learner and trainer
  print("Creating a cross entropy batch=10 SGD LR=0.005 Trainer ")
  tr_loss = C.cross_entropy_with_softmax(nnet, Y)
  tr_clas = C.classification_error(nnet, Y)

  max_iter = 5000
  batch_size = 10
  learn_rate = 0.005
  learner = C.sgd(nnet.parameters, learn_rate)
  trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])

  # 3. create reader for train data
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  heart_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  # 4. train
  print("\nStarting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size, \
      input_map=heart_input_map)
    trainer.train_minibatch(curr_batch)
    if i % int(max_iter/10) == 0:
      mcee = trainer.previous_minibatch_loss_average
      macc = (1.0 - trainer.previous_minibatch_evaluation_average) \
        * 100
      print("batch %4d: mean loss = %0.4f, accuracy = %0.2f%% " \
        % (i, mcee, macc))
  print("\nTraining complete")

  # 5. evaluate model using all data
  print("\nEvaluating accuracy using built-in test_minibatch() \n")
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)
  heart_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
  num_test = 297
  all_test = rdr.next_minibatch(num_test, input_map=heart_input_map)
  acc = (1.0 - trainer.test_minibatch(all_test)) * 100
  print("Classification accuracy on the %d data items = %0.2f%%" \
    % (num_test,acc))

  # (could save model here)
  # (use trained model to make prediction)

  print("\nEnd Cleveland Heart Disease classification ")

# =================================================================

if __name__ == "__main__":
  main()
```

A reader object is created with these statements:

```
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
heart_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src
}
```

If you examine the create_reader definition in **Figure 3**, you'll see that it specifies the tag names ("symptoms" and "disease") used in the data file. You can consider create_reader and the code to create a reader object as boilerplate code for neural binary classification problems. All you have to change are the tag names and the name of the mapping dictionary (heart_input_map).

After everything is prepared, training is performed like so:

```
for i in range(0, max_iter):
  curr_batch = rdr.next_minibatch(batch_size, \
    input_map=heart_input_map)
  trainer.train_minibatch(curr_batch)
  if i % int(max_iter/10) == 0:
    mcee = trainer.previous_minibatch_loss_average
    macc = (1.0 - trainer.previous_minibatch_evaluation_average) \
      * 100
    print("batch %4d: mean loss = %0.4f, accuracy = %0.2f%% " \
      % (i, mcee, macc))
```

An alternative to training with a fixed number of iterations is to stop training when loss/error drops below a threshold. It's important to display loss/error during training because training failure is the rule rather than the exception. Cross-entropy error is a bit difficult to interpret directly, but you want to see values that tend to get smaller. Instead of displaying average classification loss/error, the demo computes and prints the average classification accuracy, which is a more natural metric in my opinion.

Evaluating and Using the Model

After a network has been trained, you'll usually want to determine the loss/error and classification accuracy for the entire dataset that was used for training. The demo evaluates overall classification accuracy with:

```
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=False, sweeps=1)
heart_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src
}
num_test = 297
```

```
all_test = rdr.next_minibatch(num_test, input_map=heart_input_map)
acc = (1.0 - trainer.test_minibatch(all_test)) * 100
print("Classification accuracy on the %d data items = %0.2f%%" \
  % (num_test,acc))
```

A new data reader is created. Notice that unlike the reader used for training, the new reader doesn't traverse the data in random order, and that the number of sweeps is set to 1. The heart_input_map dictionary object is recreated. A common mistake is to try and use the original object—but the rdr object has changed, so you need to recreate the mapping. The test_minibatch function returns the average classification error for its mini-batch argument, which in this case is the entire dataset.

The demo program doesn't compute the loss/error for the entire dataset. You can use the previous_minibatch_loss_average function, but you have to be careful not to perform an additional training iteration, which would change the network.

After training, or during training, you'll usually want to save the model. In CNTK, saving looks like:

```
mdl_name = ".\\Models\\cleveland_bnn.model"
model.save(mdl_name)
```

This saves using the default CNTK v2 format. An alternative is to use the Open Neural Network Exchange (ONNX) format. Notice that you'll generally want to save the model object (with softmax activation) rather than the nnet object.

From a different program, a saved model could be loaded into memory along the lines of:

```
mdl_name = ".\\Models\\cleveland_bnn.model"
model = C.ops.functions.Function.load(mdl_name)
```

After loading, the model can be used as if it had just been trained.

The demo program doesn't use the trained model to make a prediction. You can write code like this:

```
unknown = np.array([0.5555, -1, ... ], dtype=np.float32)
predicted = model.eval(unknown)
```

The result returned into variable predicted would be a 1x2 matrix with values that sum to 1.0, for example [[0.2500, 0.7500]]. Because the second value is larger, the result would map to (0, 1), which in turn would map to "no disease."

Wrapping Up

Most deep learning code libraries perform neural network binary classification using the one-node technique. Using this approach, the values of the variable to predict are encoded as either 0 or 1. The output dimension would be set to 1 instead of 2. You'd have to use binary cross-entropy error instead of ordinary cross-entropy error. CNTK doesn't have a built-in classification error function that works with one node, so you'd have to implement your own function from scratch. When training, less information is typically gained on each iteration (although training is a bit faster), so you'll usually have to train a one-node model for more iterations than when using the two-node technique. For these reasons, I prefer using the two-node technique for neural binary classification. ■

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

Thanks to the following Microsoft technical experts who reviewed this article: *Chris Lee, Ricky Loynd, Ken Tran*

DAVID S. PLATT

This Is Not a Drill

You've no doubt heard about the false alarm on Jan. 13 that caused much of the state of Hawaii to duck and cover. Just what you need while evading the cold winter and enjoying a piña colada on the beach: a text shouting "BALLISTIC MISSILE THREAT INBOUND TO HAWAII. SEEK IMMEDIATE SHELTER. THIS IS NOT A DRILL."

The alarm was false, of course. The investigation revealed that the employee who triggered the alert mistakenly thought the event was real and not a drill. But the incident brought to light something else: the sad state of the software used by the emergency management office.

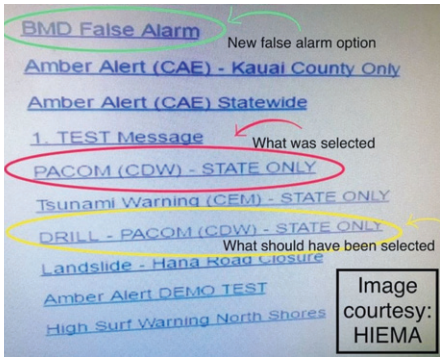


Figure 1 **The Alert Menu Screen Initially Provided by the Hawaii Governor's Office**

You can see that terrible design in **Figure 1**, which news outlets say depicts the emergency management interface in use during the event. The official should have selected "DRILL – PACOM (CDW) – STATE ONLY," the seventh link from the top, for the regular internal test that occurs on each shift. For whatever reason, the guy clicked on "PACOM (CDW) – STATE ONLY," two links above it, triggering the real thing.

Of course there was a confirmation box: "Are you sure?" Like every user everywhere in the history of the universe, this user cruised right through on autopilot: "Yes."

Oops.

Ideally we want software operations to be reversible. That's fine for cutting and pasting text. But in the real world, we encounter operations that, by their very nature, are not reversible—ejecting from a fighter plane, say, or amputating a leg. This alert falls into that category. While the alert was eventually rescinded, the damage was done—to frightened tourists; to cardiac patients; to the confidence of the population; to the innocence of children; to stockpiles of premium booze, hastily chugged. How do we design applications to handle irreversible operations, preventing false alarms without impeding the real alarms?

First, we design for cognitive clarity. I can discern no pattern to the arrangement of the items in **Figure 1**, with test alerts and real ones randomly comingled. Try putting them on different sides of the page, in different tables, maybe in different colors. Consistency in nomenclature would help as well: What's the difference between a drill and a demo test or a test message? Change the nomenclature to always use, say, DRILL and REAL.

Next, we design to avoid slips. A slip happens when the user doesn't type or click as precisely as the application requires, thereby activating something he didn't intend. That's why we don't put an ejection seat button right next to a radio button: an air pocket or a sneeze bumps the user's finger half a centimeter too far, and kablooey!

You can see these ideas in the design of real ejection seats (ejectionsite.com). The trigger is on a separate handle and uniquely marked (cognitive clarity), located away from other controls, and triggered by a motion used for nothing else (slip prevention). Our irreversible apps should use similar techniques.

Now, what about the confirmation box? It's supposed to make the user stop and think, but it never does. Ever. Anywhere. On anything. The user clicks "Yes" three times a day, and "No" essentially never, so the affirmation becomes automatic. Because this does not protect our expensive and dangerous irreversible operation, what would?

This operation is not split-second critical, as an ejection seat would be. There is time for another opinion, which the app should require: "Alice, I'm sending you a drill request." "Bob, I've received your drill request and I'm authorizing it." Perhaps require the users to type "drill" or "real" correctly, further differentiating the operations.

Similar multi-person control is required for launching these deadly missiles. Anyone wishing to explore further can read the book "Command and Control," by Eric Schlosser (Penguin, 2013).

The application UI shown in **Figure 1** would constitute gross negligence. If one of my students submitted a project like this, I'd flunk his sorry ass so fast he'd change his major to Sanskrit.

As I wrote in my very first Don't Get Me Started column in February 2010 (msdn.com/magazine/ee309884): "Humans are not going to stop being human any time soon, no matter how much you might wish they would evolve into something more logical. Good applications recognize this, and adjust to their human users, instead of hoping, futilely, for the opposite." Do the missiles actually have to fly before we learn this lesson? ■

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.*

Introducing **RavenDB 4.0**

Your Fully Transactional NoSQL Database

The amount of data your organization needs to handle is rising at an ever-increasing rate. We developed RavenDB 4.0 so you can handle this tougher challenge, and do it while improving the performance of your application at the same time.

RavenDB 4.0 is the premiere choice of Fortune 500 companies because it offers you the best of both worlds. It is a NoSQL database that is fully transactional. You can get the benefits of using next generation NoSQL while keeping the best value that relational databases offer. Our open source document database has reached over 100,000 writes and half a million reads per second using low cost commodity hardware. We ramp up your performance right up until you hit the limits of your hardware.

It's easy to set up a RavenDB 4.0 database cluster and even easier to use. The ramp up time to become an expert is quick. Our RQL query language is similar to SQL, and very familiar with what you are used to. The management studio GUI makes using RavenDB 4.0 convenient for both developers and non-developers. We've automated many of the functions normally assigned to DBAs, freeing up time and resources for other priorities.

Your data never sleeps, so why should your database? Our multi-node data cluster eliminates the single point of failure vulnerability that comes with the relational SQL approach. You can create a distributed data cluster with multiple nodes, all holding a copy of your database and replicating to each other in real time. This gives you high-availability while effectively balancing load, reducing latency, and maximizing your performance. Your customers have the technology to seek information about your business 24 hours a day. With a RavenDB 4.0 data cluster you have the ability to give it to them anytime they want.

Our in-house storage engine pushes performance to the max and minimizes the need for third party plugins. With everything at your fingertips, RavenDB 4.0 gives you all the tools to take your database technology to the next level.

Enjoy Top Performance

Fully Transactional
like a relational database

Easy to Install
Easy to Use

Works Well Alongside SQL Solutions

Scale Up Fast with a High Availability Data Cluster

All-in-One Database
Fewer third party plugins
puts everything at your fingertips

Highly Automated Features
to reduce overhead

Grab a FREE License

3-node database cluster with GUI interface,
3 cores and 6 GB RAM

www.ravendb.net/free

Hibernating Rhinos

RAVENDB 4.0
DATA MADE FASTER