THE MICROSOFT JOURNAL FOR DEVELOPERS

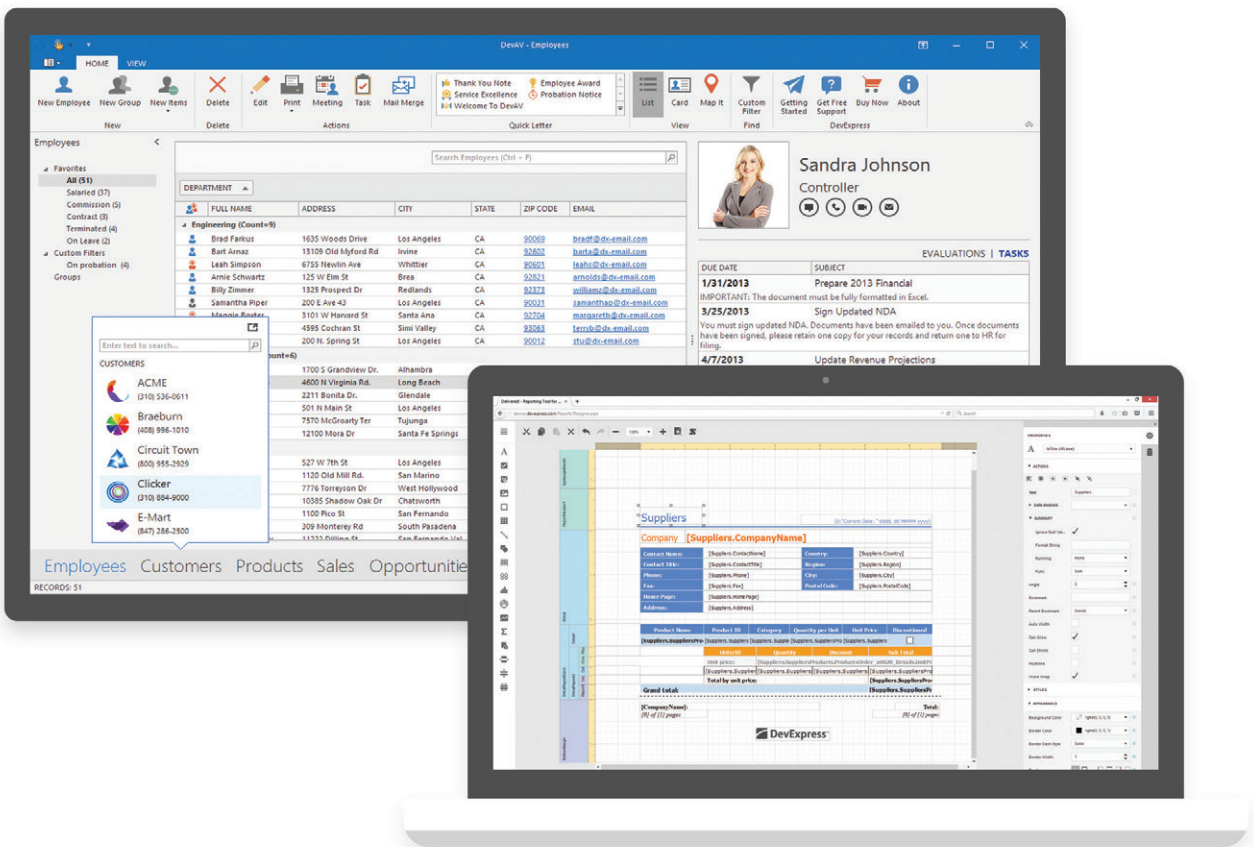
MARCH 2019 VOL 34 NO 3

msdn magazine

DevExpress®

Your Next Great App Starts Here

From apps that replicate the look and feel of Microsoft Office® to high-powered data mining and decision support systems for your enterprise, DevExpress UI components for desktops, the web and mobile world will help you build your best, without limits or compromise.



Experience the DevExpress Difference
Download Your Free 30-Day Trial Today

devexpress.com/trial

THE MICROSOFT JOURNAL FOR DEVELOPERS

MARCH 2019 VOL 34 NO 3

msdn magazine

C# Validation Logic with Blazor.....14

COLUMNS

Microsoft

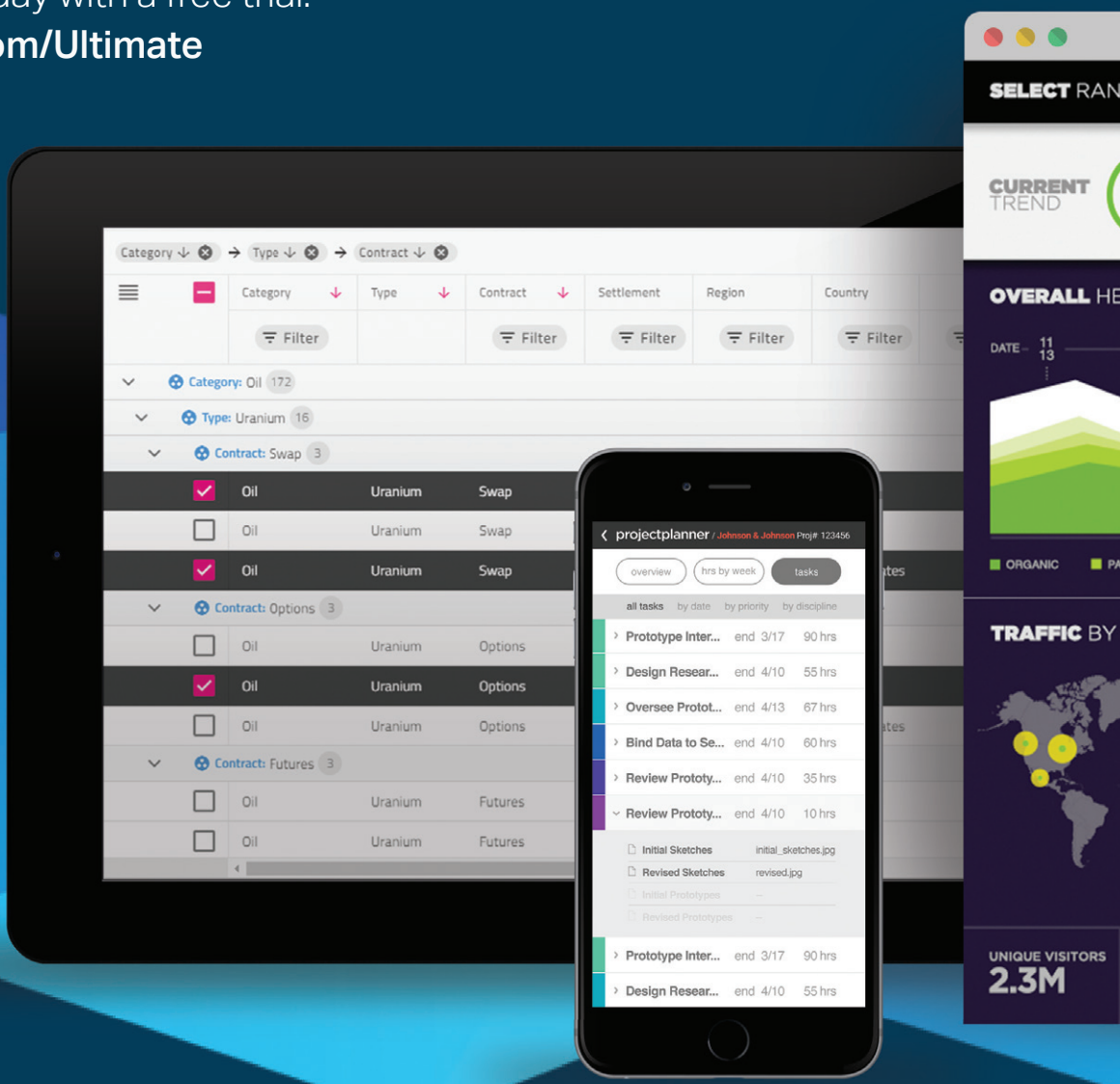
INFRAGISTICS

Faster Paths to Amazing Experiences

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

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

Get started today with a free trial:
Infragistics.com/Ultimate



To speak with sales or request a product demo with a solutions consultant call 1.800.231.8588

New Release

Infragistics Ultimate 18.2

- Fastest **grids** & **charts** on the market for the Angular developer
- The most complete Microsoft Excel and Spreadsheet solution for creating dashboards and reports without ever installing Excel
- An end-to-end design-to-code platform with Indigo.Design
- Best-of-breed charts for financial services

msdn magazine

MARCH 2019 VOLUME 34 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., 6300 Canoga Avenue, Suite 1150, Woodland Hills, CA 91367. Periodicals postage paid at Woodland Hills, CA 91367 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, File 2272, 1801 W.Olympic Blvd., Pasadena, CA 91199-2272, email MSDNmag@1105service.com or call 866-293-3194 or 847-513-6011 for U.S. & Canada; 00-1-847-513-6011 for International, fax 847-763-9564. 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, Inc.
6300 Canoga Avenue, Suite 1150, Woodland Hills 91367
1105media.com

MEDIA KITS Direct your Media Kit requests to Chief Revenue Officer Dan LaBianca, 972-687-6702 (phone), 972-687-6799 (fax), dlabianca@Converge360.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, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Jane Long, Merit Direct.
Phone: (913) 685-1301
Email: 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
2121 Alton Pkwy., Suite 240, Irvine, CA 92606

Corporate Office (weekdays, 8:30 a.m.-5:30 p.m. PT)
Telephone 818-814-5200; Fax 818-734-1522
6300 Canoga Ave., Suite 1150, Woodland Hills, CA 91367

The opinions expressed within the articles and other contentsherein do not necessarily express those of the publisher.

Microsoft

LEADTOOLS®

MEDICAL WEB VIEWER

Now with 3D

The LEADTOOLS Medical Web Viewer is an OEM-ready web application that provides a platform-independent solution to display DICOM studies for all medical disciplines and modalities. The fully customizable application is lightweight, yet includes powerful features including 3D reconstruction, hanging protocols, touch-enabled window-leveling, annotations and more, all in a web-based zero-footprint solution.

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

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

MICHAEL DESMOND

Naked Ambition

Object-oriented programming (OOP) has stood as an aspirational target for developers going back to Alan Kay and his Smalltalk programming language in the 1980s. It's hard to argue the benefits of an object-oriented approach that features modular, reusable code to speed development, ease maintenance and boost productivity. And over the decades, we've seen object orientation adopted by many of the most widely used programming languages and frameworks, from C++ to Java to C#.

But as Ted Neward pointed out in his The Working Programmer column at the beginning of the year (msdn.com/magazine/mt848703), the ultimate promise of OOP remains to an extent unfulfilled. In that column, he introduced Naked Objects (nakedobjects.org), an architectural pattern developed by Richard Pawson that's dedicated to the concept of behaviorally complete objects. Unlike most mainstream OOP implementations, Naked Objects encapsulates all the business logic as methods on domain entities. The developer doesn't need to define views or controllers, and the UI is generated automatically from metadata gathered, via reflection, from objects at run time.

"I think every senior developer, once they discover reflection, dreams or imagines building a system like this, which is why I think Naked Objects is so instructive as a model to examine," Neward says. "It's far more viable than most developers realize, and I think part of what hurts it is that a lot of folks simply don't know it exists."

Naked Objects got its start as a Ph.D. thesis by Pawson, who was inspired to explore deep object orientation after conversations with Alan Kay in the early 1990s. Pawson saw that despite the clear benefits, few commercial systems were being developed in OOP. And, he says, "Those that were doing it weren't getting the kind of dramatic benefits that OOP promised. I was keen to understand why."

His conclusion: Theoretical gains from a paradigm based on behaviorally complete objects were offset by all the work needed to create layers of code above and below the domain model in a real application. In short, "the benefit of behaviorally complete domain objects was lost in the added complexity," Pawson says.

Naked Objects addresses these disconnects, while eliminating the effort of creating and maintaining a UI layer, says Pawson. "You get the benefit of a UI that is completely consistent in its operation, even across a very complex domain model."

You won't find a lot of Naked Objects deployments, but they do exist. The Irish Department of Social Protection has built its enterprise systems on Naked Objects since 2004. Pawson says the department has moved through three radically different architectures without any change to the underlying domain code—a Windows Presentation Foundation thick-client model, a thin-client model running on ASP.MVC, and currently a single-page application written in Angular and communicating with the server via a RESTful API.

> The Irish Department of Social Protection has built its enterprise systems on Naked Objects since 2004.

"Given the complexity of its business model—4,000-plus domain classes!—that's an extraordinary accomplishment," he says.

Is Naked Objects likely to be on tap for your next project? Probably not. But as Pawson notes, it offers lessons for developers no matter the engagement.

"Naked Objects makes it a lot easier to adhere to a number of principles of sound software design," says Pawson. "Domain-driven design, separation of concerns, modularity and, above all, polymorphism."

Those lessons are sure to be present in Neward's ongoing Naked Objects series. Be sure to check out his latest in this month's issue.

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

File Format APIs

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

Try risk free - 30 day trial

ASPOSE

Download a Free Trial at

https://downloads.aspose.com

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

Deliver efficient applications to Create, Draw, Manipulate or Convert image file formats.

Aspose.Tasks

Develop high performance apps to Create, Edit or Convert Microsoft Project® document formats.

- Aspose.Diagram
- Aspose.Note
- Aspose.3D
- Aspose.CAD
- Aspose.HTML
- Aspose.GIS

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

JULIE LERMAN

A Peek at the EF Core Cosmos DB Provider Preview, Part 2

In the January 2019 Data Points column, I presented a first look at the Cosmos DB Provider for EF Core. This provider is still in preview and is expected to go live with the EF Core 3.0 release, so it's a good time to prepare for it in advance.

In that first part, you read about why there's a NoSQL provider for an ORM and learned how to perform some basic read and write actions for individual objects and their related data as defined by a simple model. Writing code that leverages this provider isn't much different than working with the more familiar relational database providers.

You also learned how EF Core can create a database and containers on the fly for pre-existing Azure Database accounts and how to view the data in the cloud using the Cosmos DB extension for Visual Studio Code.

I took a sidetrack in my February 2019 column (msdn.com/magazine/mt833267) to take a look at the MongoDB API of Azure Cosmos DB, although this isn't related to EF Core. Now I'll return to my previous subject to share some of the other interesting discoveries I made when exploring the EF Core Cosmos DB provider.

In this column, you'll learn about some of the provider's more advanced features, such as configuring the DbContext to change how EF Core targets Cosmos DB database containers, realizing embedded documents with owned entities and using EF Core logging to see the SQL along with other interesting processing information generated by the provider.

More About Containers and EF Core Mappings

Containers, also known as collections in the Cosmos DB SQL and Mongo DB APIs, are schema-agnostic groupings of items that are the "fundamental units of scalability" for Cosmos DB. That is, you can define throughput per container and a container can scale and be replicated as a unit. As your data grows, designing models and how they align with containers will impact performance and cost. If you've used EF or EF Core, you'll be familiar with the default that one DbSet<TEntity> maps to one table in a relational database. Creating a separate Cosmos DB container for each DbSet could be an expensive default, however. But the default, as you learned in Part 1, is that all of the data for a DbContext maps to a single container. And the convention is that the container has the same name as the DbContext.

The EF Core Cosmos DB Provider is in preview. All information is subject to change.

Code download available at msdn.com/magazine/0319magcode.

Let's take a look at defaults and see which ones you can control with EF Core.

In the previous article, I let EF Core trigger a new database and container to be created on the fly. I already had an Azure account, targeted to the SQL API (which is what EF Core uses). Geo-redundancy is enabled by default, but I've only configured my account to use a single datacenter in the Eastern United States. Therefore, by default, multi-region writes are disabled. So whatever databases I add to this account, and any containers to those databases, will follow those overarching specs controlled by the account.

In the first column, I had a DbContext named ExpanseDbContext. When configuring the ExpanseDbContext to use the Cosmos provider, I specified that the database name should be ExpanseCosmosDemo:

```
optionsBuilder.UseCosmos(endpointstring,accountkeystring, "ExpanseCosmosDemo")
```

The first time my code called Database.EnsureCreated on an instance of ExpanseDbContext, the ExpanseCosmosDemo database was created along with the default container, called ExpanseDbContext, following the convention to use the name of the DbContext class.

The container was created using the Azure Cosmos DB defaults shown in **Figure 1**. Not shown in the figure is the indexing policy configuration using the default, which is "consistent."

These settings can't be affected by EF Core. You can modify them in the portal, using the Azure CLI or an SDK. This makes sense because EF Core's role is to read and write data. But one thing you

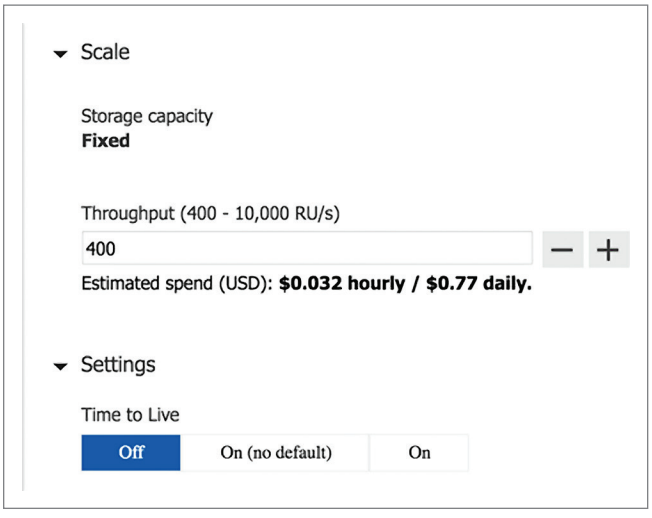


Figure 1 **Azure Cosmos DB Defaults for Creating a Container**

DevExpress®

From Desktops to Web and Mobile

Your Next Great App Starts Here

Experience the DevExpress difference and see why your peers consistently vote our products #1. With our Universal Subscription, you will build your best, see complex software with greater clarity, increase your productivity and create stunning applications for Windows, Web and your Mobile world.

DevExpress Universal ships with 500+ UI controls.
It also includes our royalty-free reporting and dashboard platform.

WIN ASP MVC WPF UWP JS

Download your free 30-day trial today.

devexpress.com/try

can affect with EF Core is container names and mapping entities to be stored in different containers.

You can override the default container name with the HasDefaultContainerName method in OnConfiguring. For example, the following will use ExpanseDocuments as the default name instead of ExpanseDbContext:

```
modelBuilder.HasDefaultContainerName("ExpanseDocuments");
```

If you've determined that you want to split data into different containers, you can map a new container name for particular entities. Here's an example that specifies the Ship entity from the previous article into a container called ExpanseShips:

```
modelBuilder.Entity<Ship>().ToContainer("ExpanseShips");
```

You can target as many entities to a single container as you want. The default container already demonstrates this. But you could use ToContainer("ExpanseShips") with other entities, as well, if you wanted.

Figure 2 **The Expanse Classes**

```
public class Consortium
{
  public Consortium()
  {
    Ships=new List<Ship>();
    Stations=new List<Station>();
  }
  public Guid ConsortiumId { get; set; }
  public string Name { get; set; }
  public List<Ship> Ships{get;set;}
  public List<Station> Stations{get;set;}
  public Origin Origin{get;set;}
}
public class Planet
{
  public Guid PlanetId { get; set; }
  public string PlanetName { get; set; }
}
public class Ship
{
  public Guid ShipId {get;set;}
  public string ShipName {get;set;}
  public int PlanetId {get;set;}
  public Origin Origin{get;set;}
}
public class Origin
{
  public DateTime Date{get;set;}
  public String Location{get;set;}
}
```

Figure 3 **A Ship Document with an Origin Sub-Document Embedded**

```
{
  "ShipId": "e5d48ffd-e52e-4d55-97c0-cee486a91629",
  "ConsortiumId": "60ccb22d-4422-45b2-a54a-71fa240435b3",
  "Discriminator": "Ship",
  "PlanetId": 0,
  "ShipName": "Nathan Hale 3rd",
  "id": "c2bdd90f-cb6a-4a3f-bacf-b0b3ac191662",
  "Origin": {
    "ShipId": "e5d48ffd-e52e-4d55-97c0-cee486a91629",
    "Date": "2019-01-22T11:40:29.117453-05:00",
    "Discriminator": "Origin",
    "Location": "Earth"
  },
  "_rid": "cgEVAKklUPgCAAAAAAAAAA==",
  "_self": "dbs/cgEVAA==/colls/cgEVAKklUPg=/docs/
           cgEVAKklUPgCAAAAAAAAAA==/",
  "_etag": "\"0000a43b-0000-0000-0000-5c47477d0000\"",
  "_attachments": "attachments/",
  "_ts": 1548175229
}
```

What happens when you add a new container to an existing database in this way? As I noted in Part 1, the only way to have EF Core create a database or container is by calling context.Database.EnsureCreated. EF Core will recognize what does and doesn't already exist and create any new containers as needed.

If you change the default container name, EF will create the new container and will work with that container going forward. But any data in the original container will remain there.

Because Azure Cosmos DB doesn't have the ability to rename an existing container, the official recommendation is to move the data into the new collection, perhaps with a bulk executor library, such as the one at bit.ly/2RbpTvp. The same holds true if you change the mapping for an entity to a different container. The original data won't be moved and you'll be responsible for ensuring that the old items are transferred. Again, it's probably more reasonable to do that one-time move outside of EF Core.

I also tested out adding graphs of Consortium with Ships where the documents would end up in separate containers in the database. When reading that data, I was able to write a query for Consortia that eager-loaded its ship data, for example:

```
context.Consortia.Include(c=>c.Ships).FirstOrDefault()
```

EF Core was able to retrieve the data from the separate containers and reconstruct the object graph.

Owned Entities Get Embedded Within Parent Documents

In Part 1, you saw that related entities were stored in their own documents. I've listed the Expanse classes in **Figure 2** as a reminder of the example model. When I built a graph of a Consortium with Ships, each object was stored as a separate document with foreign keys that allow EF Core or other code to connect them back up again. That's a very relational concept, but because consortia and ships are unique entities that have their own identity keys, this is how EF Core will persist them. But EF Core does have an understanding of document database and embedded documents, which you can witness when working with owned entities. Notice that the Origin type doesn't have a key property and it's used as a property of both Ship and of Consortium. It will be an owned entity in my model. You can read more about the EF Core Owned Entity feature in my April 2018 Data Points article at msdn.com/magazine/mt846463.

In order for EF Core to comprehend an owned type so that it can map it to a database, you need to configure it either as a data annotation or (always my preference) a fluent API configuration. The latter happens in the DbContext OnConfiguring method as I'm doing here:

```
modelBuilder.Entity<Ship>().OwnsOne(s=>s.Origin);
modelBuilder.Entity<Consortium>().OwnsOne(s=>s.Origin);
```

Here's some code for adding a new Ship, along with its origin, to a consortium object:

```
consortium.Ships.Add(new Ship{ShipId=Guid.NewGuid(),ShipName="Nathan Hale 3rd",
                              Origin= new Origin {Date=DateTime.Now,
                              Location="Earth"}});
```

When the consortium is saved via the ExpanseContext, the new ship is also saved into its own document.

Figure 3 displays the document for that Ship with its Origin represented as an embedded document. A document database

doesn't need a sub-document to have a foreign key back with its parent. However, the EF Core logic for persisting owned entities does require the foreign key (handled by EF Core Shadow Properties) in order to persist owned entities in relational databases. Therefore, it leverages its existing logic to infer the ShipId property within the Origin sub-document.

EF Core also has the ability to map owned collections with the OwnsMany mapping. In this case, you'd see multiple sub-documents within the parent document in the database.

There's a gotcha that will be fixed in EF Core 3.0.0 preview 2. EF Core currently doesn't understand null owned entity properties. The other database providers will throw a runtime exception if you attempt to add an object with a null owned entity property, a behavior you can read about in the previously mentioned April 2018 column. Unfortunately, the Cosmos DB provider doesn't prevent you from adding objects in this state, but it's not able to materialize objects that don't have the owned entity property populated. Here's the exception that was raised when I encountered this problem:

```
"System.InvalidCastException: Unable to cast object of type
'Newtonsoft.Json.Linq.JValue' to type 'Newtonsoft.Json.Linq.JObject'."
```

So if you see that error when trying to query entities that have owned type properties, I hope you'll remember that it's likely a null owned type property causing the exception.

Logging the Provider Activity

EF Core plugs into the .NET Core logging framework, as I covered in my October 2018 column (msdn.com/magazine/mt830355). Shortly after that article was published, the syntax for instantiating the LoggerFactory was simplified, although the means of using categories and log levels to determine what should get output in the logs didn't change. I reported the updated syntax in a blog post, "Logging in EF Core 2.2 Has a Simpler Syntax—More Like ASP.NET Core" (bit.ly/2UdSkuI).

When EF Core interacts with the Cosmos DB provider, it also shares details with the logger. This means you can see all of the same types of information in the logs that you can with other providers.

Keep in mind that CosmosDB doesn't use SQL for inserting, updating and deleting, as you're used to doing with relational databases. SQL is used for queries only, so SaveChanges won't show SQL in the logs. However, you can see how EF Core is fixing up the objects, creating any needed IDs, foreign keys and discriminators. I was able to see all of this information when logging all of the categories tied to the Debug LogLevel, rather than only filtering on the database commands.

Here's how I configured my GetLoggerFactory method to do that. Notice the AddFilter method. Rather than passing a category into the first parameter, I'm using an empty string, which gives me every category:

```
private ILoggerFactory GetLoggerFactory()
{
  IServiceCollection serviceCollection = new ServiceCollection();
  serviceCollection.AddLogging(builder =>
        builder.AddConsole()
              .AddFilter("" , LogLevel.Debug));
  return serviceCollection.BuildServiceProvider()
         .GetService<ILoggerFactory>();
}
```

If I'd wanted to filter on just the SQL commands, I'd have passed DbLoggerCategory.Database.Command.Name to give the correct string for just those events instead of an empty string. This relayed a lot of logging messages when inserting a few graphs and then executing a single query to retrieve some of that inserted data. I'll include the full output and my program in the download that accompanies this column.

Some interesting tidbits from those logs include this information about adding shadow properties where you can, in the case of this provider, see the special Discriminator property being populated:

```
dbug: Microsoft.EntityFrameworkCore.Model[10600]
      The property 'Discriminator' on entity type 'Station' was created
in shadow state because there are no eligible CLR members with a matching
name.
```

If you're saving data, after all of that fix-up is performed, you'll see a log message that SaveChanges is starting:

```
debug: Microsoft.EntityFrameworkCore.Update[10004]
       SaveChanges starting for 'ExpanseContext'.
```

This is followed by messages about DetectChanges being called. The provider will use internal API logic to add, modify or remove the document in the relevant collection, but you won't see any particular logs about that action. However, after these actions complete, the logs will relay typical post-save steps such as the context updating the state of the object that was just posted:

```
dbug: Microsoft.EntityFrameworkCore.ChangeTracking[10807]
      The 'Consortium' entity with key '{ConsortiumId: a4b0405e-a820-4806-
8b60-159033184cf1}' tracked by 'ExpanseContext' changed from 'Added' to
'Unchanged'.
```

If you're executing a query, you'll see a number of messages as EF Core works out the query. EF Core starts by compiling the query and then massages it until it arrives at the SQL that gets sent to the database. Here's a log message showing the final SQL:

```
dbug: Microsoft.EntityFrameworkCore.Database.Command[30000]
      Executing Sql Query [Parameters=[]]
      SELECT c
      FROM root c
      WHERE (c["Discriminator"] = "Consortium")
```

Waiting for Release

The EF Core Cosmos DB provider preview is available for EF Core 2.2+. I worked with EF Core 2.2.1 and and then, in order to see if I noticed any changes, switched to the unreleased EF Core packages in the latest preview of EF Core 3, version 3.0.0-preview.18572.1.

EF Core 3 is on the same release schedule as .NET Core 3.0, but the latest information about the timing only says "sometime in 2019." The official release of Preview 2 was announced at the end of January 2019 in the blog post at bit.ly/2UsNBp6. If you're interested in this support for Azure Cosmos DB, I recommend trying it out now and helping the EF team uncover any problems to make it a more viable provider for you when it does get released. ■

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 Microsoft technical expert for reviewing this article: *Andriy Svyryd*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

NEW IN 2019!

On-Demand Session Recordings for One Year!

Get access to all sessions (not including Hands-On Labs or Workshops) at each show for a year. Learn more at vslive.com.

Developer Training Conferences and Events

Choose VSLive! For:

- ✔ In-depth developer training
- ✔ Unparalleled networking
- ✔ World-class speakers
- ✔ Exciting city adventures

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360 An 1105 MEDIA company

JOIN US IN 2019!

LAST CHANCE TO REGISTER!

March 3-8

Bally's Hotel & Casino

Visual Studio Live! kicks off 2019 in the heart of Las Vegas with 6 days of hard-hitting Hands-On Labs, workshops, 60+ sessions, expert speakers and several networking opportunities included! Register to join us today!

REGISTER NOW!
vslive.com/lasvegas

April 22-26

Hyatt Regency

For the first time in our 20-year history, Visual Studio Live! is heading down south to New Orleans for intense developer training, bringing our hard-hitting sessions, well-known coding experts and unparalleled networking to the Big Easy!

REGISTER NOW!
vslive.com/neworleans

June 9-13

Hyatt Regency Cambridge

Join Visual Studio Live! for an amazing view of Beantown, bringin g our infamous speakers for intense developer training, Hands-On Labs, workshops, sessions and networking adventures to the Northeast.

REGISTER NOW!
vslive.com/boston

June 18-19

Microtek Training Center San Jose

Develop and ASP.NET Core 2.x Service and Website with EF Core 2.x in two days with Visual Studio Live!'s training seminar in San Jose, CA. Expand your knowledge and accelerate your career today.

REGISTER NOW!
vslive.com/sanjose

August 12-16

Microsoft HQ

Join our Visual Studio Live! experts at the Mothership for 5 days of developer training and special Microsoft perks unique to our other show locations. Plus, we are adding the ever-so popular full-day Hands-On Labs to the agenda in Redmond for the first time this year!

REGISTER NOW!
vslive.com/microsofthq

Sept. 29-Oct. 3

Westin Gaslamp Quarter

Head to the heart of the San Diego Gaslamp District with Visual Studio Live! this Fall as we immerse ourselves with all things for developers, including several workshops, sessions and networking opportunities to choose from.

REGISTER NOW!
vslive.com/sandiego

October 6-10

Swissotel

Head to the Windy City and join Visual Studio Live! this October for 5 days of unbiased, developer training and bringing our well-known Hands-On Labs to the city for the first time.

REGISTER NOW!
vslive.com/chicago

November 17-22

Royal Pacific Resort at Universal

Visual Studio Live! Orlando is a part of Live! 360, uniquely offering you 6 co-located conferences for one great price! Stay ahead of the current trends and advance your career – join us for our last conference of the year!

DETAILS COMING SOON!

TED NEWARD

Coding Naked: Naked Properties

Welcome back, NOFers. (I've decided that sounds better than calling those who use naked objects "naked coders.") In the last piece, I started building out the domain model for my conference system, which allowed me to store speakers, but it's a pretty plain-vanilla setup thus far. I haven't done any of the sorts of things you'd normally need to do, like verifying that first or last names aren't empty, or supporting a "topics" field that's one of a bound set of options, and so on. All of these are reasonable things to want to support in a UI (as well as a data model), so if this "naked" approach is going to be used for real-world scenarios, it needs to be able to do them, as well.

Fortunately, the NOF designers knew all that.

Naked Concepts

Let's go back and talk about how NOF handles this in the general case before I get into the specifics.

Remember, the goal of NOF is to keep from having to write UI code that could otherwise be signaled using some aspect of the domain objects themselves, and the best way to do that sort of signaling is through the use of custom attributes. In essence, you use NOF custom attributes to annotate various elements of the domain object—properties and methods, for the most part—and the NOF client understands, based on the presence of the attribute, or data contained inside the attribute, that it has to customize the UI for that object in some manner. Note that NakedObjects doesn't need to actually define many of these custom attributes, as they come "for free" from the System.ComponentModel namespace in the standard .NET distribution. Reusability!

> Specifying non-zero names is one of the easiest validations to apply, because it's a static one.

However, sometimes it's not quite as simple as "this should always be the case." For example, if certain properties have to be disabled based on the internal state of the object (such as an "on-parental-leave" property that needs to be disabled if an employee has no spouse or children), then code will need to be executed, and that's something a custom attribute can't provide. In those situations, NOF relies on convention: specifically, NOF will look for particularly named methods on the class. If the parental-leave property is named OnLeave, then the method that NOF will execute to determine whether to disable the OnLeave property would be called DisableOnLeave.

Let's see how this works out in practice.

Naked Speakers, Redux

Currently, the Speaker class has just three properties on it, FirstName, LastName and Age. (That's not counting the Id property, which isn't visible to the user, and the FullName property, which is computed out of the FirstName and LastName properties; because they aren't user-modifiable, they're not really of concern here. Yet.) It wouldn't make sense for this conference system to allow for empty first or last names, and a negative age probably wouldn't make much sense, either. Let's fix these first.

Specifying non-zero names is one of the easiest validations to apply, because it's a static one. There's no complicated logic required—the length of the strings supplied to each property has to be greater than zero. This is handled by the StringLength attribute on each property, like so:

```
[StringLength(100,
  ErrorMessage = "First name must be between 1 and 100 characters",
  MinimumLength = 1)]
public virtual string FirstName { get; set; }

[StringLength(100,
  ErrorMessage = "Last name must be between 1 and 100 characters",
  MinimumLength = 1)]
public virtual string LastName { get; set; }
```

That takes care of the empty-names problem.

Age is even easier, because I can use the Range custom attribute to specify acceptable minimum and maximum age ranges. (Would I really consider bringing in a speaker younger than 21? Possibly, because I want to encourage school-age kids to speak, but anyone younger than 13 would probably be a tough sell.) Applying the Range attribute, then, would look like this:

```
[Range(13, 90, ErrorMessage = "Age must be between 13 and 90")]
public virtual int Age { get; set; }
```

Note that the StringLength and Range attributes also take an ErrorMessageResourceName value, in case error messages are stored in resources (which they should be, for easy internationalization).

Build, and run; notice how the UI will now automatically enforce these constraints. Even better, to the degree possible, the constraints will be enforced in the database, as well. Nifty!

In and of themselves, these attributes act essentially as data model validations, with a small amount of UI to support them. However, you often want to change up elements of the UI that have nothing to do with data validation, as well. For example, currently, the

attributes on the Speaker object are displayed in alphabetical order, which doesn't make a ton of sense. It would be far more realistic (and useful) if the first value displayed was the full name, followed by the individual fields for first name, last name and age (as well as any other demographic information you need to capture and use).

While this could certainly become the "Welp, that was fun, time to break down and build our own UI" moment, it doesn't need to—this is a common requirement, and NOF has it covered, via the MemberOrder attribute. Using this attribute, I can establish an "order" in which attributes should appear in the UI. So, for example, if I want the FullName attribute to appear first in the UI, I use MemberOrder and pass in the relative ordinal position "1," like so:

```
[Title]
[MemberOrder(1)]
public string FullName { get { return FirstName + " " + LastName; } }
```

Next, I'd like to display first name, last name and age, but here I can begin to run into a problem. As I add new fields to this class over time (say, "middle name" or "email"), trying to keep all the ordinal positions in order can be tricky—if I move LastName to position 5, I have to go find everything 5 (and after) and bump each one to get the right positions. That's a pain.

Fortunately, MemberOrder has a nifty little trick to it: The position itself can be a floating-point value, which allows fields to be "grouped," so that now I can mark "FirstName," "LastName," and "Age" as ordinal positions "2.1," "2.2," and "2.3," respectively, which essentially means that group "2" can be demographic information, and any new demographic information about the Speaker only requires reshuffling of the members in that particular group, as **Figure 1** shows.

Note that there's nothing really special about the values themselves—they're used relative to one another and don't represent any particular location on the screen. In fact, I could've used 10, 21, 22 and 23, if I wanted to. NOF is careful to point out that these values are compared lexicographically—string-based comparisons—and not numerically, so use whichever scheme makes sense to you.

What if users aren't sure whether Age is in years or in days? It may seem completely obvious to you, but remember, not everybody looks at the world the same way. While it's probably not a piece of information that needs to be present on the UI overtly, it should be something that you can signal to the user somehow. In NOF, you use the "DescribedAs" attribute to signal how the property should be described, which typically takes the form of a tooltip over the input area. (Remember, though, a given NOF client might choose to use a different way to signal that; for example, if a NOF client emerges for phones, which are touch-centric, tooltips don't work well for that format. In that situation, that hypothetical NOF client would use a different mechanism, one more appropriate for that platform, to describe the property.)

Figure 1 **Grouping Fields**

```
[MemberOrder(2.1)]
[StringLength(100,
  ErrorMessage = "First name must be between 1 and 100 characters",
  MinimumLength = 1)]
public virtual string FirstName { get; set; }

[MemberOrder(2.2)]
[StringLength(100,
  ErrorMessage = "Last name must be between 1 and 100 characters",
  MinimumLength = 1)]
public virtual string LastName { get; set; }

[Range(13, 90, ErrorMessage = "Age must be between 13 and 90")]
[MemberOrder(2.3)]
public virtual int Age { get; set; }
```

And Speakers need a bio! Oh, my, how could I forget that—that's like the one time speakers get to write exclusively about themselves and all the amazing things they do! (That's a joke—if it's one thing speakers hate most of all, it's writing their own bio.) Bio is an easy attribute to add to the class, but most bios need to be more than just a word or two, and looking at the UI generated by NOF so far, all of the other strings are currently single-line affairs. It's for this reason that NOF provides the "MultiLine" attribute, to indicate that this field should be displayed in a larger area of text entry than the typical string.

> In and of themselves, these attributes act essentially as data model validations, with a small amount of UI to support them.

But I need to be careful about, in this case, a speaker's biography, because free-form input offers the possibility for abuse: I might want/need to screen out certain words from appearing lest people get the wrong impression about the conference. I simply can't have speakers at my show if their biographies include words like COBOL! Fortunately, NOF will allow for validation of input by looking for, and invoking, methods on the Speaker class that match a Validate[Property] convention, like so:

```
[MemberOrder(4)]
[StringLength(400, ErrorMessage = "Keep bios to under 400 characters, please")]
public virtual string Bio { get; set; }
public string ValidateBio(string bio)
{
  if (bio.IndexOf("COBOL") > -1)
    return "We are terribly sorry; nobody wants to hear that";
  else
    return "";
}
```

Wrapping Up

NOF has a pretty wide variety of options available to describe a domain object in terms that make it easier to automatically render the appropriate UI to enforce domain limitations, but thus far the model here is pretty simple. In the next piece, I'll examine how NOF can handle a more complicated topic, that of relationships between objects. (Speakers need to be able to specify Topics they speak on, for example, and most of all, the Talks they propose.) But I'm out of space for the month, so in the meantime … Happy coding! ■

Ted Neward *is a Seattle-based polytechnology consultant, speaker and mentor. 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 technical expert who reviewed this article: *Richard Pawson*

Full Stack C# with Blazor

Jonathan C. Miller

Blazor, Microsoft's experimental framework that brings C# into the browser, is the missing piece in the C# puzzle. Today, a C# programmer can write desktop, server-side Web, cloud, phone, tablet, watch, TV and IoT applications. Blazor completes the puzzle, allowing a C# developer to share code and business logic right into the user's browser. This is a powerful ability and a gigantic productivity improvement for C# developers.

In this article, I'm going to demonstrate a common-use case for code sharing. I'll demonstrate how to share validation logic between a Blazor client and a WebAPI server application. Today it's expected that you validate the input not only on the server but also in the client browser. Users of modern Web applications expect near-real-time feedback. The days of filling out a long form and clicking Submit only to see a red error returned are mostly behind us.

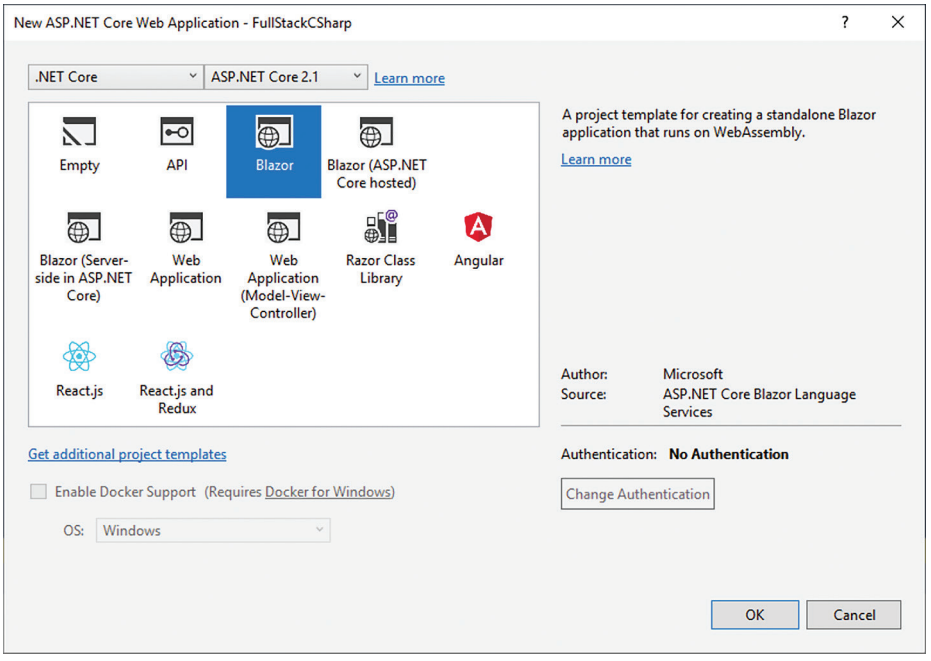


Figure 1 **Choosing a Blazor Application**

A Blazor Web application running inside the browser can share code with a C# back-end server. You can place your logic in a shared library and utilize it on the front and back ends. This has a lot of benefits. You can put all the rules in one place and know that they only have to be updated in one location. You know that they'll really work the same because they're the same code. You save a bunch of time in testing and troubleshooting issues where the client and server logic aren't always quite the same.

This article discusses:

- Creating a new Blazor application and a shared library project to house common C# code for client and server
- Creating a validation engine that shares logic in the browser and on the back end

Technologies discussed:

Blazor, C#, ASP.NET Core

Perhaps most notable, you can use one library for validation on both the client and the server. Traditionally, a JavaScript front end forces developers to write two versions of validation rules—one in JavaScript for the front end and another in the language used on the back end. Attempts to solve this mismatch involve complicated rules frameworks and additional layers of abstraction. With Blazor, the same .NET Core library runs on the client and server.

Blazor is still an experimental framework, but it's moving forward quickly. Before building this sample, make sure you have the correct version of Visual Studio, .NET Core SDK and Blazor language services installed. Please review the Getting Started steps on blazor.net.

Creating a New Blazor Application

First, let's create a new Blazor application. From the New Project dialog box click ASP.NET Core Web Application, click OK, then select the Blazor icon in the dialog box shown in **Figure 1**. Click OK. This will create the default Blazor sample application. If you've experimented with Blazor already, this default application will be familiar to you.

The shared logic that validates business rules will be demonstrated on a new registration form. **Figure 2** shows a simple form with fields for First Name, Last Name, Email and Phone. In this sample, it will validate that all the fields are required, that the name fields have a maximum length, and that the e-mail and phone number fields are in the correct format. It will display an error message under each field, and those messages will update as the user types. Last, the Register button will only be enabled if there are no errors.

Shared Library

All of the code that needs to be shared between the server and Blazor client will be placed in a separate shared library project. The shared library will contain the model class and a very simple validation engine. The model class will hold the data fields on the registration form. It looks like this:

```
public class RegistrationData : ModelBase
{
  [RequiredRule]
  [MaxLengthRule(50)]
  public String FirstName { get; set; }

  [RequiredRule]
  [MaxLengthRule(50)]
  public String LastName { get; set; }

  [EmailRule]
  public String Email { get; set; }

  [PhoneRule]
  public String Phone { get; set; }

}
```

The RegistrationData class inherits from a ModelBase class, which contains all of the logic that can be used to validate the rules and return error messages that are bound to the Blazor page. Each field is decorated with attributes that map to validation rules. I chose to create a very simple model that feels a lot like the Entity Framework (EF) Data Annotations model. All of the logic for this model is contained in the shared library.

The ModelBase class contains methods that can be used by the Blazor client application or the server application to determine if there are any validation errors. It will also fire an event when the model is changed, so the client can update the UI. Any model class can inherit from it and get all of the validation engine logic automatically.

I'll start by first creating a new ModelBase class inside of the SharedLibrary project, like so:

```
public class ModelBase
  {
}
```

Errors and Rules

Now I'll add a private dictionary to the ModelBase class that contains a list of validation errors. The _errors dictionary is keyed by the field name and then by the rule name. The value is the actual error message to be displayed. This setup makes it easy to determine if there are validation errors for a specific field and to retrieve the error messages quickly. Here's the code:

```
private Dictionary<String, Dictionary<String, String>> _errors =
  new Dictionary<string, Dictionary<string, string>>();
```

Now I'll add the AddError method for entering errors into the internal errors dictionary. AddError has parameters for fieldName, ruleName and errorText. It searches the internal errors dictionary and removes entries if they already exist, then adds the new error entry, as shown in this code:

```
private void AddError(String fieldName, String ruleName, String errorText)
{
  if (!_errors.ContainsKey(fieldName)) { _errors.Add(fieldName,
    new Dictionary<string, string>()); }
  if (_errors[fieldName].ContainsKey(ruleName))
    { _errors[fieldName].Remove(ruleName); }
  _errors[fieldName].Add(ruleName, errorText);
  OnModelChanged();
}
```

Finally, I'll add the RemoveError method, which accepts the fieldName and ruleName parameters and searches the internal errors dictionary for a matching error and removes it. Here's the code:

```
private void RemoveError(String fieldName, String ruleName)
{
  if (!_errors.ContainsKey(fieldName)) { _errors.Add(fieldName,
    new Dictionary<string, string>()); }
  if (_errors[fieldName].ContainsKey(ruleName))
    { _errors[fieldName].Remove(ruleName);
    OnModelChanged();
  }
}
```

The next step is to add the CheckRules functions that does the work of finding the validation rules attached to the model and executing them. There are two different CheckRules functions: One that lacks a parameter and checks all rules on all fields, and a second that has a fieldName parameter and only validates a specific field. This second function is used when a field is updated, and the rules for that field are validated immediately.

The CheckRules function uses reflection to find the list of attributes attached to a field. Then, it tests each attribute to see if it's a type of IModelRule. When an IModelRule is found, it calls the Validate method and returns the result, as shown in **Figure 3**.

Next, I add the Errors function. This function takes a field name as a parameter and returns a string that contains the list of errors

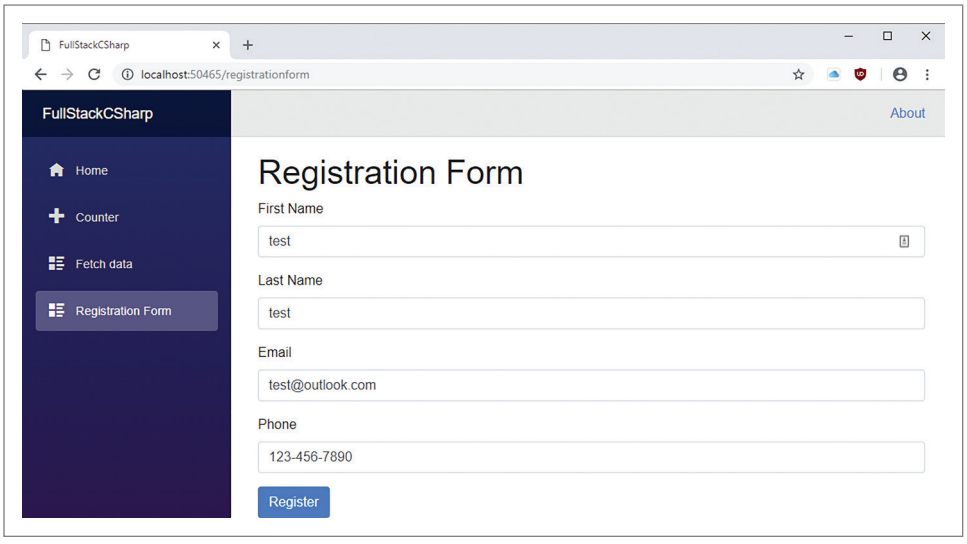


Figure 2 **Registration Form**

Figure 3 **The CheckRules Function**

```
public void CheckRules(String fieldName)
{
  var propertyInfo = this.GetType().GetProperty(fieldName);
  var attrInfos = propertyInfo.GetCustomAttributes(true);
  foreach (var attrInfo in attrInfos)
  {
    if (attrInfo is IModelRule modelrule)
    {
      var value = propertyInfo.GetValue(this);
      var result = modelrule.Validate(fieldName, value);
      if (result.IsValid)
      {
        RemoveError(fieldName, attrInfo.GetType().Name);
      }
      else
      {
        AddError(fieldName, attrInfo.GetType().Name, result.Message);
      }
    }
  }
}

public bool CheckRules()
{
  foreach (var propInfo in this.GetType().GetProperties(
    System.Reflection.BindingFlags.Public |
      System.Reflection.BindingFlags.Instance))
    CheckRules(propInfo.Name);

  return HasErrors();
}
```

for that field. It uses the internal _errors dictionary to determine if there are any errors for that field, as shown here:

```
public String Errors(String fieldName)
{
  if (!_errors.ContainsKey(fieldName)) { _errors.Add(fieldName,
    new Dictionary<string, string>()); }
  System.Text.StringBuilder sb = new System.Text.StringBuilder();
  foreach (var value in _errors[fieldName].Values)
    sb.AppendLine(value);

  return sb.ToString();
}
```

Now, I need to add the HasErrors function, which returns true if there are any errors on any field of the model. This method is used by the client to determine if the Register button should be enabled. It's also used by the WebAPI server to determine if the incoming model data has errors. Here's the function code:

```
public bool HasErrors()
{
  foreach (var key in _errors.Keys)
    if (_errors[key].Keys.Count > 0) { return true; }
  return false;
}
```

Values and Events

It's time to add the GetValue method, which takes a fieldname parameter and uses reflection to find the field in the model and return its value. This is used by the Blazor client to retrieve the current value and display it in the input box, as shown right here:

```
public String GetValue(String fieldName)
{
  var propertyInfo = this.GetType().GetProperty(fieldName);
  var value = propertyInfo.GetValue(this);

  if (value != null) { return value.ToString(); }
  return String.Empty;
}
```

Now add the SetValue method. It uses reflection to find the field in the model and update its value. It then fires off the CheckRules method that validates all the rules on the field. It's used in the Blazor client to update the value as the user types in the input textbox. Here's the code:

```
public void SetValue(String fieldName, object value)
{
  var propertyInfo = this.GetType().GetProperty(fieldName);
  propertyInfo.SetValue(this, value);
  CheckRules(fieldName);
}
```

Finally, I add the event for ModelChanged, which is raised when a value on the model has been changed or a validation rule has been added or removed from the internal dictionary of errors. The Blazor client listens for this event and updates the UI when it fires. This is what causes the errors displayed to update, as shown in this code:

```
public event EventHandler<EventArgs> ModelChanged;

protected void OnModelChanged()
{
  ModelChanged?.Invoke(this, new EventArgs());
}
```

This validation engine is admittedly a very simple design with lots of opportunities for improvement. In a production-business application, it would be useful to have severity levels for the errors, such as Info, Warning and Error. In certain scenarios, it would be helpful if the rules could be loaded dynamically from a configuration file without the need to modify the code. I'm not advocating that you create your own validation engine; there are a lot of choices out there. This one is designed to be good enough to demo a real-world example, but simple enough to make it fit into this article and be easy to understand.

Making the Rules

At this point, there's a RegistrationData class that contains the form fields. The fields in the class are decorated with attributes such as RequiredRule and EmailRule. The RegistrationData class inherits from a ModelBase class that contains all the logic to validate the rules and to notify the client of changes. The last piece of the validation engine is the rule logic itself. I'll explore that next.

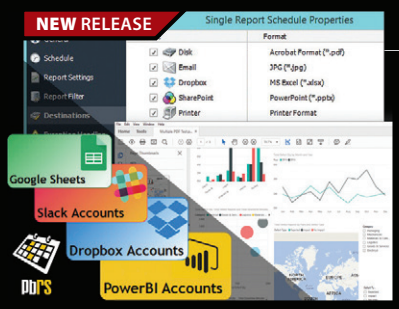
I start by creating a new class in the SharedLibrary called IModelRule. This rule consists of a single Validate method that returns a ValidationResult. Every rule must implement the IModelRule interface, as shown here:

Figure 4 **The MaxLengthRule Class**

```
public class MaxLengthRule : Attribute, IModelRule
{
  private int _maxLength = 0;
  public MaxLengthRule(int maxLength) { _maxLength = maxLength; }

  public ValidationResult Validate(string fieldName, object fieldValue)
  {
    var message = $"Cannot be longer than {_maxLength} characters";
    if (fieldValue == null) { return new ValidationResult() { IsValid = true }; }

    var stringvalue = fieldValue.ToString();
    if (stringvalue.Length > _maxLength )
    {
      return new ValidationResult() { IsValid = false, Message = message };
    }
    else
    {
      return new ValidationResult() { IsValid = true };
    }
  }
}
```



PBRS (Power BI Reports Scheduler) | from $9,811.51

Christiansteven

Date & time Scheduling for Power BI reports with one Power BI License.

- Exports reports to PDF, Excel, Excel Data, Word, PowerPoint, CSV, JPG, HTML, PNG and ePub
- Send reports to email, printer, Slack, Google Sheets, folder, FTP, DropBox & SharePoint
- Uses database queries to automatically populate report filters, email addresses & body text
- Adds flexibility with custom calendars e.g. 4-4-5, holidays, "nth" day of the month, etc.
- Responds instantly by firing off reports when an event occurs e.g. database record is updated

DevExpress DXperience 18.2 | from $1,439.99

DevExpress

A comprehensive suite of .NET controls and UI libraries for all major Microsoft dev platforms.

- WinForms – New Sunburst Chart, Office Navigation UX, SVG Office 2019 skins
- WPF – New Gantt control, improved Data Filtering UX and App Theme Designer
- ASP.NET & MVC – New Adaptive Layouts, improved Rich Text Editor and Spreadsheet
- Reporting – Vertical Band support, Free-hand drawing and improved Report wizards
- JavaScript – New HTML/Markdown WYSIWYG editor, Improved Grid and TreeList performance

LEADTOOLS Medical Imaging SDKs V20 | from $4,995.00 SRP

LEADTOOLS
THE WORLD LEADER IN IMAGING SDKs

Powerful DICOM, PACS, and HL7 functionality.

- Load, save, edit, annotate & display DICOM Data Sets with support for the latest specifications
- High-level PACS Client and Server components and frameworks
- OEM-ready HTML5 Zero-footprint Viewer with 3D rendering support and DICOM Storage Server
- Medical-specific image processing functions for enhancing 16-bit grayscale images
- Native libraries for .NET, C/C++, HTML5, JavaScript, WinRT, iOS, OS X, Android, Linux, & more

Help & Manual Professional | from $586.04

ecsoftware

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

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

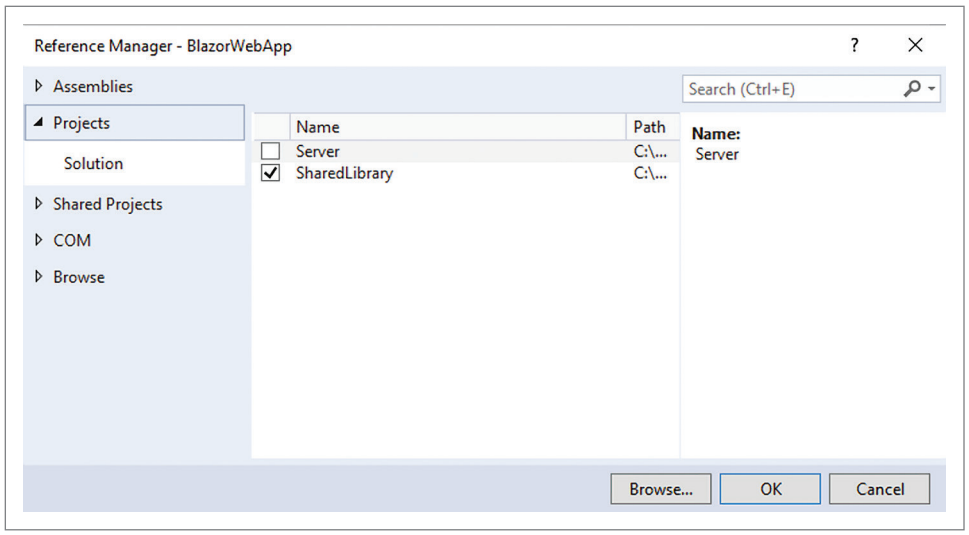


Figure 5 **Adding a Reference to the Shared Library**

```
public interface IModelRule
{
  ValidationResult Validate(String fieldName, object fieldValue);
}
```

Next, I create a new class in the SharedLibrary called ValidationResult, which consists of two fields. The IsValid field tells you whether the rule is valid or not, while the Message field contains the error message to be displayed when the rule is invalid. Here's that code:

```
public class ValidationResult
{
  public bool IsValid { get; set; }
  public String Message { get; set; }
}
```

The sample application uses four different rules, all of which are public classes that inherit from the Attribute class and implement the IModelRule interface.

Now it's time to create the rules. Keep in mind that all validation rules are simply classes that inherit from the Attribute class and implement the IModelRule interface's Validate method. The max-length rule in **Figure 4** returns an error if the text entered is longer than the specified maximum length. The other rules, for Required, Phone and Email, work similarly, but with different logic for the type of data they validate.

Figure 6 **Adding a Registration Form Link**

```
<div class=@(collapseNavMenu ? "collapse" : null) onclick=@ToggleNavMenu>
  <ul class="nav flex-column">
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="" Match=NavLinkMatch.All>
        <span class="oi oi-home" aria-hidden="true"></span> Home
      </NavLink>
    </li>
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="counter">
        <span class="oi oi-plus" aria-hidden="true"></span> Counter
      </NavLink>
    </li>
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="fetchdata">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data
      </NavLink>
    </li>
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="registrationform">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Registration Form
      </NavLink>
    </li>
  </ul>
</div>
```

Creating the Blazor Registration Form

Now that the validation engine is complete in the shared library, it can be applied to a new registration form in the Blazor application. I start by first adding a reference to the shared-library project from the Blazor application. You do this from the Solution window of the Reference Manager dialog box, as shown in **Figure 5**.

Next, I add a new navigation link to the application's NavMenu. I open the Shared\NavMenu.cshtml file and add a new Registration Form link to the list, as shown in **Figure 6**.

Finally, I add the new RegistrationForm.cshtml file in the Pages folder. You do this with the code shown in **Figure 7**.

The cshtml code in **Figure 7** includes four <TextInput> fields inside the <form> tag. The <TextInput> tag is a custom Blazor component that handles the data binding and error-display logic for the field. The component only needs three parameters to work:

- Model field: Identifies the class it's data-bound to.
- FieldName: Identifies the data member to data bind to.
- DisplayName field: Enables the component to display user-friendly messages.

Inside the @functions block of the page, the code is minimal. The OnInit method initializes the model class with some test data inside it. It binds to the ModelChanged event and calls the CheckRules method to validate the rules. The ModelChanged handler calls the base.StateHasChanged method to force a UI refresh. The Register method is called when the Register button is clicked, and it sends the registration data to a back-end WebAPI service.

The TextInput component contains the input label, the input textbox, the validation error message and the logic to update the model as the user types. Blazor components are very simple to write and provide a powerful way to decompose an interface into reusable parts. The parameter members are decorated with the Parameter attribute, letting Blazor know that they're component parameters.

The input textbox's oninput event is wired to the OnFieldChanged handler. It fires every time the input changes. The OnFieldChanged handler then calls the SetValue method, which causes the rules for that field to be executed, and the error message to be updated in real time as the user types. **Figure 8** shows the code.

Validation on the Server

The validation engine is now wired up and working on the client. The next step is to use the shared library and the validation engine on the server. To do this, I start by adding another ASP.NET Core Web Application project to the solution. This time I choose API

Figure 7 **Adding the RegistrationForm.cshtml File**

```
@page "/registrationform"
@inject HttpClient Http
@using SharedLibrary

<h1>Registration Form</h1>

@if (!registrationComplete)
{
  <form>
    <div class="form-group">
      <TextInput Model="model" FieldName="FirstName" DisplayName="First Name" />
    </div>
    <div class="form-group">
      <TextInput Model="model" FieldName="LastName" DisplayName="Last Name" />
    </div>
    <div class="form-group">
      <TextInput Model="model" FieldName="Email" DisplayName="Email" />
    </div>
    <div class="form-group">
      <TextInput Model="model" FieldName="Phone" DisplayName="Phone" />
    </div>

    <button type="button" class="btn btn-primary" onclick="@Register"
      disabled="@model.HasErrors()">Register</button>
  </form>
}
else
{
  <h2>Registration Complete!</h2>
}

@functions {
  bool registrationComplete = false;
  RegistrationData model { get; set; }

  protected override void OnInit()
  {
    base.OnInit();
    model = new RegistrationData() { FirstName =
      "test", LastName = "test", Email = "test@test.com", Phone = "1234567890" };
    model.ModelChanged += ModelChanged;
    model.CheckRules();
  }

  private void ModelChanged(object sender, EventArgs e)
  {
    base.StateHasChanged();
  }

  async Task Register()
  {
    await Http.PostJsonAsync<RegistrationData>(
      "https://localhost:44332/api/Registration", model);
    registrationComplete = true;
  }
}
```

Figure 8 **Updating the Error Message**

```
@using SharedLibrary

<label>@DisplayName</label>
<input type="text" class="form-control" placeholder="@DisplayName"
  oninput="@(e => OnFieldChanged(e.Value))"
  value="@Model.GetValue(FieldName)" />
<small class="form-text" style="color:darkred;">@Model.Errors(FieldName)
  </small>

@functions {

  [Parameter]
  ModelBase Model { get; set; }

  [Parameter]
  String FieldName { get; set; }

  [Parameter]
  String DisplayName { get; set; }

  public void OnFieldChanged(object value)
  {
    Model.SetValue(FieldName, value);
  }
}
```

Figure 9 **The Registration Controller**

```
[Route("api/Registration")]
[ApiController]
public class RegistrationController : ControllerBase
{
  [HttpPost]
  public IActionResult Post([FromBody] RegistrationData value)
  {
    if (value.HasErrors())
    {
      return BadRequest();
    }
    // TODO: Save data to database
    return Created("api/registration", value);
  }
}
```

instead of Blazor in the New ASP.NET Core Web Application dialog box shown in **Figure 1**.

Once the new API project is created, I add a reference to the shared project, just as I did in the Blazor client application (see **Figure 5**). Next, I add a new controller to the API project. The new controller will accept the RegistrationData call from the Blazor client, as shown in **Figure 9**. The registration controller runs on the server and is typical of a back-end API server. The difference here is that it now runs the same validation rules that run on the client.

The registration controller has a single POST method that accepts the RegistrationData as its value. It calls the HasErrors method, which validates all the rules and returns a Boolean. If there are errors, the controller returns a BadRequest response; otherwise, it returns a success response. I've intentionally left out the code that would save the registration data to a database so I can focus on the validation scenario. The shared validation logic now runs on the client and server.

The Big Picture

This simple example of sharing validation logic in the browser and the back end barely scratches the surface of what's possible in a full-stack C# environment. The magic of Blazor is that it allows the army of existing C# developers to build powerful, modern and responsive single-page applications with a minimal ramp-up period. It allows businesses to reuse and repackage existing code so it can run right in the browser. The ability to share C# code among browser, desktop, server, cloud and mobile platforms will greatly increase developer productivity. It will also allow developers to deliver more features and more business value to customers faster. ■

Jonathan Miller *is a senior architect. He's been developing products on the Microsoft stack for a decade and programming on .NET since its inception. Miller is a full-stack product developer with expertise in front-end technologies (Windows Forms, Windows Presentation Foundation, Silverlight, ASP.NET, Angular/Bootstrap), middleware (Windows services, Web API), and back ends (SQL server, Azure).*

Thanks to the following technical expert for reviewing this article: *Dino Esposito*

Parse the Command Line with System.CommandLine

Mark Michaelis

Going all the way back to. NET Framework 1.0, I've been astounded that there's been no simple way for developers to parse the command line of their applications. Applications start execution from the Main method, but the arguments are passed in as an array (string[] args) with no differentiation between which items in the array are commands, options, arguments and the like.

I wrote about this problem in a previous article ("How to Contribute to Microsoft Open Source Software Projects," msdn.com/magazine/mt830359), and described my work with Microsoft's Jon Sequeira. Sequeira has lead an open source team of developers to create a new command-line parser that can accept command-line arguments and parse them into an API called System.CommandLine, which does three things:

- Allows for the configuration of a command line.
- Enables parsing of command-line generic arguments (tokens) into distinct constructs, where each word on the command line is a token. (Technically, command-line hosts allow for the combining of words into a single token using quotes.)
- Invokes functionality that's configured to execute based on the command-line value.

This article discusses:

- System.CommandLine—a new API for parsing the command line that launches an application
- Support for self-contained applications, including on Linux

Technologies discussed:

.NET Core, System.CommandLine

The constructs supported include commands, options, arguments, directives, delimiters and aliases. Here's a description of each construct:

Commands: These are the actions that are supported by the application command line. Consider, for example, git. Some of the built-in commands for git are branch, add, status, commit and push. Technically, the commands specified after the executable name are, in fact, subcommands. Subcommands to the root command—the name of the executable itself (for example, git.exe)—may themselves have their own subcommands. For instance, the command "dotnet add package" has "dotnet" as the root command, "add" as a subcommand and "package" as the subcommand to add (perhaps call it the sub-subcommand?).

Options: These provide a way to modify the behavior of a command. For example, the dotnet build command includes the --no-restore option, which you can specify to disable the restore command from running implicitly (and instead relying on prior execution of the restore command). As the name implies, options are generally not a required element of a command.

Arguments: Both commands and options can have associated values. For example, the dotnet new command includes the template name. This value is required when you specify the new command. Similarly, options may have values associated with them. Again, with dotnet new, the --name option has an argument for specifying the name of the project. The value associated with a command or option is called the argument.

Directives: These are commands that are cross-cutting for all applications. For example, a redirect command can force all output (stderr and stdout) to go into an .xml format. Because directives are part of

the System.CommandLine framework, they're included automatically, without any effort on the part of the command-line interface developer.

Delimiters: The association of an argument to a command or an option is done via a delimiter. Common delimiters are space, colon and the equal sign. For example, when specifying the verbosity of a dotnet build, you can use any of the following three variations: --verbosity=diagnostic, --verbosity diagnostic or --verbosity:diagnostic.

Aliases: These are additional names that can be used to identify commands and options. For example, with dotnet, "classlib" is an alias for "Class library" and -v is an alias for "--verbosity."

Fortunately, the new System.CommandLine API provides a significant improvement on this simple scenario, and does so in a way I haven't previously seen.

Prior to System.CommandLine, the lack of built-in parsing support meant that when your application launched, you as the developer had to analyze the array of arguments to determine which corresponded to which argument type, and then correctly associate all of the values together. While .NET includes numerous attempts at solving this problem, none has emerged as a default solution, and none scales well to support both simple and complex scenarios. With this in mind, System.CommandLine was developed and released in alpha form (see github.com/dotnet/command-line-api).

Keep Simple Things Simple

Imagine that you're writing an image conversion program that converts an image file to a different format based on the output name specified. The command line could be something like this:

```
imageconv --input sunrise.CR2 --output sunrise.JPG
```

Given this command line (see "Passing Parameters to the .NET Core Executable" for alternative command-line syntax), the imageconv program will launch into the Main entry point, static void Main(string[] args), with a string array of four corresponding arguments. Unfortunately, there's no association between --input and sunrise.CR2 or between --output and sunrise.JPG. Neither is there any indication that --input and --output identify options.

Fortunately, the new System.CommandLine API provides a significant improvement on this simple scenario, and does so in a way I haven't previously seen. The simplification is that you can program a Main entry point with a signature that matches the command line. In other words, the signature for Main becomes:

```
static void Main(string input, string output)
```

That's right, System.CommandLine enables the automatic conversion of the --input and --output options into parameters on Main, replacing the need to even write a standard Main(string[] args) entry point. The only additional requirement is to reference an assembly that enables this scenario. You can find details on what to reference at itl.tc/syscmddf, as any instructions provided here are likely to be quickly dated once the assembly is released on NuGet. (No, there's no language change to support this. Rather, when adding the reference, the project file is modified to include a build task that generates a standard Main method with a body that uses reflection to call the "custom" entry point.)

Furthermore, arguments aren't limited to strings. There's a host of built-in converters (and support for custom converters) that allow you, for example, to use System.IO.FileInfo for the parameter type on input and output, like so:

```
static void Main(FileInfo input, FileInfo output)
```

As described in the article section, "System.CommandLine Architecture," System.CommandLine is broken into a core module and an app provider module. Configuring the command line from Main is an App Model implementation, but for now I'll just refer to the entire API set as System.CommandLine.

The mapping between command-line arguments and Main method parameters is basic today, but still relatively capable for lots of programs. Let's consider a slightly more complex imageconv command line that demonstrates some of the additional features. **Figure 1** displays the command-line help.

The corresponding Main method that enables this updated command line is shown in **Figure 2**. Even though the example has nothing more than a fully documented Main method, there are numerous features enabled automatically. Let's explore the functionality that's built-in when you use System.CommandLine.

The first bit of functionality is the help output for the command line, which is inferred from the XML comments on Main. These comments not only allow for a general description of the program

Figure 1 **Sample Command Line for imageconv**

```
imageconv:
  Converts an image file from one format to another.

Usage:
  imageconv [options]

Options:
  --input          The path to the image file that is to be converted.
  --output         The target name of the output after conversion.
  --x-crop-size    The X dimension size to crop the picture.
                   The default is 0 indicating no cropping is required.
  --y-crop-size    The Y dimension size to crop the picture.
                   The default is 0 indicating no cropping is required.
  --version        Display version information
```

Figure 2 **Main Method Supporting the Updated imageconv Command Line**

```
/// <summary>
/// Converts an image file from one format to another.
/// </summary>
/// <param name="input">The path to the image file that is to be
    converted.</param>
/// <param name="output">The name of the output from the conversion.
    </param>
/// <param name="xCropSize">The x dimension size to crop the picture.
    The default is 0 indicating no cropping is required.</param>
/// <param name="yCropSize">The x dimension size to crop the picture.
    The default is 0 indicating no cropping is required.</param>
public static void Main(
  FileInfo input, FileInfo output,
  int xCropSize = 0, int yCropSize = 0)
```

(specified in the summary XML comment), but also for the documentation on each argument using parameter XML comments. Leveraging the XML comments requires enabling doc output, but this is configured automatically for you when referencing the assembly that enables configuration via Main. There's built-in help output with any of three command-line options: -h, -?, or --help. For example, the help displayed in **Figure 1** is automatically generated by System.CommandLine.

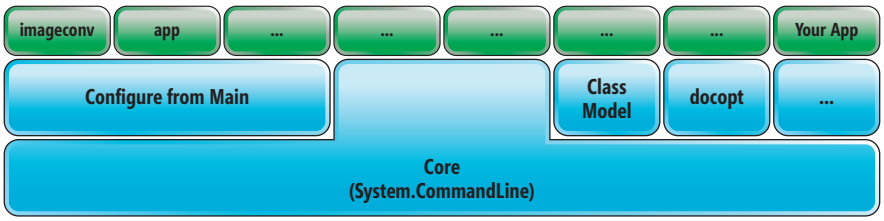


Figure 3 **System.CommandLine Architecture**

Similarly, while there's no version parameter on Main, System.CommandLine automatically generates a --version option that outputs the assembly version of the executable.

Another feature, command-line syntax verification, detects if a required argument (for which no default is specified on the parameter) is missing. If a required argument isn't specified, System.CommandLine automatically issues an error that reads, "Required argument missing for option: --output." Although somewhat counterintuitive, by default options with arguments are required. However, if the argument value associated with an option isn't required, you can leverage C# default parameter value syntax. For example:

```
int xCropSize = 0
```

There's also built-in support for parsing options regardless of the sequence in which the options appear on the command line. And it's worth noting that the delimiter between the option and the argument may be a space, a colon or an equal sign by default. Finally, Camel casing on Main's parameter names is converted to Posix-style argument names (that is, xCropSize translates to --x-crop-size on the command line).

If you type an unrecognized option or command name, System.CommandLine automatically returns a command-line error that reads, "Unrecognized command or argument …." However, if the name specified is similar to an existing option, the error will prompt with a typo correction suggestion.

There are some built-in directives available to all command-line applications that use System.CommandLine. These directives are enclosed in square brackets and appear immediately following the application name. For example, the [debug] directive triggers a breakpoint that allows you to attach a debugger, while [parse] displays a preview of how tokens are parsed, as shown here:

```
imageconv [parse] --input sunrise.CR2 --output sunrise.JPG
```

In addition, automated testing via an IConsole interface and TestConsole class implementation is supported. To inject the TestConsole into the command-line pipeline, add an IConsole parameter to Main, like so:

```
public static void Main(
  FileInfo input, FileInfo output,
  int xCropSize = 0, int yCropSize = 0,
    IConsole console = null)
```

To leverage the console parameter, replace invocations to System.Console with the IConsole parameter. Note that the IConsole parameter will be set automatically for you when invoked directly from the command line (rather than from a unit test), so even though the parameter is assigned null by default, it shouldn't have a null value unless you write test code that invokes it that way. Alternatively, consider putting the IConsole parameter first.

One of my favorite features is support for tab completion, which end users can opt into by running a command to activate it (see bit.ly/2sSRsQq). This is an opt-in scenario because users tend to be protective of implicit changes to the shell. Tab completion for options and command names happens automatically, but there's also tab completion for arguments via suggestions. When configuring a command or option, the tab completion values can come from a static list of values, such as the q, m, n, d or diagnostic values of --verbosity. Or they can be dynamically provided at run time, such as from REST invocation that returns a list of available NuGet packages when the argument is a NuGet reference.

> The mapping between command-line arguments and Main method parameters is basic today, but still relatively capable for lots of programs.

Using the Main method as the specification for the command line is just one of several ways that you can program using System.CommandLine. The architecture is flexible, allowing other ways to define and work with the command line.

The System.CommandLine Architecture

System.CommandLine is architected around a core assembly that includes an API for configuring the command line and a parser that resolves the command-line arguments into a data structure. All the features listed in the previous section can be enabled via the core assembly, except for enabling a different method signature for Main. However, support for configuring the command line, specifically using a domain-specific language (such as a Main like method) is enabled by an app model. (The app model used for the Main like method implementation described earlier is code-named "DragonFruit.") However, the System.CommandLine architecture enables support for additional app models (as shown in **Figure 3**).

For example, you could write an app model that uses a C# class model to define the command-line syntax for an application. In such a model, property names might correspond to the option name and the property type would correspond to the data type into which to

convert an argument. In addition, the model might leverage attributes to define aliases, for example. Alternatively, you could write a model that parses a docopt file (see docopt.org) for the configuration. Each of these app models would invoke the System.CommandLine configuration API. Of course, developers might prefer to call System.CommandLine directly from their application rather than via an app model, and this approach is also supported.

Figure 4 **Working with System.CommandLine Directly**

```
using System;
using System.CommandLine;
using System.CommandLine.Invocation;
using System.IO;

...

public static async Task<int> Main(params string[] args)
{
  RootCommand rootCommand = new RootCommand(
    description: "Converts an image file from one format to another."
    , treatUnmatchedTokensAsErrors: true);

  Option inputOption = new Option(
    aliases: new string[] { "--input", "-i" }
    , description: "The path to the image file that is to be converted."
    , argument: new Argument<FileInfo>());
  rootCommand.AddOption(inputOption);

  Option outputOption = new Option(
    aliases: new string[] { "--output", "-o" }
    , description: "The target name of the output file after conversion."
    , argument: new Argument<FileInfo>());
  rootCommand.AddOption(outputOption);

  Option xCropSizeOption = new Option(
    aliases: new string[] { "--x-crop-size", "-x" }
    , description: "The x dimension size to crop the picture.
      The default is 0 indicating no cropping is required."
    , argument: new Argument<FileInfo>());
  rootCommand.AddOption(xCropSizeOption);

  Option yCropSizeOption = new Option(
    aliases: new string[] { "--y-crop-size", "-y" }
    , description: "The Y dimension size to crop the picture.
      The default is 0 indicating no cropping is required."
    , argument: new Argument<FileInfo>());
  rootCommand.AddOption(yCropSizeOption);

  rootCommand.Handler =
    CommandHandler.Create<FileInfo, FileInfo, int, int>(Convert);

  return await rootCommand.InvokeAsync(args);
}

static public void Convert(
  FileInfo input, FileInfo output, int xCropSize = 0, int yCropSize = 0)
{
  // Convert...
}
```

Figure 5 **Using Method-First Approach to Configure System.CommandLine**

```
public static async Task<int> Main(params string[] args)
{
  RootCommand rootCommand = new RootCommand(
    description: "Converts an image file from one format to another."
    , treatUnmatchedTokensAsErrors: true);

  MethodInfo method = typeof(Program).GetMethod(nameof(Convert));

  rootCommand.ConfigureFromMethod(method);
  rootCommand.Children["--input"].AddAlias("-i");
  rootCommand.Children["--output"].AddAlias("-o");

  return await rootCommand.InvokeAsync(args);
}
```

Passing Parameters to the .NET Core Executable

When specifying command-line arguments in combination with the dotnet run command, the full command line would be:

```
dotnet run --project imageconv.csproj -- --input sunrise.CR2
  --output sunrise.JPG
```

If you're running dotnet from the same directory in which the csproj file was located, however, the command line would read:

```
dotnet run -- --input sunrise.CR2 --output sunrise.JPG
```

The dotnet run command uses the "--" as the identifier, indicating that all other arguments should be passed to the executable for it to parse.

Starting with .NET Core 2.2, there's also support for self-contained applications (even on Linux). With a self-contained application, you can launch it without using dotnet run and instead just rely on the resulting executable, like so:

```
imageconv.exe --input sunrise.CR2 --output sunrise.JPG
```

Obviously, this is expected behavior to Windows users.

Making the Complex Possible

Earlier, I mentioned that the functionality for keeping simple things simple was basic. This is because enabling command-line parsing via the Main method still lacks some features that some might consider important. For example, you can't configure a (sub) command or an option alias. If you encounter these limitations, you can build your own app model or call into the Core (System.CommandLine assembly) directly.

System.CommandLine includes classes that represent the constructs of a command line. This includes Command (and RootCommand), Option and Argument. **Figure 4** provides some sample code for invoking System.CommandLine directly and configuring it to accomplish the basic functionality defined in the help text of **Figure 1**.

> System.CommandLine includes classes that represent the constructs of a command line.

In this example, rather than rely on a Main app model to define the command-line configuration, each construct is instantiated explicitly. The only functional difference is the addition of aliases for each option. Leveraging the Core API directly, however, provides more control than what's possible with the Main like approach.

For example, you could define subcommands, like an image-enhance command that includes its own set of options and arguments related to the enhance action. Complex command-line programs have multiple subcommands and even sub-subcommands. The dotnet command, for example, has the dotnet sln add command, where dotnet is the root command, sln is one of the many subcommands, and add (or list and remove) is a child command of sln.

The final call to InvokeAsync implicitly sets up many of the features automatically including:

- Directives for parse and debug.
- The configuration of the help and version options.
- Tab completion and typo corrections.

There are also separate extension methods for each feature if finer-grain control is necessary. There are also numerous other configuration capabilities exposed by the Core API. These include:

- Handling tokens that are explicitly unmatched by the configuration.
- Suggestion handlers that enable tab completion, returning a list of possible values given the current command-line string and the location of the cursor.
- Hidden commands that you don't want to be discoverable using tab completion or help.

In addition, while there are lots of knobs and buttons to control the command-line parsing with System.CommandLine, it also provides a method-first approach. In fact, this is what's used internally to bind to the Main like method. With the method-first approach you can use a method like Convert at the bottom of **Figure 4** to configure the parser (as shown in **Figure 5**).

In this case, notice that the Convert method is used for the initial configuration and then you navigate the root command's object model to add aliases. The Children indexable property contains all the options and commands attached to the root command.

Wrapping Up

I'm very excited about the functionality available in System.CommandLine. The fact that achieving the simple scenarios explored here requires so little code is marvelous. Furthermore, the amount of functionality achieved—including things like tab completion, argument conversion and automated testing support, just to name a few—means that with little effort you can have a fully functioning command-line support in all of your dotnet applications.

Finally, System.CommandLine is open source. That means if there's functionality missing that you require, you can develop the enhancement and submit it back to the community as a pull request. A couple things I would personally love to see added are support for not always specifying the option or command names on the command line and relying instead on the position of the arguments to imply what the names are. Additionally, it would be great if you could declaratively add an additional alias (such as short aliases) when using the Main like or method-first approach. ■

Mark Michaelis *is founder of IntelliTect, where he serves as its chief technical architect and trainer. He has been a Microsoft MVP for more than two decades, and a Microsoft Regional Director since 2007. Michaelis serves on several Microsoft software design review teams, including C#, Microsoft Azure, SharePoint and Visual Studio ALM. He speaks at developer conferences and has written numerous books including his most recent, "Essential C# 7.0 (6th Edition)" (itl.tc/EssentialCSharp). Contact him on Facebook at facebook.com/Mark.Michaelis, on his blog at IntelliTect.com/Mark, on Twitter: @markmichaelis or via e-mail at mark@IntelliTect.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Kevin Bost, Kathleen Dollard, Jon Sequeira*

Verify e-Documents with Smart Contracts in Azure Blockchain Development Kit

Stefano Tempesta

The introduction of smart contracts in blockchain networks has created a business logic tier that was missing in the early iterations of blockchain. Smart contracts offer the ability to apply conditional logic to transactions before they're executed. Still, smart contracts can operate only on data that's stored on the blockchain digital ledger. Business processes, however, rarely run in isolation. They often need data integration with external systems and devices.

For example, processes may include transactions initiated on a distributed ledger that employs data sourced from an external system, service or device. External systems may be required to react to events raised by smart contracts in response to validation logic. This article describes how to automate document sign and verify workflows in SharePoint using the recently released Azure Blockchain Development Kit (aka.ms/bcdevkit) for persisting files' hash and metadata on a blockchain digital ledger.

This article discusses:

- Signing and verification of electronic documents on a blockchain
- Integration of SharePoint with Azure Blockchain Workbench using the Blockchain Development Kit

Technologies discussed:

Blockchain, Smart Contract, Azure Blockchain Development Kit

Azure Blockchain Development Kit

The release of the Azure Blockchain Development Kit, built on Microsoft's serverless technology, represents a milestone in the adoption of blockchain technologies in the enterprise space. Thanks to the Blockchain Development Kit, you can now build solutions that seamlessly integrate blockchain with the best of Microsoft and third-party software applications. As mentioned on its release notes, the initial version of the kit prioritizes capabilities related to three key themes: connecting interfaces, integrating data and systems, and deploying smart contracts and blockchain networks.

Connection includes communication channels such as mobile and Web, SMS and voice, as well as IoT devices and even chat bots. Integration with line-of-business applications spans multiple systems, including SharePoint, OneDrive for Business, Dynamics 365, open source, and any API-enabled platforms, as well as legacy protocols like file systems, FTP servers, or SQL databases. The deployment of smart contracts and blockchain networks will help mainstream blockchain technology in enterprise software development, and introduce governance and DevOps to the blockchain software development practice.

Blockchain Development Kit works in combination with Azure Logic Apps and Flow, which provide a visual design environment for workflows that include more than 200 connectors to Microsoft and third-party systems and services. In concert, they dramatically simplify the development of end-to-end blockchain applications

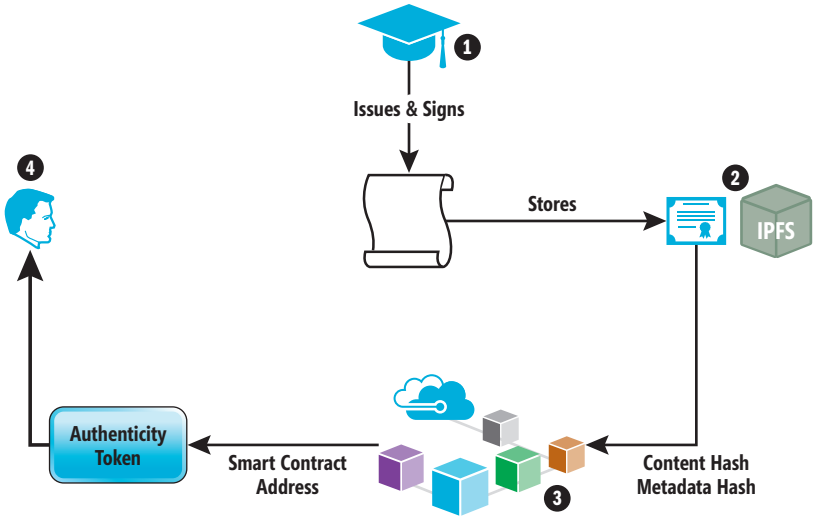


Figure 1 **The Signing Actors and Process**

that access on- and off-chain data, handle events generated by the digital ledger, and leverage the Azure ecosystem for a seamless and integrated solution. Let's explore a practical application in the context of enterprise content management.

Signing Digital Assets

With blockchain, you can imagine a world in which documents are embedded in digital code and stored in transparent, shared databases, where they're protected from deletion, tampering and revision. In this world every agreement, every process, every task, and every payment would have a digital record and signature that could be identified, validated, stored, and shared. Intermediaries like lawyers, brokers and institutions might no longer be necessary. Individuals, organizations, and machines would freely transact and interact with one another with little friction. This is the immense potential of blockchain.

The potential application of content decentralization and distribution is enormous. With a single, immutable and verifiable record store, people will own their digital identity and records—think of identity or residence documents, medical records, educational or professional certificates and licenses. All these documents and their metadata can be issued on the blockchain and be digitally signed. No more fake certifications, no more degree mills, no more "photoshopped" papers.

> The potential application of content decentralization and distribution is enormous.

Students, for example, may apply for further study, a job, or immigration to another country; and in the process may be required to prove their level of study or knowledge of language to attend university. Entities like recruiters, employers, governments and universities can verify the student's credentials without relying on central authorities—in just minutes, and with no other intermediaries.

Figure 1 describes the mentioned scenario. Certificates are issued by an authority, such as an education institute (1), stored on a centralized document management server (2), or on a distributed file system like IPFS (ipfs.io) and signed with a cryptographic function. I'll go into more about IPFS later in the article. The content hash and certificate's metadata hash are then stored on the blockchain digital ledger (3) and attached to the user's digital identity as a smart contract address that stores this information (4). This represents a sort of unique authenticity token, which identifies the document in a non-questionable way.

A common pattern is to generate a unique hash of the digital asset and a unique hash of the metadata that describes it. Those hashes are then stored on a blockchain. If authenticity of a document is ever questioned, the off-chain file can be re-hashed at a later time and that hash compared to the on-chain value. If the hash values match, the document is authentic, but if just a character in a document is modified, the hashes won't match, making obvious that a change has occurred.

Build the Signing Logic App Flow

Let's look at a potential implementation of this workflow using Azure Logic App. The Logic App flow will generate a document and metadata hashes, and store the former on SharePoint and the latter on an Ethereum network, using the Ethereum connector available as part of the Azure Blockchain Development Kit. The calculation of the hash value is done in an Azure Function built on the .NET runtime stack. The function is based on the HTTP trigger template, and it will be run as soon as it receives an HTTP request.

Figure 2 **The ComputeHashFunction**

```
public static class ComputeHashFunction
{
  [FunctionName("ComputeHashFunction")]
  public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function,
      "get", "post", Route = null)] HttpRequest req, ILogger log)
  {
    string requestBody =
      await new StreamReader(req.Body).ReadToEndAsync();

    string hash = ComputeHash(requestBody);

    return (ActionResult)new OkObjectResult(hash);
  }

  private static string ComputeHash(string data)
  {
    // Create a SHA256 hash
    using (SHA256 sha256 = SHA256.Create())
    {
      byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(data));

      // Convert the byte array to a string
      return Encoding.UTF8.GetString(bytes);
    }
  }
}
```

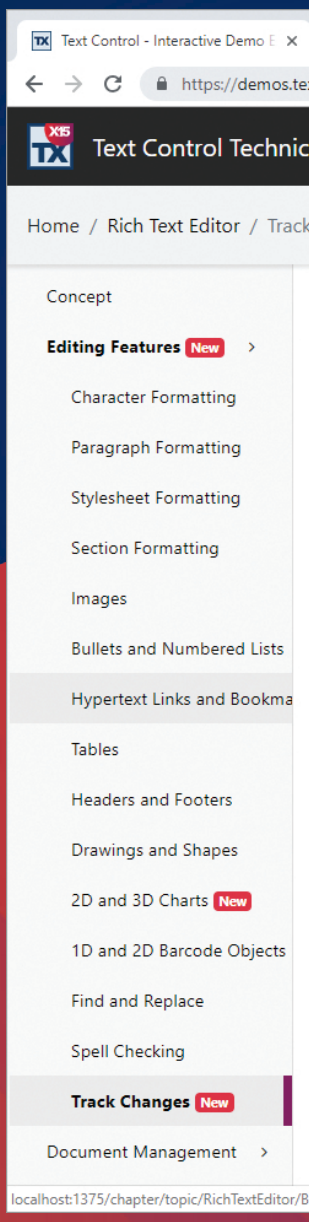
TEXT CONTROL

INTEGRATE DOCUMENT COLLABORATION

Integrate MS Word compatible track changes into cross-platform web applications. Share and review documents with a true WYSIWYG document editor.

See our technology live:

demos.textcontrol.com

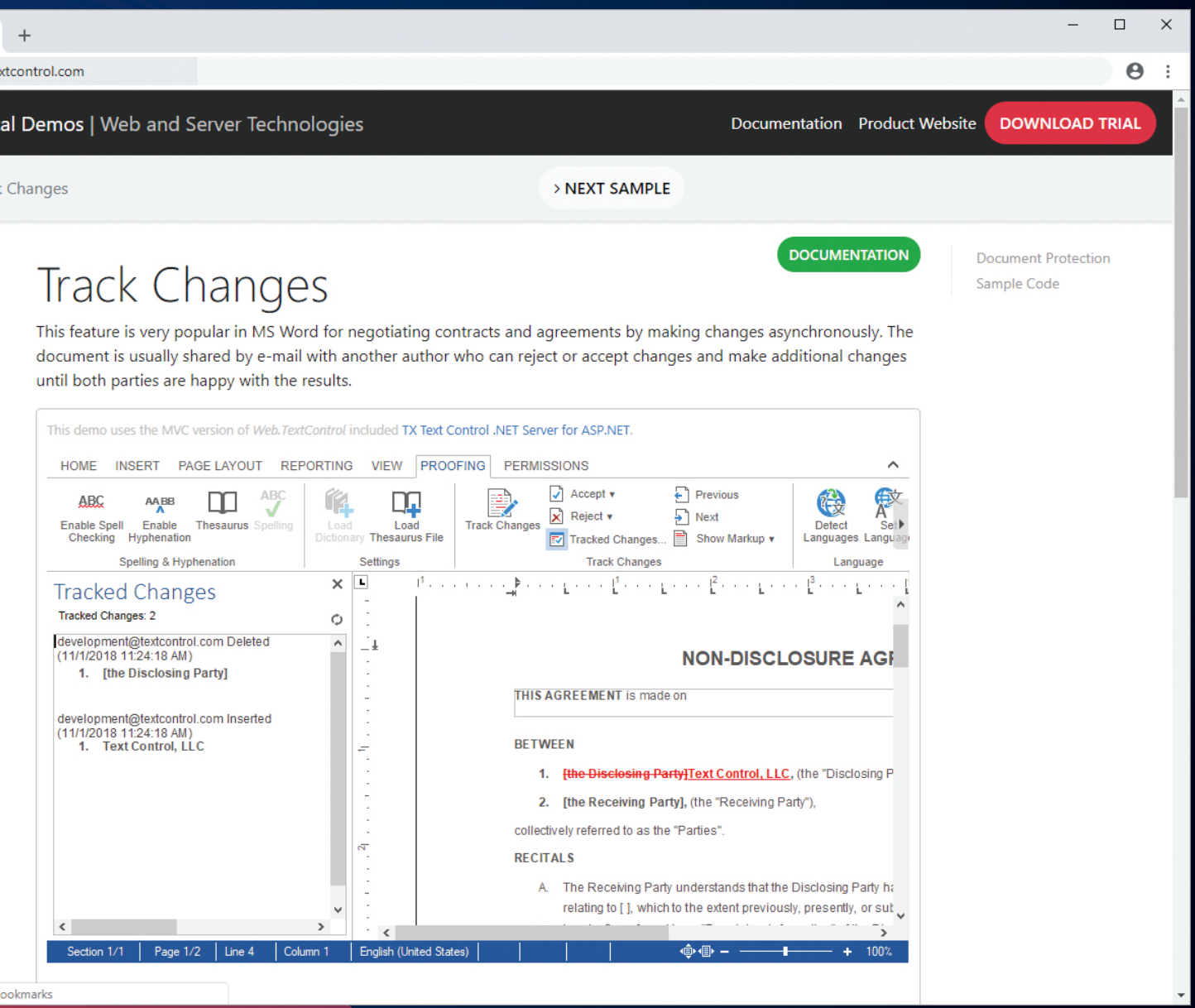


WE ARE CHANGING THE WAY YOU LOOK AT REPORTING

Visual Studio
Partner

TX Text Control X16 Released

Evaluate our technology and test the most sophisticated cross-browser, and true WYSIWYG, rich text editor. Merge MS Word compatible templates with JSON data and create pixel-perfect Adobe PDF documents on-the-fly. See what's possible today!



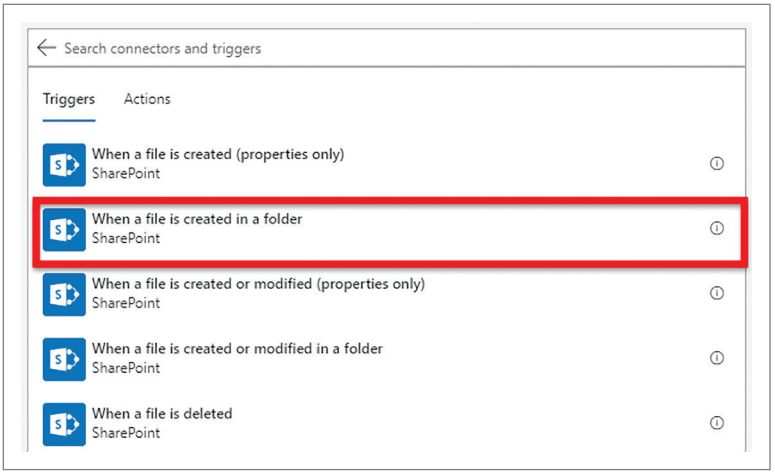


Figure 3 **The Logic App Action that Handles the File Creation Event in SharePoint**

The code in **Figure 2** implements the ComputeHashFunction Azure Function for computing a hash using the SHA256 algorithm. After reading the request body in the Run method, the function computes the hash using the SHA256 library available in the System.Security.Cryptography namespace. The hash value is returned as a UTF8-encoded string.

from the Azure Functions connector, but this time, instead of the several file attributes, pick File Content.

Once you've obtained the hash values for both file metadata and file content, it's time to store it on the blockchain network. For this purpose, I'm using Azure Blockchain Workbench (aka.ms/abcworkbench) as the runtime environment for smart contracts running on Ethereum. Blockchain Workbench is expected to support multiple blockchain platforms, but for now I'll stick to Ethereum.

Access to the digital ledger can be obtained by sending a message to the Azure Service Bus deployed as part of the Blockchain Workbench solution. An external system like a Logic App action can communicate with a smart contract hosted in Blockchain Workbench by sending a message to Service Bus. The message is picked by the Blockchain Workbench runtime and a new blockchain transaction is created, containing the message. Communication with Ethereum can happen only by generating a transaction that invokes a smart contract, as depicted in **Figure 5**.

To send a message from a Logic App flow to Service Bus you can use the Send message action on the Service Bus connector. A connection to an Azure Service Bus is identified by a connection name and a connection string. You can enter any convenient name

The Logic App flow is triggered when a new document is uploaded to a SharePoint site.

The Logic App flow is triggered when a new document is uploaded to a SharePoint site. This event is handled by one of the "When a file is created …" actions on the SharePoint connector (as depicted in **Figure 3**). To configure this action, after entering your authentication credentials for SharePoint, you have to specify the site address of the SharePoint site to monitor for new files, and the specific folder where files are uploaded. You can also set the frequency of polling this folder and checking for new files. A reasonable setting is to check once per minute.

The next step in the flow is, as already anticipated, the hashing of the uploaded file's content and metadata. As I've implemented the hashing function as an Azure Function, all you need to invoke this function is the Choose an Azure function action from the Azure Functions connector. Once you select ComputeHashFunction from the list of available functions, you'll be prompted to specify the request body that will be passed to the function itself. This is a JSON object that will be transferred in input to the function, obtaining its hash value as output. I've defined the following attributes as file metadata, as shown in **Figure 4**: contentType, etag, id, name and path.

The previous step is needed to hash the file metadata. Now I must hash also the entire file content, to preserve it in an immutable state in the blockchain network. As before, add another Choose an Azure function action as connection name, and you obtain the Service Bus connection string from the Azure Portal where it's deployed. The message to send to the Service Bus also requires the following parameters:

- requestId: A unique identifier for the request generated by the Logic App action
- processedDateTime: Timestamp of the request being sent
- userChainIdentifier: User address in the deployed Ethereum network
- applicationName: Name of the smart contract being invoked on Ethereum
- workflowName: Name of the workflow being invoked on Blockchain Workbench

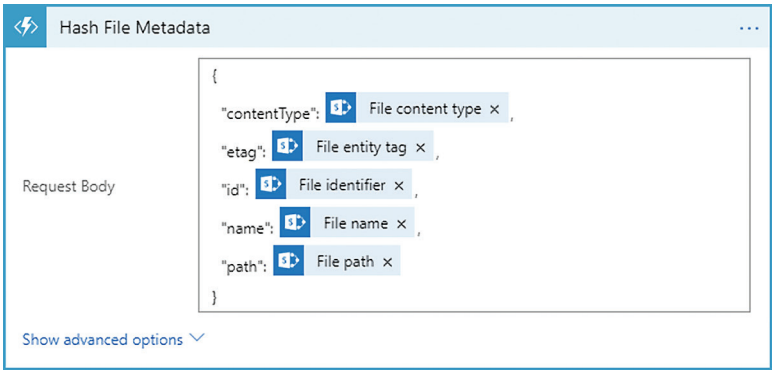


Figure 4 **Attributes in the Request Body for the Hash Function**

Manipulating Documents?

APIs to view, convert, annotate, compare, sign, assemble and search documents in your applications.

Try GroupDocs APIs for FREE

Download a Free Trial at

https://downloads.groupdocs.com

GROUPDOCS

GroupDocs.Viewer

View over 50 documents and image formats in any application using document viewer APIs.

GroupDocs.Annotation

Add annotations to specific words, phrases and any region of the document.

GroupDocs.Conversion

Fast batch document conversion APIs for any .NET, Java or Cloud app.

GroupDocs.Comparison

Compare two documents and get a difference summary report.

GroupDocs.Signature

Digitally sign Microsoft Word, Excel, PowerPoint and PDF documents.

GroupDocs.Assembly

Document automation APIs to create reports from templates and various data sources.

GroupDocs.Metadata

Organize documents with metadata within any cross platform application.

GroupDocs.Search

Transform your document search process for advance full text search capability.

- GroupDocs.Text
- GroupDocs.Editor
- GroupDocs.Parser
- GroupDocs.Watermark

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

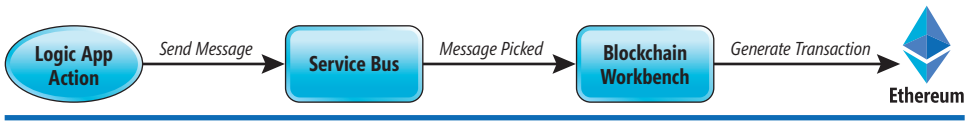


Figure 5 **Sending a Message to a Smart Contract**

I define these parameters as variables in the Logic App flow, by using the Initialize variable action from the Variables connector. The requestId variable can be set to guid, which is an expression that generates a unique GUID. The processedDateTime variable can be set to utcNow, which represents the current coordinated universal time. For userChainIdentifier, you can enter the address of a user in Blockchain Workbench that's authorized to run the smart contract, whereas applicationName and workflowName are defined as per name and workflow of the smart contract that processes this transaction.

The next section describes the smart contract for processing these messages sent by the Logic App flow. **Figure 6** summarizes the message body, in JSON format, to send to Service Bus. The expressions in <acute angle brackets> have to be replaced with the corresponding value.

Smart Contract for Processing Digital Assets

First of all, let me reinforce the message that digital assets aren't stored on the blockchain. Hash values of the file metadata and content are. In this article, I describe the storage of documents on SharePoint, which is a centralized service. In a "pure" blockchain deployment, you may want to obtain decentralization also of the storage service. The Interplanetary File System (IPFS) is a peer-to-peer hypermedia protocol (which I mentioned earlier) that provides decentralized file storage. Integration with IPFS is beyond the scope of this article, but if you're interested in knowing how this technology can help remove centralization of storage that isn't part of a block in a blockchain, you can refer to the "IPFS in Azure" video on Channel 9 (bit.ly/2CURRq0).

As I'm using Azure Blockchain Workbench for running my smart contract, I need two files:

- FileContract.sol to describe the smart contract itself, in Solidity programming language.
- FileContract.json to configure the workflow that's loaded in Azure Blockchain Workbench as an application.

The FileContract smart contract describes a file through its metadata, based on the values passed in the message sent by Logic App to Blockchain Workbench via Azure Service Bus. Here's a snippet of the source code of the smart contract that defines these parameters:

```
contract FileContract
{
  // File metadata
  string public FileId; // File identifier
  string public Location; // File path
  string public FileHash; // File content hash
  string public FileMetadataHash; // File Metadata Hash
  string public ContentType; // File content type
  string public Etag; // File entity tag
  string public ProcessedDateTime; // Timestamp
  address public User; // User address
```

To store the file metadata on a blockchain, I need a file structure defined, as follows:

```
struct File {
  string FileId;
  address FileContractAddress;
}
```

A file entity is identified by its file ID and the address on blockchain of the FileContract smart contract that contains the metadata. This structure is saved in a private collection defined as a dictionary, whose key is the FileId string. The mapping keyword in Solidity defines a dictionary and its key and value types as follows:

```
mapping(string => File) private Registry;
```

To save a file entity (its ID and metadata), I simply add the constituent values to the Registry dictionary in the Save method. For simplicity, I've omitted any necessary control on validity of file ID and contract address, and whether the file already exists in the registry. Here's the code:

```
function Save(string fileId, address fileContractAddress) public
{
  Registry[fileId].FileId = fileId;
  Registry[fileId].FileContractAddress = fileContractAddress;
}
```

The Verification Process

Users who need to verify their certificates with a third party do so by sharing the authenticity token (that is, the file contract address), which contains all the necessary information to verify that the document exists and is authentic and not counterfeited. **Figure 7** describes the parties and actions involved in the verification

Figure 6 **Structure of the Message Sent to Azure Service Bus**

```
{
  "requestId": "<The requestId variable>",
  "userChainIdentifier": "<User address in Azure Blockchain Workbench>",
  "applicationName": "<Smart contract name>",
  "workflowName": "<Smart contract workflow name>",
  "parameters": [
    {
      "name": "registryAddress",
      "value": "<Contract address in Azure Blockchain Workbench>"
    },
    {
      "name": "fileId",
      "value": "<File identifier>"
    },
    {
      "name": "location",
      "value": "<File path>"
    },
    {
      "name": "fileHash",
      "value": "<File content hash>"
    },
    {
      "name": "fileMetadataHash",
      "value": "<File metadata hash>"
    },
    {
      "name": "contentType",
      "value": "<File content type>"
    },
    {
      "name": "etag",
      "value": <File entity tag>
    },
    {
      "name": "processedDateTime",
      "value": "<The processedDateTime variable>"
    }
  ],
  "connectionId": 1,
  "messageSchemaVersion": "1.0.0",
  "messageName": "CreateContractRequest"
}
```

process. The user retrieves the certificate to verify from its location (1) and initiates a new transaction on the blockchain network, transferring the authenticity token (2) to the verification authority. The authority obtains the signed content and metadata of the certificate being verified (3), which is stored on the immutable digital ledger, and then compares them with the equivalent hash values from the off-chain copy. If the values match, the document is verified (4).

Once documents and unstructured data are signed and verified—and a hash of their content and metadata are stored on a blockchain—it creates an immutable and independent, verifiable record of transactions. This process is referred to as proof of existence and proof of authenticity of digital assets.

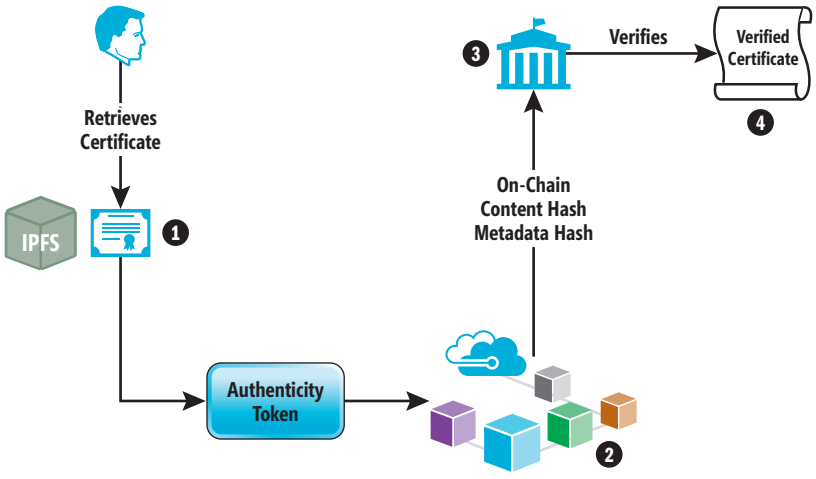


Figure 7 **The Verification Actors and Process**

Proof of existence refers to creating an unalterable date and time stamp for a specific object. This means that you can prove that a certain information object—like an e-mail, document or image—existed at a certain point in time.

Proof of authenticity asserts that an object is authentic—that is, it hasn't been changed since it was stored at the indicated time instant. This is accomplished by digitally signing an object and thus creating a hash, its unique identifier. The identifier then gets committed into the distributed blockchain ledger, and the transaction gets time-stamped, as well. Because every entry in the blockchain is immutable, this means you have proof that this specific object existed at a certain point in time.

Using the same approach, an object can be verified and validated. A flow similar to the one I described for the signing process creates a unique identifier and verifies this unique identifier against the blockchain ledger. If there's a match, the smart contract returns the original hash value. If not, the document being verified isn't identical to the original copy and should not be trusted implicitly. Thus, you're able, beyond any doubt, to prove that the document, or any digital object, is authentic and existed at a certain moment in time.

The FileContract smart contract exposes a GetFile method that, given a file ID in input, returns its contract address on the blockchain. From the file contract address it's possible to obtain the file content and metadata hash values and compare them with the hash values of the document being verified, like so:

```
function GetFile(string fileId) public constant
returns(address fileContractAddress)
{
  return Registry[fileId].FileContractAddress;
}
```

Wrapping Up

Why use blockchain to sign and verify digital assets, when solutions for electronic signature already exist and are broadly adopted in the industry? In short, blockchains remove the need for a central certificate authority or central time-stamping server and enable digital signatures stored on a blockchain to live independently of the object being signed. This opens to opportunities for parallel signing and independent verification, with or without the object itself.

Traditional e-signing solutions store digital signatures inside the document. This means that whoever needs to check if a document is signed will have full read access to all the content in the document. Also, because the document changes with each signature, signing documents in parallel isn't possible—everybody needs to sign the document sequentially. By signing documents on a blockchain, the object itself isn't changed by the signature, and this enables you to sign documents in parallel and implement business rules based on mandates, 4-eyes, majority vote, seniority and the like.

Finally, but not less important, you can register multiple actions in a sequence on a blockchain. Each registration is linked to a specific case, document and task performed by the parties involved, creating a chain of transactions: an auditable trail. This audit trail can be verified by authorized third parties, providing transparency, compliance and, most importantly, trust.

To learn more about the Azure Blockchain Development Kit, you can find a host of videos on Channel 9, under the "Block Talk" show (aka.ms/bcblocktalk). If you wish, you can also stay up-to-date with the latest announcements from the Azure Blockchain product group by following the @MSFTBlockchain Twitter handle (twitter.com/MSFTBlockchain).

The Azure Blockchain Development Kit project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant Microsoft the rights to use your contribution. When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the request appropriately (that is, add labels or comments to your code). ■

Stefano Tempesta *is a Microsoft Regional Director, MVP on AI and Business Applications, and member of the Blockchain Council. A regular speaker at international IT conferences, including Microsoft Ignite and Tech Summit, Tempesta's interests extend to blockchain and AI-related technologies. He created "Blogchain Space" (blogchain.space), a blog about blockchain technologies, writes for* MSDN Magazine *and* MS Dynamics World, *and publishes machine learning experiments on the Azure AI Gallery.*

Thanks to the following technical expert for reviewing this article: *Jonathan Waldman*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

April 22-26, 2019 | Hyatt Regency New Orleans

Spice Up Your Coding Skills in the Bayou

New Orleans

Intense Developer Training Conference

In-depth Technical Content On:

- AI, Data and Machine Learning
- Cloud, Containers and Microservices
- Delivery and Deployment
- Developing New Experiences
- DevOps in the Spotlight
- Full Stack Web Development
- .NET Core and More

Register by March 22 to save $300!

Use Promo Code MSDN

YOUR ADVENTURE STARTS HERE

vslive.com/neworleans

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360 an 1105 MEDIA company

Agenda-at-a-Glance

#VSLive

DevOps in the Spotlight | Cloud, Containers and Microservices | AI, Data and Machine Learning | Developing New Experiences | Delivery and Deployment | .NET Core and More | Full Stack Web Development

START TIME	END TIME				
		Pre-Conference Full Day Hands-On Labs: Monday, April 22, 2019 *(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: Cross-Platform Mobile Development in a Day with Xamarin and Xamarin.Forms - *Marcel de Vries & Roy Cornelissen*		**HOL02** Full Day Hands-On Lab: Building a Modern DevOps Pipeline on Microsoft Azure with ASP.NET Core and Azure DevOps - *Brian Randell & Mickey Gousset*	
6:45 PM	9:00 PM	Dine-A-Round			
START TIME	**END TIME**	**Day 1: Tuesday, April 23, 2019**			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**T01** Moving to ASP.NET Core 2.X - *Philip Japikse*	**T02** Building Your First Mobile App with Xamarin Forms - *Robert Green*	**T03** Crack the Code: How Machine Learning Models Work - *Jen Underwood*	**T04** Unit Testing Makes Me Faster: Convincing Your Boss, Your Co-Workers, and Yourself - *Jeremy Clark*
9:30 AM	10:45 AM	**T05** Getting Started with ASP.NET Core 2.0 Razor Pages - *Walt Ritscher*	**T06** (WPF + WinForms) * .NET Core = Modern Desktop - *Oren Novotny*	**T07** Containers Demystified - *Robert Green*	**T08** Azure DevOps in the Cloud and in Your Data Center - *Brian Randell*
11:00 AM	12:00 PM	Keynote: To Be Announced - *Donovan Brown, Principal DevOps Manager, Cloud Developer Advocacy Team, 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** Diving Deep Into ASP.NET Core 2.x - *Philip Japikse*	**T10** Cross-Platform Development with Xamarin, C#, and CSLA .NET - *Rockford Lhotka*	**T11** How to Avoid Building Bad Predictive Models - *Jen Underwood*	**T12** To Be Announced
3:00 PM	4:15 PM	**T13** Up and Running with Angular in 60 Minutes - *Justin James*	**T14** Programming with PowerApps and Microsoft Flow - *Walt Ritscher*	**T15** Azure DevOps and AKS - *Brian Randell*	**T16** Get Func-y: Understanding Delegates in .NET - *Jeremy Clark*
4:15 PM	5:30 PM	Welcome Reception			
START TIME	**END TIME**	**Day 2: Wednesday, April 24, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**W01** Angular Unit Testing from the Trenches - *Justin James*	**W02** Power BI: What Have You Done for Me Lately? - *Andrew Brust*	**W03** Secure Your App with Azure AD B2C - *Oren Novotny*	**W04** Putting the Ops into DevOps - *Mickey Gousset*
9:30 AM	10:45 AM	**W05** TypeScript: Moving Beyond the Basics - *Allen Conway*	**W06** AI and Analytics with Apache Spark on Azure Databricks - *Andrew Brust*	**W07** Architecting and Developing Microservices Apps - *Eric D. Boyd*	**W08** Architecting Systems for DevOps and Continuous Delivery - *Marcel de Vries*
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: Hybrid Web Frameworks - *Allen Conway*	**W10** Fast Focus: Graph DB from SQL to Cosmos - *Leonard Lobel*		**W11** Fast Focus: What's New in EF Core 2.x - *Jim Wooley*
2:00 PM	2:20 PM	**W12** Fast Focus: Ultimate Presentation Formula for Nerds - *Justin James*	**W13** Fast Focus: Serverless of Azure 101 - *Eric D. Boyd*		**W14** Fast Focus: Scrum in 20 Minutes - *Benjamin Day*
2:30 PM	3:45 PM	**W15** Migrating from AngularJS to Angular + TypeScript - *Allen Conway*	**W16** Introduction to Azure Cosmos DB - *Leonard Lobel*	**W17** Demystifying Microservice Architecture - *Miguel Castro*	**W18** From One Release per Quarter to 30 Times a Day - *Marcel de Vries*
4:00 PM	5:15 PM	**W19** Getting Pushy with SignalR and Reactive Extensions - *Jim Wooley*	**W20** Modern SQL Server Security Features for Developers - *Leonard Lobel*	**W21** Make Your App See, Hear and Think with Cognitive Services - *Roy Cornelissen*	**W22** Monitor Your Applications and Infrastructure - *Eric D. Boyd*
6:45 PM	9:00 PM	VSLive! Event			
START TIME	**END TIME**	**Day 3: Thursday, April 25, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**TH01** Advanced Fiddler Techniques - *Robert Boedigheimer*	**TH02** Busy Developer's Guide to Flutter - *Ted Neward*	**TH03** To Be Announced	**TH04** UX Design Fundamentals: What Do Your Users Really See? - *Billy Hollis*
9:30 AM	10:45 AM	**TH05** SASS and CSS for Developers - *Robert Boedigheimer*	**TH06** The Next Frontier - Conversational Bots - *Sam Basu*	**TH07** Porting Your Code from .NET Framework to .NET Standard - *Rockford Lhotka*	**TH08** WSL, Bash, Developers Love Linux! -
11:00 AM	12:15 PM	**TH09** Upload and Store a File Using MVC - *Paul Sheriff*	**TH10** Essential Tools for Xamarin Developers! - *Sam Basu*	**TH11** What's New in C# 8 - *Jason Bock*	**TH12** How to Interview a Developer - *Billy Hollis*
12:15 PM	1:30 PM	Lunch			
1:30 PM	2:45 PM	**TH1** Blazing the Web - Building Web Applications in C# - *Jason Bock*	**TH14** Entity Framework for Enterprise Applications - *Benjamin Day*	**TH15** Exposing an Extensibility API for your Applications - *Miguel Castro*	**TH16** To Be Announced
3:00 PM	4:15 PM	**TH17** What's New in Bootstrap 4 - *Paul Sheriff*	**TH18** Busy .NET Developer's Guide to Python - *Ted Neward*	**TH19** Improving Code Quality with Static Analyzers - *Jim Wooley*	**TH20** Unit Testing & Test-Driven Development (TDD) for Mere Mortals - *Benjamin Day*
START TIME	**END TIME**	**Post-Conference Workshops: Friday, April 26, 2019** *(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: DI for the Dev Guy - *Miguel Castro*	**F02** Workshop: SQL Server for Developers: The Grand Expedition - *Andrew Brust & Leonard Lobel*		**F03** Workshop: Cross-Platform C# Using .NET Core, Kubernetes, and WebAssembly - *Rockford Lhotka & Jason Bock*

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!

Support Vector Machines Using C#

James McCaffrey

A support vector machine (SVM) is a software system that can make predictions using data. The original type of SVM was designed to perform binary classification, for example predicting whether a person is male or female, based on their height, weight, and annual income. There are also variations of SVMs that can perform multiclass classification (predicting one of three or more classes) and regression (predicting a numeric value).

In this article I present a complete working example of an SVM that's implemented using only raw C# without any external libraries. A good way to see where this article is headed is to examine the demo program in **Figure 1**.

The demo begins by setting up eight dummy training data items. Each item has three predictor values, followed by the class to predict encoded as -1 or +1. The dummy data doesn't represent a real problem, so the predictor values have no particular meaning. In a realistic SVM problem, it's important to normalize the predictor values, usually by applying min-max normalization so that all values are scaled between 0.0 and 1.0.

The demo creates an SVM classifier that uses a polynomial kernel function. (I'll explain kernel functions shortly.) Training the SVM model generated three support vectors: (4, 5 7), (7, 4, 2) and (9, 7, 5). These are training data items [0], [1] and [4]. Each support vector has an associated weight value: (-0.000098, -0.000162, 0.000260). Additionally, an SVM model has a single value called a bias, which is -2.505727 for the demo data.

The support vectors, weights and biases define the trained SVM model. The demo program uses the model to generate predicted class values for each of the eight training items. A decision value that's negative corresponds to a predicted class label of -1 and a positive decision value corresponds to a predicted class of +1. Therefore, the trained model correctly predicts all eight of the training items. This isn't too surprising because the problem is so simple.

The demo concludes by predicting the class for a new, previously unseen item with predictor values (3, 5, 7). The computed decision value is -1.274 and therefore the predicted class label is -1. The computation of the decision value is explained later in this article.

This article assumes you have intermediate or better programming skill with C#, but doesn't assume you know anything about SVMs. The demo program is coded using C# and though it's complicated, you should be able to refactor it to another language, such as Java or Python, if you wish.

This article discusses:

- The support vector machine (SVM) demo program
- Understanding and using SVMs
- Kernel functions
- The sequential minimal optimization algorithm

Technologies discussed:

C#, Visual Studio 2107

Code download available at:

msdn.com/magazine/0319magcode

The code for the demo program is too long to present in its entirety in this article, but the complete source code is available in the accompanying file download. All normal error checking has been removed to keep the main ideas as clear as possible.

Overall Program Structure

The structure of the demo program is shown in **Figure 2**. All the control logic is contained in a single Main method. Program-defined class SupportVectorMachine declares all member fields as public so you can more easily inspect them programmatically.

I used Visual Studio 2017 to create the demo program, but there are no significant .NET Framework dependencies so any version of Visual Studio will work fine. I created a new C# console application and named it SVM_CSharp. After the template code loaded into the editor window, I removed all unneeded using statements and then added a reference to the Collections.Generic assembly. In the Solution Explorer window, I right-clicked on file Program.cs and renamed it to SVM_Program.cs and allowed Visual Studio to automatically rename class Program.

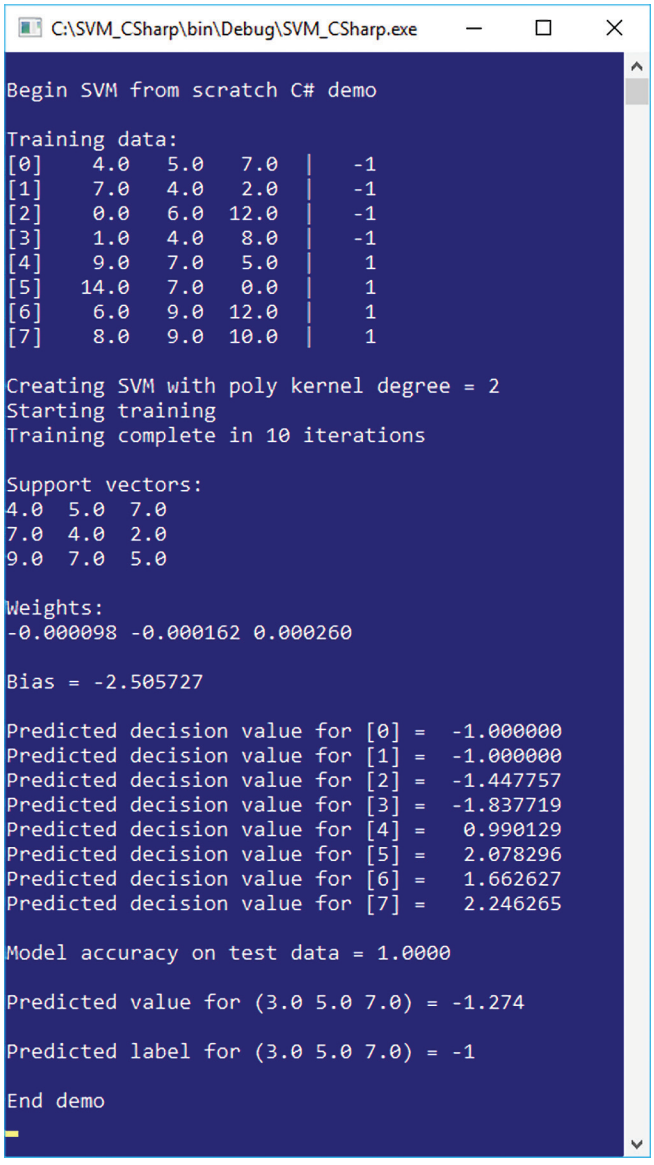


Figure 1 **SVM Demo Program in Action**

Using the SVM

The demo program sets up eight hardcoded training items:

```
double[][] train_X = new double[8][] {
  new double[] { 4,5,7 },
  ...
  new double[] { 8,9,10 }  };
int[] train_y = new int[8]{ -1, -1, -1, -1, 1, 1, 1, 1 };
```

In a non-demo scenario, you should normalize the predictor values, and you'd likely store your data in a text file. The SVM classifier is created like so:

```
var svm = new SupportVectorMachine("poly", 0);
svm.gamma = 1.0;
svm.coef = 0.0;
svm.degree = 2;
```

The "poly" argument is really a dummy value because the SVM is hardcoded to use a polynomial kernel function. The 0 argument is a seed value for the random component of the training algorithm. The gamma, coef (also called constant), and degree arguments are parameters for the polynomial kernel function. Next, parameters for the training algorithm are specified:

```
svm.complexity = 1.0;
svm.epsilon = 0.001;
svm.tolerance = 0.001;
int maxIter = 1000;
```

All of these values are hyperparameters that must be determined by trial and error. The main challenge when using any implementation of an SVM is understanding which kernel to use, the kernel parameters and the training parameters. Training is performed by a single statement:

```
int iter = svm.Train(train_X, train_y, maxIter);
```

Figure 2 **Demo Program Structure**

```
using System;
using System.Collections.Generic;
namespace SVM_CSharp
{
  class SVM_Program
  {
    static void Main(string[] args)
    {
      // Set up training data
      // Create SVM object, set parameters
      // Train the SVM
      // Display SVM properties
      // Use trained SVM to make a prediction
    }
  }
  public class SupportVectorMachine
  {
    // All member fields are declared public

    public SupportVectorMachine(string kernelType,
      int seed) . .
    public double PolyKernel(double[] v1, double[] v2) . .
    public double ComputeDecision(double[] input) . .
    public int Train(double[][] X_matrix,
      int[] y_vector, int maxIter) . .
    public double Accuracy(double[][] X_matrix,
      int[] y_vector) . .
    private bool TakeStep(int i1, int i2,
      double[][] X_matrix, int[] y_vector) . .
    private int ExamineExample(int i2, double[][] X_matrix,
      int[] y_vector) . .
    private double ComputeAll(double[] vector,
      double[][] X_matrix, int[] y_vector) . .
  }
}
```

Training an SVM classifier is an iterative process and method Train returns the actual number of iterations that were executed, as an aid for debugging when things go wrong. After training, the SVM object holds a List<double[]> collection of the support vectors, an array that holds the model weights (one per support vector) and a single bias value. They're displayed like this:

```
foreach (double[] vec in svm.supportVectors) {
  for (int i = 0; i < vec.Length; ++i)
    Console.Write(vec[i].ToString("F1") + "  ");
  Console.WriteLine("");
}
for (int i = 0; i < svm.weights.Length; ++i)
  Console.Write(svm.weights[i].ToString("F6") + " ");
Console.WriteLine("");
Console.WriteLine("Bias = " + svm.bias.ToString("F6") + "\n");
```

The demo concludes by making a prediction:

```
double[] unknown = new double[] { 3, 5, 7 };
double predDecVal = svm.ComputeDecision(unknown);
Console.WriteLine("Predicted value for (3.0 5.0 7.0) = " +
  predDecVal.ToString("F3"));
int predLabel = Math.Sign(predDecVal);
Console.WriteLine("Predicted label for (3.0 5.0 7.0) = " +
  predLabel);
```

The decision value is type double. If the decision value is negative, the predicted class is -1 and if the decision value is positive, the predicted class is +1.

Understanding SVMs

SVMs are quite difficult to understand, and they're extremely difficult to implement. Take a look at the graph in **Figure 3**. The goal is to create a rule that distinguishes between the red data and the blue data. The graph shows a problem where the data has just two dimensions (number of predictor variables) only so that the problem can be visualized, but SVMs can work with data with three or more dimensions.

An SVM works by finding the widest possible lane that separates the two classes and then identifies the one or more points from each class that are closest to the edge of the separating lane.

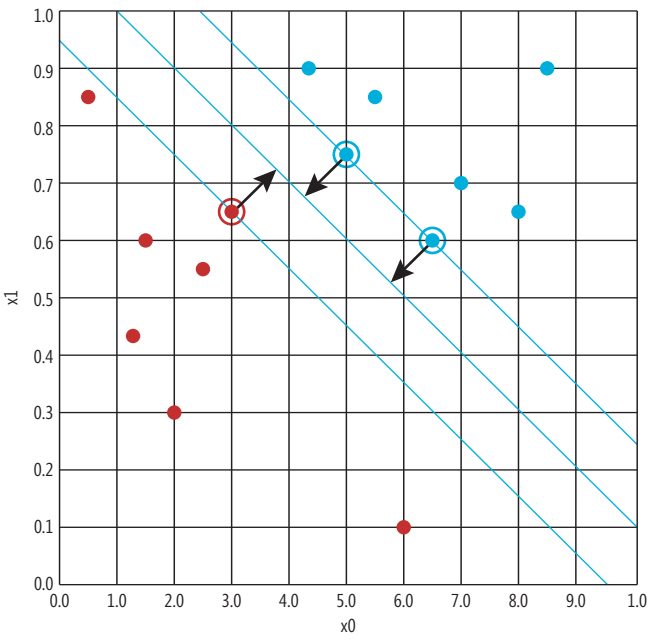


Figure 3 **Basic SVM Concepts**

To classify a new, previously unseen data point, all you have to do is see which side of the middle of the lane the new point falls. In **Figure 3**, the circled red point at (0.3, 0.65) and the circled blue points at (0.5, 0.75) and (0.65, 0.6) are called the support vectors. In my head, however, I think of them as "support points" because I usually think of vectors as lines.

There are three major challenges that must be solved to implement a useable SVM. First, what do you do if the data isn't linearly separable as it is in **Figure 3**? Second, just how do you find the support vectors, weights and biases values? Third, how do you deal with training data points that are anomalous and are on the wrong side of the boundary lane?

> SVMs are quite difficult to understand, and they're extremely difficult to implement.

As this article shows, you can deal with non-linearly separable data by using what's called a kernel function. You can determine the support vectors, weights and biases using an algorithm called sequential minimal optimization (SMO). And you can deal with inconsistent training data using an idea known as complexity, which penalizes bad data.

Kernel Functions

There are many different types of kernel functions. Briefly, a kernel function takes two vectors and combines them in some way to produce a single scalar value. Although it's not obvious, by using a kernel function, you can enable an SVM to handle data that's not linearly separable. This is called "the kernel trick."

Suppose you have a vector v1 = (3, 5, 2) and a second vector v2 = (4, 1, 0). A very simple kernel is called the linear kernel and it returns the sum of the products of the vector elements:

$$K(v1, v2) = (3 * 4) + (5 * 1) + (2 * 0) = 17.0$$

Many kernel functions have an optional scaling factor, often called gamma. For the previous example, if gamma is set to 0.5, then:

$$K(v1, v2) = 0.5 * [(3 * 4) + (5 * 1) + (2 * 0)] = 8.5$$

The demo program uses a polynomial kernel with degree = 2, gamma = 1.0 and constant = 0. In words, you compute the sum of products, then multiply by gamma, then add the constant, then raise to the degree. For example:

$$K(v1,v2) = [1.0 * ((3*4) + (5*1) + (2*0)) + 0]\text{^}2 = (1*17 + 0)\text{^}2 = 289.0$$

The polynomial kernel is implemented by the demo program like so:

```
public double PolyKernel(double[] v1, double[] v2)
{
  double sum = 0.0;
  for (int i = 0; i < v1.Length; ++i)
    sum += v1[i] * v2[i];
  double z = this.gamma * sum + this.coef;
  return Math.Pow(z, this.degree);
}
```

The values of gamma, degree and constant (named coef to avoid a name clash with a language keyword) are class members and their values are supplied elsewhere. The demo program hard codes the

kernel function so if you want to experiment with a different function, you'll have to implement it yourself.

A common alternative to the polynomial kernel is the radial basis function (RBF) kernel. In words, you sum the squared differences between elements, multiply by negative gamma, then raise to e (Euler's number, approximately 2.718). In code the RBF kernel could look like:

```
public double RbfKernel(double[] v1, double[] v2)
{
  double sum = 0.0;
  for (int i = 0; i < v1.Length; ++i)
    sum += (v1[i] - v2[i]) * (v1[i] - v2[i]);
    return Math.Exp(-this.gamma * sum);
}
```

Notice that different kernel functions have different parameters. The linear and RBF kernels only require gamma, but the polynomial kernel requires gamma, degree and constant. The choice of kernel function to use, and the values of the parameters that apply to the kernel function being used, are free parameters and must be determined by trial and error. All machine learning classification techniques have hyperparameters, but SVMs tend to be particularly sensitive to their hyperparameter values.

The Sequential Minimal Optimization Algorithm

There are many algorithms that can be used to determine the support vectors for an SVM problem. The SMO algorithm is the most common. The demo program follows the original explanation of SMO given in the 1998 research paper, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," which can be found in many places on the Internet.

The SMO algorithm is very complex and a full explanation would require roughly 200 pages (I know because I once reviewed an entire book dedicated just to SMO). SMO has three key functions: a top-level Train function that calls a helper ExamineExample function, which calls a helper TakeStep function.

The signature of TakeStep is: private bool TakeStep(int i1, int i2, double[][] X_matrix, int[] y_vector). Parameter X_matrix holds the training data predictor values. Parameter y_vector holds the training data target values, which are either -1 or +1. The i1 and i2 parameters are a first index and a second index pointing into the training data. On each call to TakeStep, the algorithm attempts to find a better solution and returns true if an improvement is found using the two training data items, false otherwise.

The signature of ExamineExample is: private int ExamineExample(nt i2, double[][] X_matrix, int[] y_vector). The function returns the number of changes that occurred so that TakeStep can determine if progress has been made.

Both TakeStep and ExamineExample use a class-scope variable named complexity. Larger values of complexity increasingly penalize outlier training data and force the SMO algorithm to try to find a solution that deals with them, at the expense of model overfitting. Because complexity is a parameter used by SMO, it will always be present, unlike parameters associated with the kernel function used, which might be present or absent.

The TakeStep function uses a class-scope variable named epsilon, and the ExamineExample function uses a class-scope variable named tolerance. Both are small values, set to 0.001 by default in the demo. Epsilon is used internally to determine when to stop iterating, which in turn affects the number of support vectors found. Tolerance is used when computing error. The values of epsilon and tolerance are free parameters and the effect of changing them varies quite a bit (from a very small to a very large effect) depending on the particular problem at which you're looking.

The code for method Train is presented in **Figure 4**. The method is iterative and returns the number of iterations that were performed. The method accepts a maxIter parameter to set a hard limit on the number of iterations performed. In theory, the SMO algorithm will always converge and stop iterating, but theory doesn't always match practice with an algorithm as complex as SMO.

In addition to explicitly returning the number of iterations performed, Train finds and stores the indices of the training data items that are support vectors. After the number of support vectors is known, the array that holds the values of the weights can be allocated. The weight values are alpha values that are non-zero.

The Train method has many possible customization points. For example, the demo code stores support vectors as a List<double[]> collection. An alternative is to store just the indices of the support vectors, in a List<int> collection or in an int[] array object. Examining the Train method carefully is the best way to start to understand SVMs and the SMO algorithm.

Figure 4 **The Training Method**

```
public int Train(double[][] X_matrix, int[] y_vector, int maxIter)
{
  int N = X_matrix.Length;
  this.alpha = new double[N];
  this.errors = new double[N];
  int numChanged = 0;
  bool examineAll = true;
  int iter = 0;

  while (iter < maxIter && numChanged > 0 || examineAll == true) {
    ++iter;
    numChanged = 0;
    if (examineAll == true) {
      for (int i = 0; i < N; ++i)
        numChanged += ExamineExample(i, X_matrix, y_vector);
    }
    else {
      for (int i = 0; i < N; ++i)
        if (this.alpha[i] != 0 && this.alpha[i] !=
          this.complexity)
          numChanged += ExamineExample(i, X_matrix, y_vector);
    }

    if (examineAll == true)
      examineAll = false;
    else if (numChanged == 0)
      examineAll = true;
  } // While

  List<int> indices = new List<int>();  // support vectors
  for (int i = 0; i < N; ++i) {
    // Only store vectors with Lagrange multipliers > 0
    if (this.alpha[i] > 0) indices.Add(i);
  }

  int num_supp_vectors = indices.Count;
  this.weights = new double[num_supp_vectors];
  for (int i = 0; i < num_supp_vectors; ++i) {
    int j = indices[i];
    this.supportVectors.Add(X_matrix[j]);
    this.weights[i] = this.alpha[j] * y_vector[j];
  }
  this.bias = -1 * this.bias;
  return iter;
}
```

Understanding the SVM Mechanism

If you refer to **Figure 1**, you'll see the trained SVM has three support vectors: (4, 5, 7), (7, 4, 2) and (9, 7, 5). And the model has three weight values = (-0.000098, -0.000162, 0.000260) and bias = -2.506. The decision value for input (3, 5, 7) is computed by calculating the value of the kernel function with each of the three support vectors, then multiplying each kernel value by its corresponding weight, summing, then adding the bias:

```
x = (3, 5, 7)
sv1 = (4, 5, 7)
sv2 = (7, 4, 2)
sv3 = (9, 7, 5)

K(x, sv1) * wt1 = 7396.0 * -0.000098 = -0.725
K(x, sv2) * wt2 = 3025.0 * -0.000162 = -0.490
K(x, sv3) * wt3 = 9409.0 * 0.000260 = 2.446

decision = -0.725 + -0.490 + 2.446 + -2.506 = -1.274
prediction = Sign(decision) = -1
```

Notice that if the predictor values are not normalized, as in the demo, the values of the kernels can become very large, forcing the values of the weights to become very small, which could possibly lead to arithmetic errors.

The SVM mechanism points out strengths and weaknesses of the technique. SVM focuses only on the key support vectors, and therefore tends to be resilient to bad training data. When the number of support vectors is small, an SVM is somewhat interpretable, an advantage compared to many other techniques. Compared to many other classification techniques, notably neural networks, SVMs can often work well with limited training data, but SVMs can have trouble dealing with very large training datasets. The major disadvantages of SVMs is that SVMs are very complex and they require you to specify the value of many hyperparameters.

Wrapping Up

As this article shows, implementing a support vector machine is quite complex and difficult. Because of this, there are very few SVM library implementations available. Most SVM libraries are based on a C++ implementation called LibSVM, which was created by a group of researchers. Because calling C++ is often difficult, there are several libraries that provide wrappers over LibSVM, written in Python, Java, C# and other languages.

By experimenting with the code presented in this article, you'll gain a good understanding of exactly how SVMs work and be able to use a library implementation more effectively. Because the code in this article is self-contained and simplified, you'll be able to explore alternative kernel functions and their parameters, and the SMO training algorithm parameters epsilon, tolerance, and complexity. ■

Dr. James McCaffrey *works for Microsoft Research in Redmond, Wash. He has worked on several key 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: *Yihe Dong, Chris Lee*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

June 9-13, 2019 | Hyatt Regency Cambridge

SPARK YOUR CODE REVOLUTION IN HISTORIC BOSTON

INTENSE DEVELOPER TRAINING CONFERENCE

In-depth Technical Content On:

- AI, Data and Machine Learning
- Cloud, Containers and Microservices
- Delivery and Deployment
- Developing New Experiences
- DevOps in the Spotlight
- Full Stack Web Development
- .NET Core and More

AGENDA COMING SOON!

New This Year!

On-Demand Session Recordings Now Available

Get on-demand access for one full year to all keynotes and sessions from Visual Studio Live! Boston, including everything Tuesday – Thursday at the conference.

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360 an 1105 MEDIA company

#VSLive

Hear From Your Peers!

See what past attendees had to say about Visual Studio Live!

Timothy Franzke

Software Engineer, Ascend Learning

"This conference has done a great job of getting attendees to mingle and network. I've enjoyed the talks but the social aspects of this conference have been great. I've met a lot of great people and hope to see them at future conferences."

Katie Gray

Senior Lecturer, The University of Texas at Austin

"I have loved learning about all of the latest offerings in the Visual Studio ecosystem from experts in the field. All of the sessions provided candid information about what is new and what is coming. It has also been great to network with fellow developers to hear about what they are working on at their jobs."

Ritesh Salot

Software Architect, Netsmart Technologies

"Learning latest and greatest stuff from industry experts will allow me to come up with new directions to find solutions for our business and technology challenges."

Register by April 19 & Save Up To $400!

Use Promo Code MSDN

Your Adventure Starts Here!

vslive.com/boston

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

DINO ESPOSITO

Hierarchical Blazor Components

As the latest framework to join the single-page application (SPA) party, Blazor had the opportunity to build on the best characteristics of other frameworks, such as Angular and React. While the core concept behind Blazor is to leverage C# and Razor to build SPA applications, one aspect clearly inspired by other frameworks is the use of components.

Blazor components are written using the Razor language, in much the same way that MVC views are built, and this is where things get really interesting for developers. In ASP.NET Core you can reach unprecedented levels of expressivity through new language artifacts called tag helpers. A tag helper is a C# class instructed to parse a given markup tree to turn it into valid HTML5. All of the branches you may face while creating a complex, made-to-measure chunk of HTML are handled in code, and all that developers write in text files is plain markup. With tag helpers the amount of code snippets decreases significantly. Tag helpers are great, but still present some programming wrinkles that Blazor components brilliantly iron out. In this article, I'll build a new Blazor component that presents a modal dialog box through the services of the Bootstrap 4 framework. In doing so, I'll deal with Blazor-templated components and cascading parameters.

Figure 1 **The Bootstrap Markup for Modal Dialogs**

```
<button type="button" class="btn btn-primary"
        data-toggle="modal"
        data-target="#exampleModal">
  Open modal
</button>

<div class="modal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Modal title</h5>
        <button type="button" class="close" data-dismiss="modal">
          <span>&times;</span>
        </button>
      </div>
      <div class="modal-body">
        <p>Modal body text goes here.</p>
      </div>
      <div class="modal-footer">
        <button type="button"
                class="btn btn-secondary"
                data-dismiss="modal">Close</button>
      </div>
    </div>
  </div>
</div>
```

Code download available at bit.ly/2FdGZat.

Wrinkles of Tag Helpers

In my book, "Programming ASP.NET Core" (Microsoft Press, 2018), I present a sample tag helper that does nearly the same job discussed earlier. It turns some ad hoc non-HTML markup into Bootstrap-specific markup for modal dialog boxes (see bit.ly/2RxmWJS).

Any transformation between the input markup and the desired output is performed via C# code. A tag helper, in fact, is a plain C# class that inherits from the base class TagHelper and overrides a single method. The problem is that the transformation and markup composition must be expressed in code. While this adds a lot of flexibility, any change also requires a compile step. In particular, you need to use C# code to describe a DIV tree with all of its sets of attributes and child elements.

In Blazor, things come much easier as you don't need to resort to tag helpers in order to create a friendlier markup syntax for sophisticated elements, such as a Bootstrap modal dialog box. Let's see how to create a modal component in Blazor.

Modal Dialog Boxes

The idea is to set up a Blazor reusable component that wraps the Bootstrap modal dialog component. **Figure 1** presents the familiar HTML5 markup tree required for Bootstrap (both 3.x and 4.x versions) to work.

No Web developer is happy to reiterate that chunk of markup over and over again across multiple views and pages. Most of the markup is pure layout and the only variable information is the text to display and perhaps some style and buttons. Here's a more expressive markup that's easier to remember:

```
<Modal>
  <Toggle class="btn"> Open </Toggle>
  <Content>
    <HeaderTemplate> ... </HeaderTemplate>
    <BodyTemplate> ... </BodyTemplate>
    <FooterTemplate> ... </FooterTemplate>
  </Content>
</Modal>
```

The constituent elements of a modal component are immediately visible in the more expressive markup code. The markup includes a wrapper Modal element with two child subtrees: one for the toggle button and one for the actual content.

According to the Bootstrap syntax of modals, any dialog needs a trigger to be displayed. Typically, the trigger is a button element decorated with a pair of data-toggle and data-target attributes. The modal, however, can also be triggered via JavaScript. The Toggle sub-component just serves as the container for the trigger markup. The Content sub-component, instead, wraps the entire content of the dialog and is split in three segments: header, body and footer.

Figure 2 **Source Code of the Modal Component**

```
<CascadingValue Value="@Context">
  <div>
    @ChildContent
  </div>
</CascadingValue>

@functions
{
  protected override void OnInit()
  {
    Context = new ModalContext
    {
      Id = Id,
      AutoClose = AutoClose
    };
  }

  ModalContext Context { get; set; }
  [Parameter] private string Id { get; set; }
  [Parameter] private bool AutoClose { get; set; }

  [Parameter] RenderFragment ChildContent { get; set; }
}
```

In summary, based on the previous code snippet, the resulting UI is made of a primary button labeled "Open." Once clicked, the button will pop up a DIV filled with three layers: header, body and footer.

To create the nested components required for the modal dialog box, you need to deal with templated components and cascading parameters. Note that cascading parameters require you to run Blazor 0.7.0 or newer.

The Modal Component

Let's have a look at the code displayed in **Figure 2**. The markup is fairly minimal and includes a DIV element around a chunk of templated markup. The modal.cshtml file in **Figure 2** declares a template property named ChildContent that collects (obviously enough) any child content. The result of the markup is to push out a surrounding DIV element that gathers both the toggle markup and the actual content to display in the dialog.

Apparently this container component is not of great use. Nonetheless, it plays a crucial role given the required structure of the markup for Bootstrap dialog boxes. Both the Toggle and Content components share the same ID that uniquely identifies the modal dialog. By using a wrapper component, you can capture the ID value in only one place and cascade it down the tree. In this particular case, though, the ID is not even the sole parameter you want to cascade through the innermost layers of markup. A modal dialog can optionally have a Close button in the header, as well as other attributes related to the size of the dialog or the animation. All of this information can be grouped together in a custom data transfer object and cascaded through the tree.

The ModalContext class is used to collect the ID and the Boolean value for the closing button, as shown in the code here:

```
public class ModalContext
{
  public string Id { get; set; }
  public bool AutoClose { get; set; }
}
```

The CascadingValue element captures the provided expression and automatically shares it with all innermost components that explicitly bind to it. Without the cascading parameters feature, any shared value in complex and hierarchical components must be explicitly injected wherever needed. Without this feature, you would have to indicate the same ID twice, as shown in this code:

```
<Modal>
  <Toggle id="myModal" class="btn btn-primary btn-lg">
    ...
  </Toggle>
  <Content id="myModal">
    ...
  </Content>
</Modal>
```

Cascading values are helpful in situations where the same set of values must be passed along the hierarchy of a complex component made of multiple sub-components. Note that cascading values must be grouped in a single container; therefore, if you need to pass on multiple scalar values, you should first define a container object. **Figure 3** illustrates how parameters flow through the hierarchy of modal components.

Inside the Modal Component

The inner content of the Modal component is parsed recursively, and the Toggle and Content components take care of that. Here's the source of the Toggle.cshtml component:

```
<button class="@Class"
        data-toggle="modal"
        data-target="#@OutermostEnv.Id">
  @ChildContent
</button>

@functions
{
  [CascadingParameter] protected ModalContext OutermostEnv { get; set; }
  [Parameter] string Class { get; set; }
  [Parameter] RenderFragment ChildContent { get; set; }
}
```

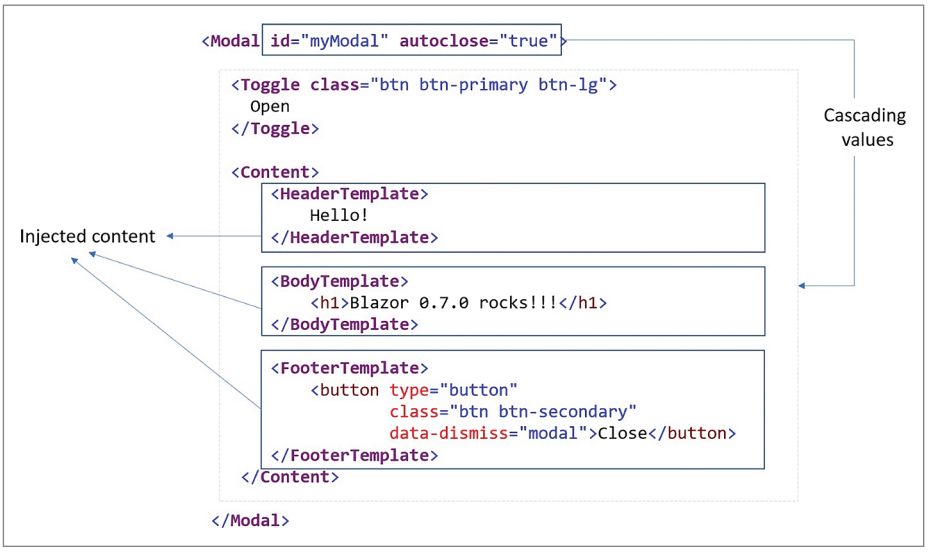


Figure 3 **Cascading Values in Hierarchical Components**

In the present implementation, the toggle element is styled through a public property named Class. The content of the button is captured through a templated property named ChildContent. Note that in Blazor, a template property named ChildContent automatically captures the entire child markup of the parent element. Also, a template property in Blazor is a property of type RenderFragment.

The interesting thing in the previous source code is the binding to the cascading values. You use the CascadingParameter attribute to decorate a component property, such as OutermostEnv. The property is then populated with cascaded values from the innermost level. As a result, OutermostEnv takes the value assigned to the instance of ModalContext freshly created in the Init method of the root component (refer back to **Figure 2**).

In the Toggle component, the Id cascaded value is used to set the value for the data-target attribute. In Bootstrap jargon, the data-target attribute of a dialog toggle button identifies the ID of the DIV to be popped up when the toggle is clicked.

The Content of the Modal Dialog

A Bootstrap dialog box is made of up to three DIV blocks laid out vertically: header, body and footer. All of them are optional, but you want to have at least one defined in order to give users some minimal feedback. A templated component is the right fit here. Here's the public interface of the Content component as it results from the Content.cshtml file:

```
@functions
{
  [CascadingParameter] ModalContext OutermostEnv { get; set; }
  [Parameter] RenderFragment HeaderTemplate { get; set; }
  [Parameter] RenderFragment BodyTemplate { get; set; }
  [Parameter] RenderFragment FooterTemplate { get; set; }
}
```

The OutermostEnv cascaded parameter will bring the data defined outside the realm of the Content component. Both the ID and AutoClose properties are used here. The Id value is used to identify the outermost container of the dialog box. The DIV signed with the ID will pop up when the modal is triggered. The AutoClose value, instead, is used to control an IF statement that decides whether or not a Dismiss button should go in the header bar.

Finally, three RenderFragment template properties define the actual content for customizable areas—header, footer and body.

As you can see in **Figure 4**, the Content component does most of the work to render the expected Bootstrap markup for modal dialog boxes. It defines the overall HTML layout and uses template properties to import the details of the markup that would make a given dialog unique—the header, footer and body markup. Thanks to Blazor templates, any actual markup can be specified as inline content in the caller page. Note that the source code of the caller page (called Cascade in the sample application) is depicted back in **Figure 3**.

Figure 4 **Markup of the Content Component**

```
<div class="modal" id="@OutermostEnv.Id">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">
          @HeaderTemplate
        </h5>
        @if (OutermostEnv.AutoClose)
        {
          <button type="button" class="close"
                  data-dismiss="modal">
            <span>&times;</span>
          </button>
        }
      </div>

      <div class="modal-body">
        @BodyTemplate
      </div>

      <div class="modal-footer">
        @FooterTemplate
      </div>

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

More on Cascading Values and Parameters

Cascading values solve the problem of effectively flowing values down the stack of subcomponents. Cascading values can be defined at various levels in a complex hierarchy and go from an ancestor component to all of its descendants. Each ancestor element can define a single cascading value, possibly a complex object that gathers together multiple scalar values.

To make use of cascaded values, descendant components declare cascading parameters. A cascading parameter is a public or protected property decorated with the CascadingParameter attribute. Cascading values can be associated with a Name property, like so:

```
<CascadingValue Value=@Context Name="ModalDialogGlobals">
  ...
</CascadingValue>
```

In this case, descendants will use the Name property to retrieve the cascaded value, as shown here:

```
[CascadingParameter(Name = "ModalDialogGlobals")]
ModalContext OutermostEnv { get; set; }
```

When no name is specified, cascading values are bound to cascading parameters by type.

Wrapping Up

Cascading values are specifically designed for hierarchical components, but at the same time hierarchical (and templated) components are realistically the most common type of Blazor components that developers are expected to write. This article demonstrated cascading parameters and templated and hierarchical components, but also showed how powerful it could be to use Razor components to express specific pieces of markup through a higher-level syntax. In particular, I worked out a custom markup syntax to render a Bootstrap modal dialog box. Note that you can achieve the same in plain ASP.NET Core using tag helpers or HTML helpers in classic ASP.NET MVC.

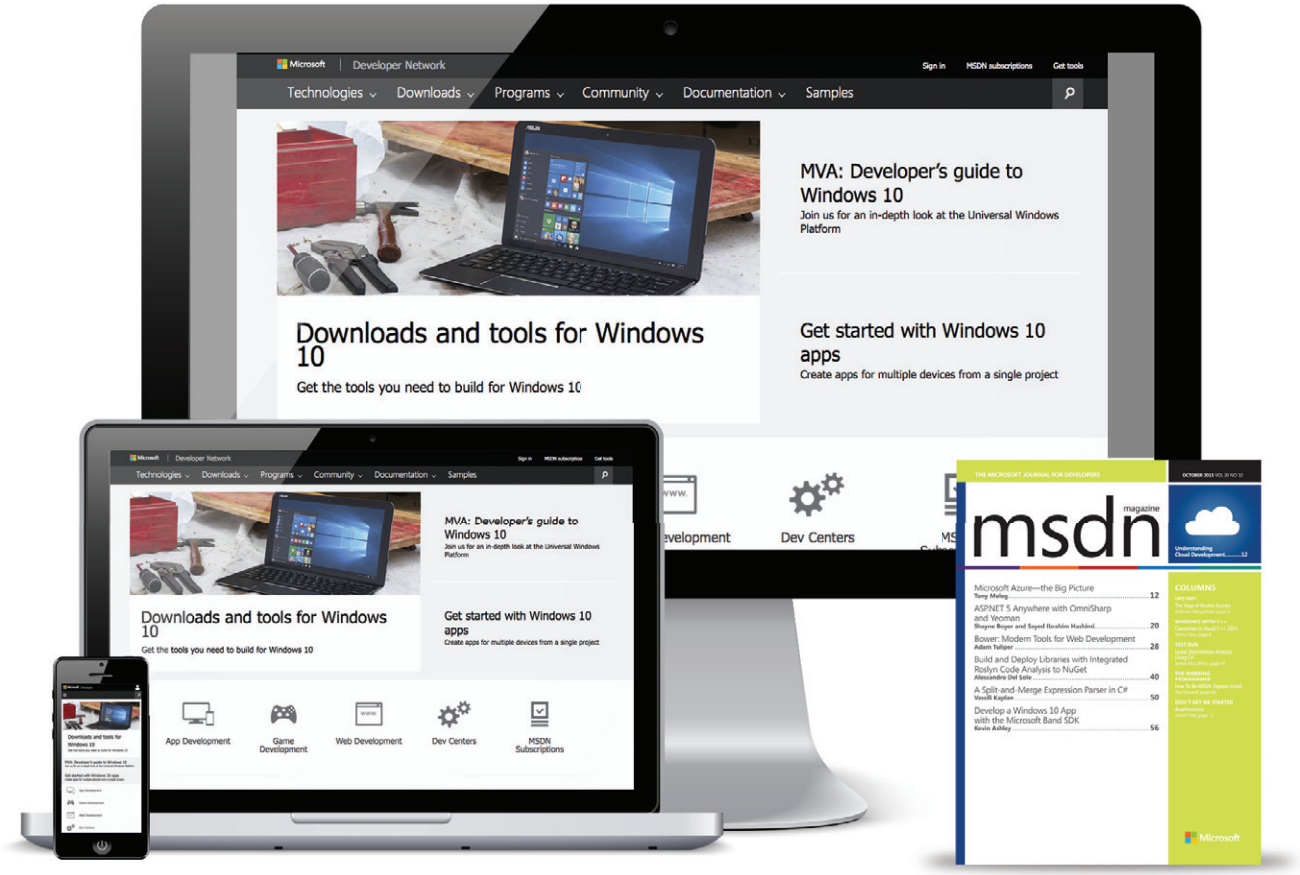
The source code for the article is available at bit.ly/2FdGZat. ■

Dino Esposito *has authored more than 20 books and 1,000-plus articles in his 25-year career. Author of "The Sabbatical Break," a theatrical-style show, Esposito is busy writing software for a greener world as the digital strategist at BaxEnergy. Follow him on Twitter: @despos.*

Thanks to the following Microsoft technical expert for reviewing this article: *Daniel Roth*

msdn magazine

Where you need us most.



Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

LIVE! 360
TECH EVENTS WITH PERSPECTIVE

MSDN.microsoft.com

JAMES MCCAFFREY

Neural Regression Using PyTorch

The goal of a regression problem is to predict a single numeric value. For example, you might want to predict the price of a house based on its square footage, age, ZIP code and so on. In this article I show how to create a neural regression model using the PyTorch code library. The best way to understand where this article is headed is to take a look at the demo program in **Figure 1**.

The demo program creates a prediction model based on the Boston Housing dataset, where the goal is to predict the median house price in one of 506 towns close to Boston. The data comes from the early 1970s. Each data item has 13 predictor variables, such as crime index of the town, average number of rooms per house in the town and so on. There's only one output value because the goal is to predict a single numeric value.

The demo loads 404 training items and 102 test items into memory, and then creates a 13-(10-10)-1 neural network. The neural network has two hidden processing layers, each of which has 10 nodes. The number of input and output nodes is determined by the data, but the number of hidden layers and the number of nodes in each are free parameters that must be determined by trial and error.

The demo trains the neural network, meaning the values of the weights and biases that define the behavior of the neural network are computed using the training data, which has known correct input and output values. After training, the demo computes the accuracy of the model on the test data (75.49 percent, 77 out of 102 correct). The test accuracy is a rough measure of how well you'd expect the model to do on new, previously unseen data.

The demo concludes by making a prediction for the first test town. The 13 raw input values are (0.09266, 34.0, . . 8.67). When the neural regression model was trained, normalized data (scaled so all values are between 0.0 and 1.0) was used, so when making a prediction the demo had to use normalized data, which is (0.00097, 0.34, . . 0.191501). The model predicts that the median house price is $24,870.07, quite close to the actual median price of $26,400.

This article assumes you have intermediate or better programming skill with a C-family language and a basic familiarity with machine learning. The complete demo code is presented in this article. The source code and the two data files used by the demo are also available in the accompanying download. All normal error checking has been removed to keep the main ideas as clear as possible.

Code download available at msdn.com/magazine/0319magcode.

Installing PyTorch

Installing PyTorch involves two main steps. First you install Python and several required auxiliary packages, such as NumPy and SciPy, then you install PyTorch as an add-on Python package.

> The goal of a regression problem is to predict a single numeric value.

Although it's possible to install Python and the packages required to run PyTorch separately, it's much better to install a Python distribution. For my demo, I installed the Anaconda3 5.2.0 distribution (which contains Python 3.6.5) and PyTorch 1.0.0. If you're

```
C:\WINDOWS\system32\cmd.exe
C:\PyTorch\Boston>python boston_dnn.py

Boston regression using PyTorch 1.0

Loading Boston data into memory
Creating 13-(10-10)-1 DNN regression model

Starting training
batch =      0  batch loss =  4.8365  accuracy = 2.48%
batch =   4000  batch loss =  0.1528  accuracy = 71.04%
batch =   8000  batch loss =  0.0512  accuracy = 73.02%
batch =  12000  batch loss =  0.1166  accuracy = 73.51%
batch =  16000  batch loss =  0.0902  accuracy = 75.50%
batch =  20000  batch loss =  0.1892  accuracy = 75.99%
batch =  24000  batch loss =  0.0405  accuracy = 79.21%
batch =  28000  batch loss =  0.0128  accuracy = 78.96%
batch =  32000  batch loss =  0.0520  accuracy = 81.93%
batch =  36000  batch loss =  0.0583  accuracy = 81.68%
Training complete

Accuracy on test data = 75.49%
For a town with raw input values:

   0.092660   34.000000    6.090000    0.000000    0.433000
   6.495000   18.400000    5.491700    7.000000  329.000000
  16.100000  383.609985    8.670000

Predicted median house price = $24870.07

C:\PyTorch\Boston>
```

Figure 1 **Neural Regression Using a PyTorch Demo Run**

#VSLIVE

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

August 12-16, 2019
Microsoft Campus in Redmond, WA

INTENSE DEVELOPER TRAINING CONFERENCE

In-depth Technical Content On:

- AI, Data and Machine Learning
- Cloud, Containers and Microservices
- Delivery and Deployment
- Developing New Experiences
- DevOps in the Spotlight
- Full Stack Web Development
- .NET Core and More

New This Year!

On-Demand Session Recordings Now Available

Get on-demand access for one full year to all keynotes and sessions from Visual Studio Live! Microsoft HQ, including everything Tuesday – Thursday at the conference.

Save $400
When You Register By June 21!

vslive.com/microsofthq

SUPPORTED BY
Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY
CONVERGE 360 an 1105 MEDIA company

new to Python, be aware that installing and managing add-on package dependencies is non-trivial.

After installing Python via the Anaconda distribution, the PyTorch package can be installed using the pip utility function with a .whl ("wheel") file. PyTorch comes in a CPU-only version and in a GPU version. I used the CPU-only version.

Understanding the Data

The Boston Housing dataset comes from a research paper written in 1978 that studied air pollution. You can find different versions of the dataset in many locations on the Internet. The first data item is:

```
0.00632, 18.00, 2.310, 0, 0.5380, 6.5750, 65.20
4.0900, 1, 296.0, 15.30, 396.90, 4.98, 24.00
```

Each data item has 14 values and represents one of 506 towns near Boston. The first 13 numbers are the values of predictor variables and the last value is the median house price in the town (divided by 1,000). Briefly, the 13 predictor variables are: crime rate in the town, large lot percentage, percentage zoned for industry, adjacency to Charles River, pollution, average number rooms per house, house age information, distance to Boston, accessibility to highways, tax rate, pupil-teacher ratio, proportion of Black residents, and percentage of low-status residents.

Because there are 14 variables, it's not possible to visualize the dataset, but you can get a rough idea of the data from the graph in **Figure 2**. The graph shows median house price as a function of the percentage of town zoned for industry for the 102 items in the test dataset.

When working with neural networks, you must encode non-numeric data and you should normalize numeric data so that large values, such as a pupil-teacher ratio of 20.0, don't overwhelm small values, such as a pollution reading of 0.538. The Charles River variable is a categorical value stored either as 0 (town is not adjacent) or 1 (adjacent). Those values were re-encoded as -1 and +1. The other 12 predictor variables are numeric. For each variable, I computed the min value and the max value, and then for every value x, normalized it as (x - min) / (max - min). After min-max normalization, all values will be between 0.0 and 1.0.

The median house values in the raw data were already normalized by dividing by 1,000, so the values ranged from 5.0 to 50.0, with most at about 25.0. I applied an additional normalization by dividing the prices by 10 so that all median house prices were between 0.5 and 5.0, with most being around 2.5.

The Demo Program

The complete demo program, with a few minor edits to save space, is presented in **Figure 3**. I indent two spaces rather than the usual four spaces to save space. And note that Python uses the '\' character for line continuation. I used Notepad to edit my program, but many of my colleagues prefer Visual Studio or VS Code, both of which have excellent support for Python.

The demo imports the entire PyTorch package and assigns it an alias of T. An alternative is to import just the modules and functions needed.

The demo defines a helper function called accuracy. When using a regression model, there's no inherent definition of the accuracy of a prediction. You must define how close a predicted value must be to a target value in order to be counted as a correct prediction.

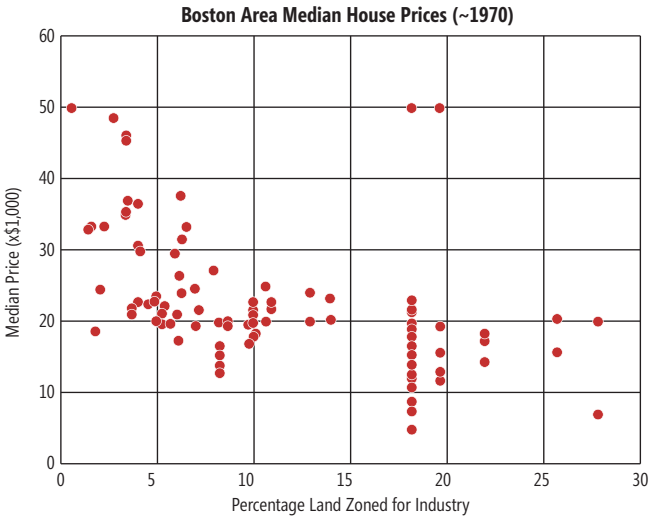


Figure 2 **Partial Boston Area House Dataset**

The demo program counts a predicted median house price as correct if it's within 15 percent of the true value.

All the control logic for the demo program is contained in a single main function. Program execution begins by setting the global NumPy and PyTorch random seeds so results will be reproducible.

Loading Data into Memory

The demo loads data in memory using the NumPy loadtxt function:

```
train_x = np.loadtxt(train_file, delimiter="\t",
  usecols=range(0,13), dtype=np.float32)
train_y = np.loadtxt(train_file, delimiter="\t",
  usecols=[13], dtype=np.float32)
test_x = np.loadtxt(test_file, delimiter="\t",
  usecols=range(0,13), dtype=np.float32)
test_y = np.loadtxt(test_file, delimiter="\t",
  usecols=[13], dtype=np.float32)
```

The code assumes that the data is located in a subdirectory named Data. The demo data was preprocessed by splitting it into training and test sets. Data wrangling isn't conceptually difficult, but it's almost always quite time-consuming and annoying. Many of my colleagues like to use the pandas (Python data analysis) package to manipulate data.

You must define how close a predicted value must be to a target value in order to be counted as a correct prediction.

Defining the Neural Network

The demo defines the 13-(10-10)-1 neural network in a program-defined class named Net that inherits from the nn.Module module. You can think of the Python __init__ function as a class constructor. Notice that you don't explicitly define an input layer because input values are fed directly to the first hidden layer.

TECHMENTOR

IN-DEPTH TRAINING FOR IT PROS

Training Conference for IT Pros at Microsoft HQ!

The FUTURE of TECH is HERE

AGENDA COMING SOON!

MICROSOFT HEADQUARTERS
REDMOND, WA
AUGUST 5-9, 2019

In-depth Technical Tracks on:

- Client and EndPoint Management
- Cloud
- Infrastructure
- PowerShell and DevOps
- Security
- Soft Skills for IT Pros

SAVE $400
WHEN YOU REGISTER BY JUNE 14
Use Promo Code MSDN

EVENT PARTNER
Microsoft

SUPPORTED BY
Redmond
Redmond Channel Partner
Virtualization & Cloud Review

PRODUCED BY
Converge 360
An 1105 Media company

TechMentorEvents.com/MicrosoftHQ

The network has (13 * 10) + (10 * 10) + (10 * 1) = 240 weights. Each weight is initialized to a small random value using the Xavier Uniform algorithm. The network has 10 + 10 + 1 = 21 biases. Each bias value is initialized to zero.

The Net class forward function defines how the layers compute output. The demo uses tanh (hyperbolic tangent) activation on the two hidden layers, and no activation on the output layer:

```
def forward(self, x):
  z = T.tanh(self.hid1(x))
  z = T.tanh(self.hid2(z))
  z = self.oupt(z)
  return z
```

For hidden layer activation, the main alternative is rectified linear unit (ReLU) activation, but there are many other functions.

Because PyTorch works at a relatively low level of abstraction, there are many alternative design patterns you can use. For example, instead of defining a class Net with the __init__ and forward functions, and then instantiating with net = Net(), you can use the Sequential function, like so:

```
net = T.nn.Sequential(
  T.nn.Linear(13,10),
  T.nn.Tanh(),
  T.nn.Linear(10,10),
  T.nn.Tanh(),
  T.nn.Linear(10,1))
```

Figure 3 **The Boston Housing Demo Program**

```
# boston_dnn.py
# Boston Area House Price dataset regression
# Anaconda3 5.2.0 (Python 3.6.5), PyTorch 1.0.0

import numpy as np
import torch as T  # non-standard alias

# -----------------------------------------------------------

def accuracy(model, data_x, data_y, pct_close):
  n_items = len(data_y)
  X = T.Tensor(data_x)  # 2-d Tensor
  Y = T.Tensor(data_y)  # actual as 1-d Tensor

  oupt = model(X)       # all predicted as 2-d Tensor
  pred = oupt.view(n_items)  # all predicted as 1-d

  n_correct = T.sum((T.abs(pred - Y) < T.abs(pct_close * Y)))
  result = (n_correct.item() * 100.0 / n_items)  # scalar
  return result

# -----------------------------------------------------------

class Net(T.nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.hid1 = T.nn.Linear(13, 10)  # 13-(10-10)-1
    self.hid2 = T.nn.Linear(10, 10)
    self.oupt = T.nn.Linear(10, 1)

    T.nn.init.xavier_uniform_(self.hid1.weight)  # glorot
    T.nn.init.zeros_(self.hid1.bias)
    T.nn.init.xavier_uniform_(self.hid2.weight)
    T.nn.init.zeros_(self.hid2.bias)
    T.nn.init.xavier_uniform_(self.oupt.weight)
    T.nn.init.zeros_(self.oupt.bias)

  def forward(self, x):
    z = T.tanh(self.hid1(x))
    z = T.tanh(self.hid2(z))
    z = self.oupt(z)  # no activation, aka Identity()
    return z

# -----------------------------------------------------------

def main():
  # 0. Get started
  print("\nBoston regression using PyTorch 1.0 \n")
  T.manual_seed(1);  np.random.seed(1)

  # 1. Load data
  print("Loading Boston data into memory ")
  train_file = ".\\Data\\boston_train.txt"
  test_file = ".\\Data\\boston_test.txt"

  train_x = np.loadtxt(train_file, delimiter="\t",
    usecols=range(0,13), dtype=np.float32)
  train_y = np.loadtxt(train_file, delimiter="\t",
    usecols=[13], dtype=np.float32)
  test_x = np.loadtxt(test_file, delimiter="\t",
    usecols=range(0,13), dtype=np.float32)
  test_y = np.loadtxt(test_file, delimiter="\t",
    usecols=[13], dtype=np.float32)

  # 2. Create model
  print("Creating 13-(10-10)-1 DNN regression model \n")
  net = Net()  # all work done above

  # 3. Train model
  net = net.train()  # set training mode
  bat_size = 10
  loss_func = T.nn.MSELoss()  # mean squared error
  optimizer = T.optim.SGD(net.parameters(), lr=0.01)
  n_items = len(train_x)
  batches_per_epoch = n_items // bat_size
  max_batches = 1000 * batches_per_epoch

  print("Starting training")
  for b in range(max_batches):
    curr_bat = np.random.choice(n_items, bat_size,
      replace=False)
    X = T.Tensor(train_x[curr_bat])
    Y = T.Tensor(train_y[curr_bat]).view(bat_size,1)
    optimizer.zero_grad()
    oupt = net(X)

    loss_obj = loss_func(oupt, Y)
    loss_obj.backward()
    optimizer.step()

    if b % (max_batches // 10) == 0:
      print("batch = %6d" % b, end="")
      print("  batch loss = %7.4f" % loss_obj.item(), end="")
      net = net.eval()
      acc = accuracy(net, train_x, train_y, 0.15)
      net = net.train()
      print("  accuracy = %0.2f%%" % acc)
  print("Training complete \n")

  # 4. Evaluate model
  net = net.eval()  # set eval mode
  acc = accuracy(net, test_x, test_y, 0.15)
  print("Accuracy on test data = %0.2f%%" % acc)

  # 5. Save model - TODO

  # 6. Use model
  raw_inpt = np.array([[0.09266, 34, 6.09, 0, 0.433, 6.495, 18.4,
    5.4917, 7, 329, 16.1, 383.61, 8.67]], dtype=np.float32)

  norm_inpt = np.array([[0.000970, 0.340000, 0.198148, -1,
    0.098765, 0.562177, 0.159629, 0.396666, 0.260870, 0.270992,
    0.372340, 0.966488, 0.191501]], dtype=np.float32)

  X = T.Tensor(norm_inpt)
  y = net(X)
  print("For a town with raw input values: ")
  for (idx,val) in enumerate(raw_inpt[0]):
    if idx % 5 == 0: print("")
    print("%11.6f " % val, end="")
  print("\n\nPredicted median house price = $%0.2f" %
    (y.item()*10000))

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

The Sequential approach is much simpler, but notice you don't have direct control over the weight and bias initialization algorithms. The tremendous flexibility you get when using PyTorch is an advantage once you become familiar with the library.

Training the Model

Training the model begins with these seven statements:

```
net = net.train()  # Set training mode
bat_size = 10
loss_func = T.nn.MSELoss()  # Mean squared error
optimizer = T.optim.SGD(net.parameters(), lr=0.01)
n_items = len(train_x)
batches_per_epoch = n_items // bat_size
max_batches = 1000 * batches_per_epoch
```

PyTorch has two modes: train and eval. The default mode is train, but in my opinion it's a good practice to explicitly set the mode. The batch (often called mini-batch) size is a hyperparameter. For a regression problem, mean squared error is the most common loss function. The stochastic gradient descent (SGD) algorithm is the most rudimentary technique and in many situations the Adam algorithm gives better results.

The demo program uses a simple approach for batching training items. For the demo, there are about 400 training items, so if the batch size is 10, on average visiting each training item once (this is usually called an epoch in machine learning terminology) will require 400 / 10 = 40 batches. Therefore, to train the equivalent of 1,000 epochs, the demo program needs 1000 * 40 = 40,000 batches.

The core training statements are:

```
for b in range(max_batches):
  curr_bat = np.random.choice(n_items, bat_size,
    replace=False)
  X = T.Tensor(train_x[curr_bat])
  Y = T.Tensor(train_y[curr_bat]).view(bat_size,1)
  optimizer.zero_grad()
  oupt = net(X)
  loss_obj = loss_func(oupt, Y)
  loss_obj.backward()  # Compute gradients
  optimizer.step()     # Update weights and biases
```

The choice function selects 10 random indices from the 404 available training items. The items are converted from NumPy arrays to PyTorch tensors. You can think of a tensor as a multi-dimensional array that can be efficiently processed by a GPU (even though the demo doesn't take advantage of a GPU). The oddly named view function reshapes the one-dimensional target values into a two-dimensional tensor. Converting NumPy arrays to PyTorch tensors, and dealing with array and tensor shapes is a major challenge when working with PyTorch.

Once every 4,000 batches the demo program displays the value of the mean squared error loss for the current batch of 10 training items, and the prediction accuracy of the model, using the current weights and biases on the entire 404-item training dataset:

```
if b % (max_batches // 10) == 0:
  print("batch = %6d" % b, end="")
  print("  batch loss = %7.4f" % loss_obj.item(), end="")
  net = net.eval()
  acc = accuracy(net, train_x, train_y, 0.15)
  net = net.train()
  print("  accuracy = %0.2f%%" % acc)
```

The "//" operator is integer division in Python. Before calling the program-defined accuracy function, the demo sets the network into eval mode. Technically, this isn't necessary because train and eval modes only give different results if the network uses dropout or layer batch normalization.

Evaluating and Using the Trained Model

After training completes, the demo program evaluates the prediction accuracy of the model on the test datasets:

```
net = net.eval()  # set eval mode
acc = accuracy(net, test_x, test_y, 0.15)
print("Accuracy on test data = %0.2f%%" % acc)
```

The eval function returns a reference to the model on which it's applied; it could have been called without the assignment statement.

In most situations, after training a model you want to save the model for later use. Saving a trained PyTorch model is a bit outside the scope of this article, but you can find several examples in the PyTorch documentation.

> PyTorch has two modes: train and eval. The default mode is train, but in my opinion it's a good practice to explicitly set the mode.

The whole point of training a regression model is to use it to make a prediction. The demo program makes a prediction using the first data item from the 102 test items:

```
raw_inpt = np.array([[0.09266, 34, 6.09, 0, 0.433, 6.495, 18.4,
  5.4917, 7, 329, 16.1, 383.61, 8.67]], dtype=np.float32)

norm_inpt = np.array([[0.000970, 0.340000, 0.198148, -1,
  0.098765, 0.562177, 0.159629, 0.396666, 0.260870, 0.270992,
  0.372340, 0.966488, 0.191501]], dtype=np.float32)
```

When you have new data, you must remember to normalize the predictor values in the same way that the training data was normalized. For min-max normalization, that means you need to save the min and max value for every variable that was normalized.

The demo concludes by making and displaying the prediction:

```
...
  X = T.Tensor(norm_inpt)
  y = net(X)
  print("Predicted = $%0.2f" % (y.item()*10000))
if __name__=="__main__":
  main()
```

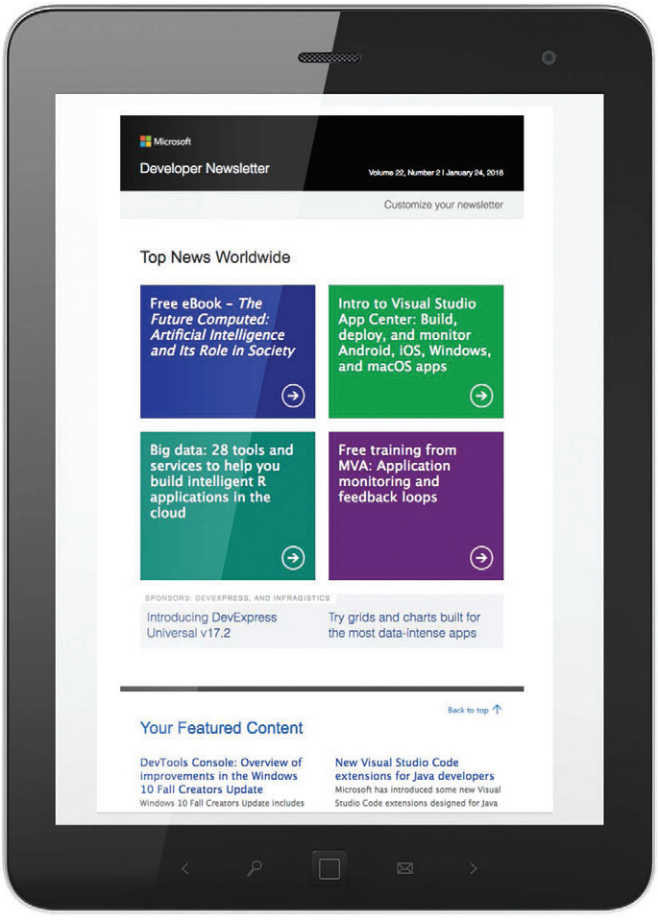
The predicted value is returned as a tensor with a single value. The item function is used to access the value so it can be displayed.

Wrapping Up

The PyTorch library is somewhat less mature than alternatives TensorFlow, Keras and CNTK, especially with regard to example code. But among my colleagues, the use of PyTorch is growing very quickly. I expect this trend to continue and high-quality examples will become increasingly available to you. ■

Dr. James McCaffrey *works for Microsoft Research in Redmond, Wash. He has worked on several key 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*



Get news from MSDN in your inbox!

Sign up to receive the
MICROSOFT DEVELOPER NEWSLETTER,
which delivers the latest resources, SDKs, downloads, partner offers, security news, and updates on national and local developer events.

msdn
magazine

msdn.microsoft.com/flashnewsletter

Do As I Say, Not As I Do

Being a parent really opens your eyes to hypocrisy. Any time your kids catch you practicing differently from what you preach, you get it right back in the teeth: "But, Daddy, you told me that downloading unauthorized music was stealing, and we shouldn't do it." We mumble something about, yeah, well, maybe not exactly in this case, but that doesn't get us far. We're left with the inevitable conflict that occurs when most humans (including you and me, dear reader) say one thing but do another. Don't believe me? Read on.

There's no greater example of hypocrisy than people's behavior regarding digital privacy. At a recent conference, I joined several other speakers on a panel discussing that topic. The other speakers solemnly intoned that privacy was important. The audience agreed, yes, important, very important. Very, very important.

I couldn't resist pouring oil on this troubled fire. "OK," I asked the crowd. "Suppose your government required you to wear a location tracker at all times, like a convicted felon, so they could tell where you are and where you've been. Sounds awful, right?" The audience nodded, it sure did sound awful. "And suppose the government could share your location with anyone they wanted, without telling you. Sell it to the highest bidder. Really awful, right?" Yes, really, really awful. "And now, suppose they made you pay for it? Fifty Euros a month, they charged you actual money? You'd storm the parliament building and throw the bums out, right?" Yells now from the crowd, I wondered if I'd agitated them too much.

"And suppose to keep you pacified, once in a while the tracker would show you a cat video." Widespread groans; they saw my point coming, but way too late. "OK, then, wise guys, who here does *not* have a smartphone in your pocket right now?" No hands. Not one. "And who has bothered to turn off location sharing?" Two hands, maybe three, of 700 attendees. "So you say, vehemently, that privacy is important. But when you have the choice of privacy versus a little less functionality, like taking five seconds longer to find the nearest espresso stand, you fall all over yourselves handing everything to Big Brother? Don't any of you ever tell me that you give a flying fish about privacy while you have your phone turned on."

Figure 1 **Janus the Two-Faced God**

I know you logical geeks are squirming here. I am myself. In theory, we don't want anyone watching us, but in practice, we don't care until something bites us on the butt, and then it's too late. When users make choices, immediate convenience always, *always*, displaces abstract ideals. As security expert Jesper Johansson once said to me, "Given the choice between security and dancing pigs, users will take the dancing pigs every time."

Many writers would call here for a consciousness-raising educational effort, but I won't. This denial, believing what we want to believe (that our phone is magically taking care of everything and won't hurt us) simply because we find that belief convenient, is a fundamental part of the human organism. As I wrote in my very first DGMS ("The Human Touch" msdn.com/magazine/ee309884) back in February 2010: "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."

Human users are two-faced. They say one thing and do the exact opposite. My daughter Annabelle, now 18, is starting to realize that—perhaps the beginning of her graduation from teen to human? Lucy, 16, still expects hypocrisy to vanish when she recognizes and exposes it, and get furious when it doesn't. She'll learn better soon, while I mourn that she has to.

The ancient Romans dedicated an entire god, Janus, to this dichotomy (en.wikipedia.org/wiki/Janus). His statue in **Figure 1** is more than 2,000 years old. This condition—I won't call it a problem, it's simply a part of life, basic as gravity—is not a new one. We need to take care of our users anyway, even if—*especially* if—being human, they won't take care of themselves. ■

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 have taped down two of his daughter's fingers so she would learn how to count in octal. You can contact him at rollthunder.com.*

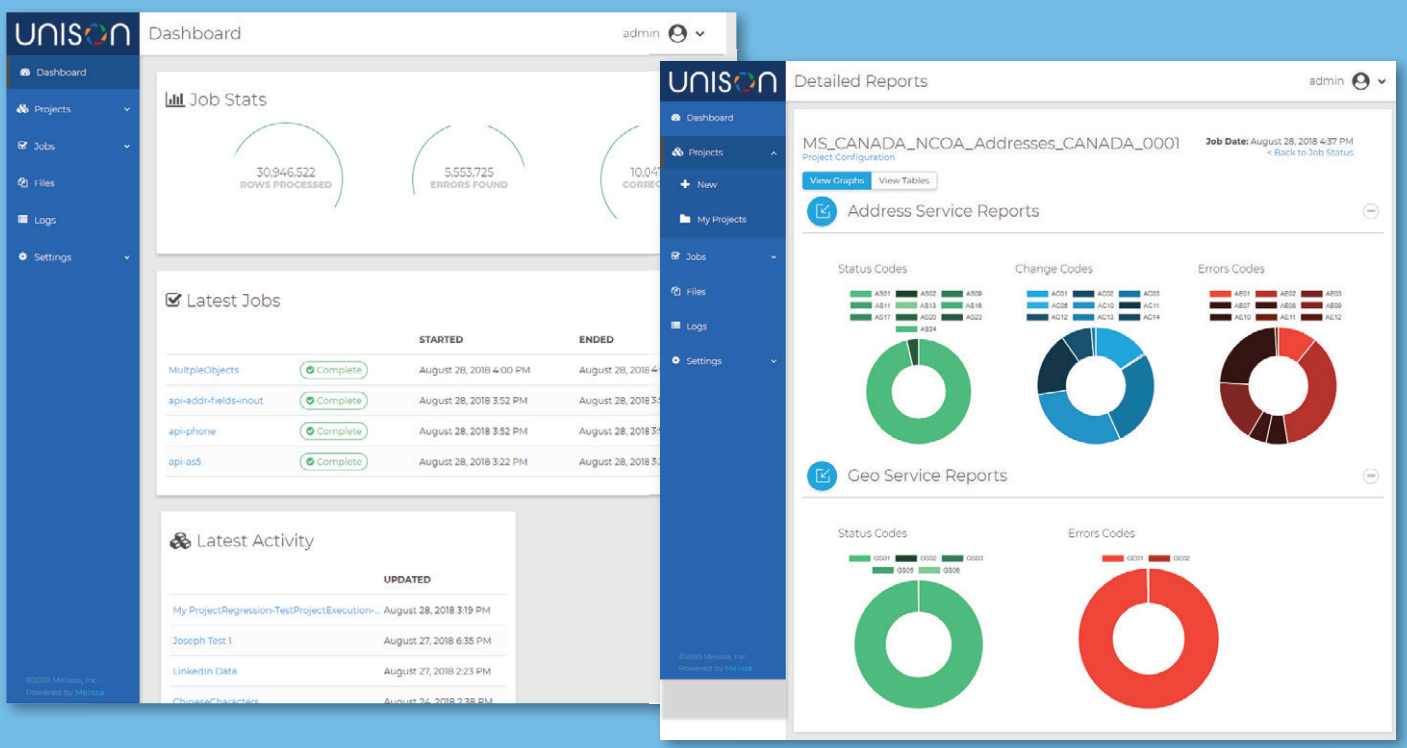
Multiplatform Data Quality Management

Unison – Speedy, Secure, Scalable

Melissa's Unison is a data steward's best friend. Unison is a unique multiplatform solution that establishes and maintains contact data quality at lightning speeds – processing 50 million addresses per hour – while meeting the most stringent security requirements. With Unison, you can design, administer and automate data quality routines that cleanse, validate and enrich even your most sensitive customer information – name, address, phone, email address – as data never leaves your organization.

- **Lightning fast processing (50M records/hour)**
- **Streamline data prep workflows**
- **Reduce analytics busy work**
- **Services: Address, Name, Email, Phone Verification & Geocoding**

UNISON by melissa



Request a Demo

Melissa.com/msdn-unison

1-800-MELISSA

melissa®