THE MICROSOFT JOURNAL FOR DEVELOPERS

FEBRUARY 2019 VOL 34 NO 2

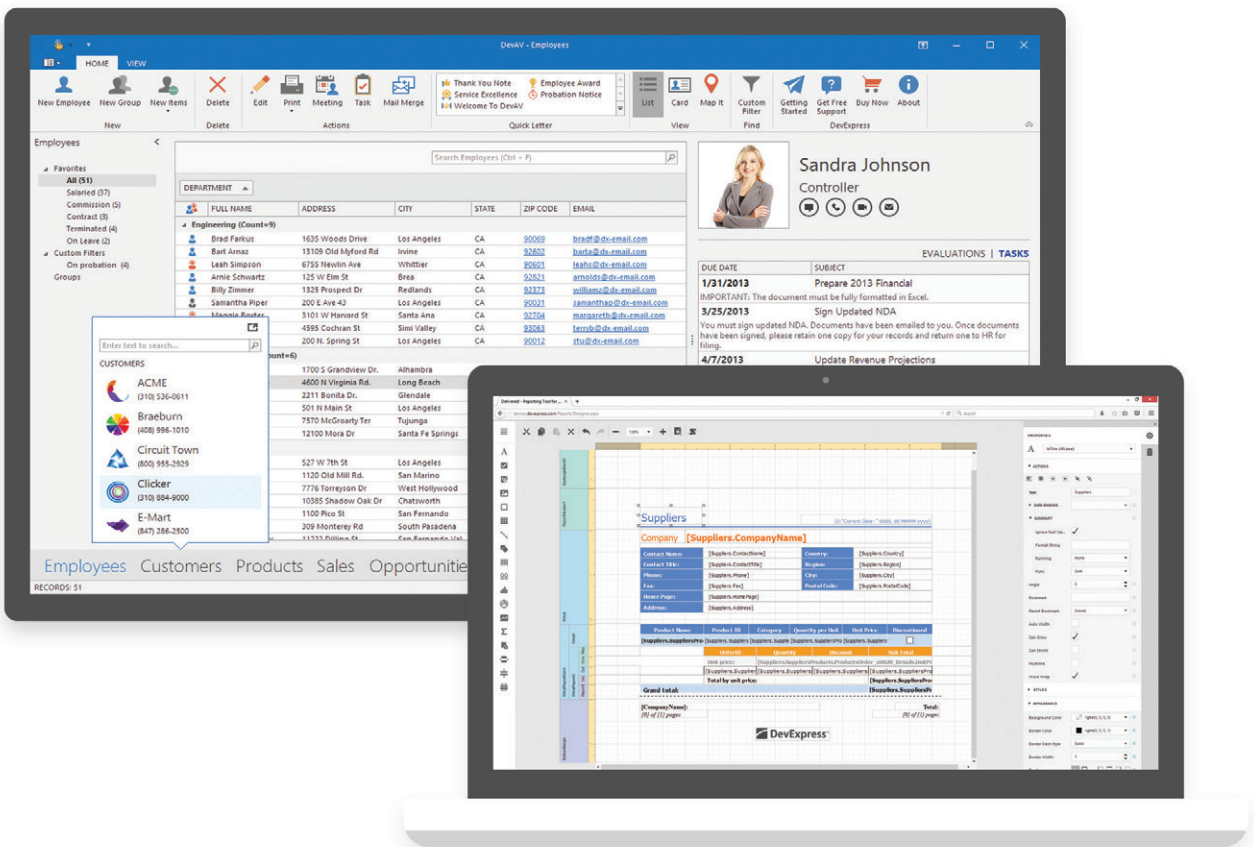
msdn magazine

C#

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

FEBRUARY 2019 VOL 34 NO 2

msdn magazine

C#

Multithreaded Programming.....20

COLUMNS

Microsoft

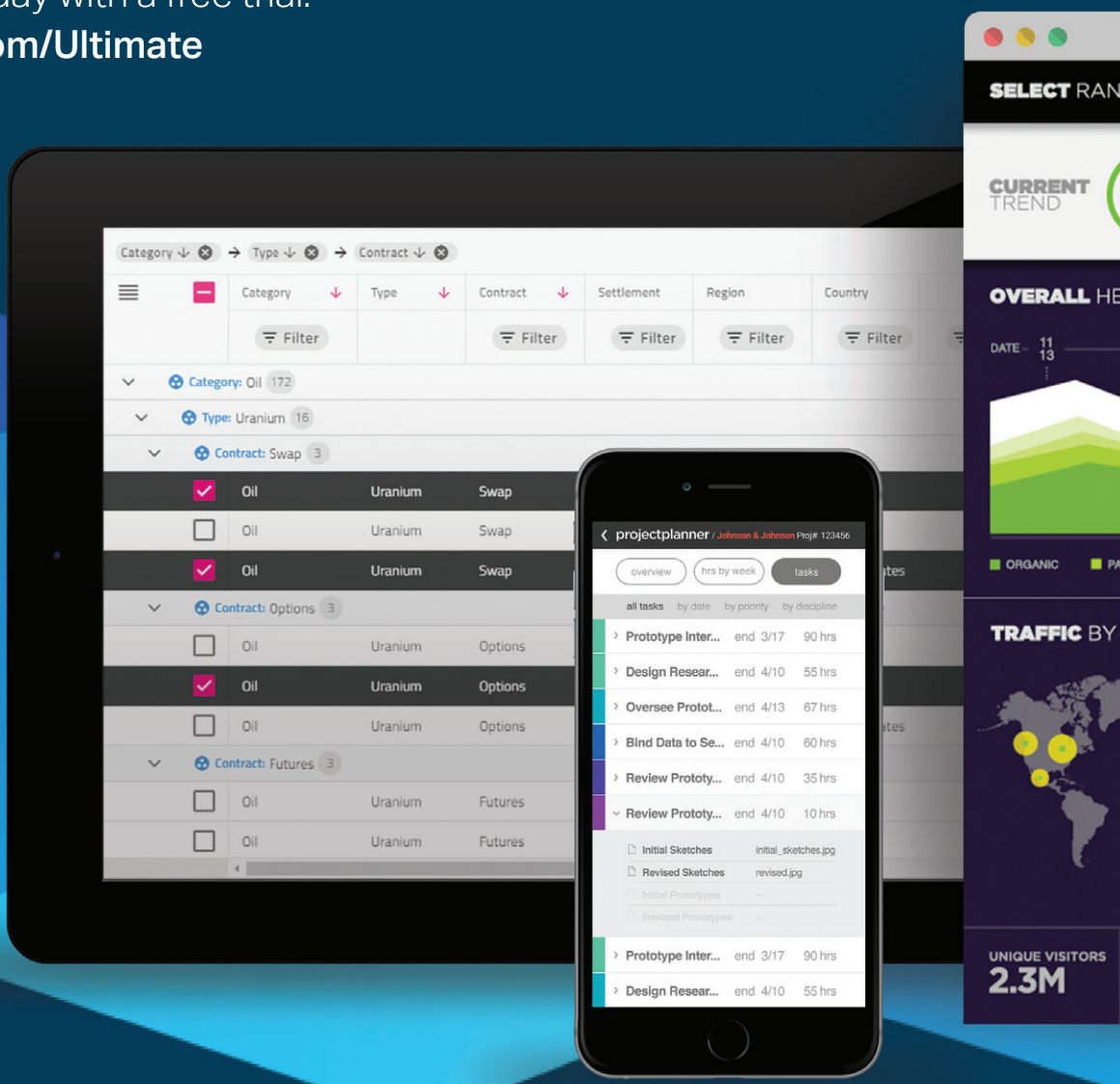
INFRAGISTICS

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

FEBRUARY 2019 VOLUME 34 NUMBER 2

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.

LEADTOOLS®

MULTI-PLATFORM BARCODE SDK

SDKs for WEB

SDKs for MOBILE

SDKs for DESKTOP

SDKs for SERVER

Use LEADTOOLS multi-platform Barcode SDK to develop applications that require reading and writing **1D** and **2D barcodes**. With comprehensive support of over 100 different barcode types such as UPC, EAN, Code 39, Code 128, ITF (2 of 5), CodaBar, GS1 DataBar, QR, Data Matrix, PDF417, and Aztec, LEADTOOLS is the right choice.

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

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

MICHAEL DESMOND

6 More Weeks of What?

A year ago in this space, I drew inspiration from the odd cultural phenomenon that is Groundhog Day to offer predictions for the year ahead. With magical, weather-prognosticating rodents once again seizing control of our climate, I thought it a good time to return with a few fresh predictions. And once again, I reached out to the team of contributing editors at *MSDN Magazine* for their vision of the future.

Last year, machine learning (ML) and artificial intelligence (AI) were top of mind. Both Test Run columnist James McCaffrey and Artificially Intelligent columnist Frank La Vigne predicted that enterprise developers' roles would expand to include coding for AI and ML, much the way they did for mobile app development and DevOps. La Vigne says 2019 will be the year that AI hits fast-forward. "Last year, I saw a few technologies go from published paper to deployed service within six months. That kind of unprecedented pace is both exciting and terrifying."

Keeping pace with fast-changing technologies is old hat to developers, but they'll have help from automated ML solutions that streamline mundane tasks, La Vigne says. Longer term, he says evolving technologies may even "automate away much of the work that keeps data scientists gainfully employed."

McCaffrey in his role at Microsoft Research has a front-row seat to ML developments. He expects rapid uptake of the PyTorch neural network library through 2019. And he predicts that enterprises will apply neural network-based prediction code to increasingly complex challenges, such as to sentiment analysis and to hybrid systems like deep neural networks combined with reinforcement learning systems.

Beyond AI, McCaffrey says, "I'm cautiously hopeful that in 2019, big progress will be made in the field of homomorphic encryption—a form of encryption that allows computation directly on encrypted data." While such systems work in theory today, they're still too slow for practical use.

Thomas Hansen is a regular contributor to the magazine, and a guy completely unafraid to articulate his vision of programming. He posits that new tools will emerge to create a fusion between the server and client, and offers as evidence the GraphQL query language for APIs and his own, home-brewed Lizzie scripting language.

Along those lines, Ted Neward (of The Working Programmer fame) says innovation around WebAssembly (WASM) will bridge gaps between disparate server and client code. Accessing code written in another language for a single-page application means implementing an HTTP-based API channel for the browser to call that other-language code on a server. It gets involved. Now, says Neward, "I can cross-compile my code to WASM, embed it into my webapp, and run everything out of the browser." Neward says 2019 will see the rise of language UI libraries and frameworks using WASM to let developers write browser-based UIs in a higher-level language, while getting away from traditional DOM-based browser controls.

The Darkest Timeline

Last year's predictions included concerns around attacks like Sceptre and Meltdown. Now concerns shift to impacts from AI. La Vigne says autonomous vehicles will eventually threaten to push millions of truck drivers out of work. In the nearer term, he says keep an eye out for DeepFakes, which take Fake News to the next level by using neural networks to seamlessly map the image of one person onto another. La Vigne thinks 2019 will produce "the first political scandal brought on by a DeepFake." That'll be fun.

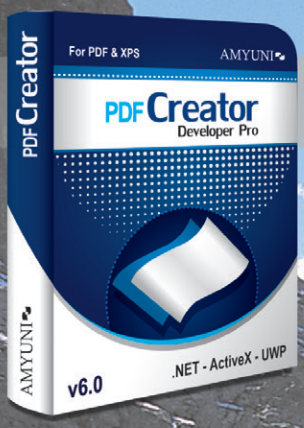
The most troubling prediction may come from David Platt, who thinks his Don't Get Me Started columns could be used to train an AI-powered rant generator. His dark vision for 2019? "A rogue TV commercial triggers home speaker voice AIs to start shouting at each other," Platt says. "Riot spreads until the city cuts electric power."

Now *that's* the darkest timeline.

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

Switch to Amyuni® PDF

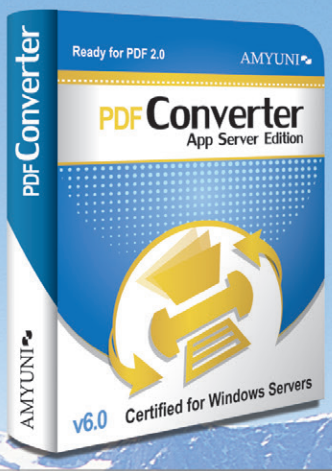
AMYUNI



Create and Process PDFs in .NET, COM/ActiveX and UWP

NEW v6.0

- Edit, process and print PDF 2.0 documents
- Create, fill-out and annotate PDF forms with Javascript support
- Fast and lightweight 64-bit components
- Universal Apps DLLs enable publishing C#, C++, CX or JS apps to Windows Store
- Plus a number of exciting features in v6.0



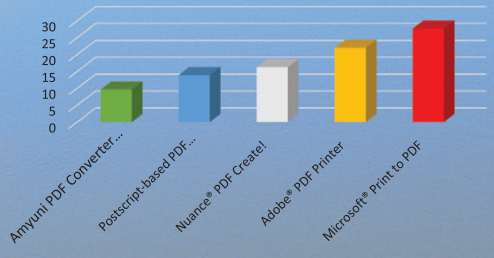
High Performance PDF Printer for Desktops and Servers

Print to PDF in a fraction of the time needed with other tools. WHQL certified for all Windows platforms. Version 6.0 updated for Server 2016.

Complete Suite of PDF, XPS and DOCX Components

- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment
- Includes our Microsoft® WHQL certified PDF and DOCX printer driver
- Export PDF documents into other formats such as images, HTML5 or editable DOCX
- Import/Export XPS files using our native libraries

Benchmark Testing - Amyuni vs Others
Seconds required to convert a document to PDF



Other Developer Components from Amyuni®

- Postscript to PDF Library: For document workflow applications that require processing of Postscript documents
- OCR Module: Free add-on to PDF Creator uses the Tesseract 3 engine for accurate character recognition
- Javascript Engine: Integrate a full JS interpreter into your PDF processing applications
- USB Mobile Monitor: Mirror the display of your Windows or Linux system onto your Android device

Windows Server 2016 Certified

Windows Server 2012 Certified

Windows

All development tools available at
www.amyuni.com

AMYUNI Technologies

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

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

JULIE LERMAN

Exploring the Multi-Model Capability of Azure Cosmos DB Using Its API for MongoDB

In last month's column (msdn.com/magazine/mt848702), and quite a few before that, I discussed different ways of working with Azure Cosmos DB, the globally distributed, multi-model database service supporting various NoSQL APIs. In all of my work thus far, however, I've used only one particular API—the SQL API—which allows you to interact with the data using SQL as the query language. One of the other models lets you interact with a Cosmos DB database using most of the tools and APIs available for MongoDB. This model also benefits from the BSON document format, a binary serialization format that's compact, efficient and provides concise data type encoding. But how is it possible for a single database to be accessed through these and the other models (Cassandra, Gremlin and Table)? The answer is found in the underlying database engine, which is based on what's called the atom-record-sequence (ARS) data model, enabling it to natively support a number of different APIs and data models. Each of the APIs is wire protocol-compatible to a popular NoSQL engine and data structure, such as JSON, by which you interact with the data.

It's important to understand that the multi-model APIs are currently not interchangeable. Cosmos DB databases are contained in a Cosmos DB Account. You can have multiple Cosmos DB accounts in your Azure subscription, but when you create an account, you select which API you'll be using for the databases in that account. Once selected, that's the only API you can use for its databases. Azure documentation uses terms like "currently" when discussing this, so the expectation is that at some point there will be more flexibility along these lines.

I've been curious about trying out another API. Except from some exploration of Azure Table Storage a while ago (see my July 2010 article at msdn.com/magazine/ff796231), which is aligned with the Table API in Azure Cosmos DB, I haven't ever used the other types of databases anyway. MongoDB is one of the most commonly used APIs, so that's the one I decided to investigate, and I've been having fun with my first explorations. I'll share some of what I've learned here, but please keep in mind that this is not intended as a "Getting Started with MongoDB" article. I'd recommend checking out Nuri Halperin's Pluralsight courses (bit.ly/2Sl2Vxw), which include beginner and expert MongoDB content. You'll find links to many other resources within the article, as well.

The MongoDB support is targeted more toward developers and systems that are already using MongoDB, because Azure provides a lot of benefits. There's even a guide for migrating data from existing MongoDB databases to Azure Cosmos DB (bit.ly/2FhzmPi).

Code download available at msdn.com/magazine/0219magcode.

The first benefit I was able to realize is that using MongoDB allows you to have a local instance of the database on your laptop to work with during development. While this is possible on Windows for the SQL and MongoDB APIs using the Cosmos DB Emulator (bit.ly/2sHNsAn), that emulator *only* runs on Windows. But you can install MongoDB—I'm using the Community Edition—on a variety of OSes, including macOS. This allows you to emulate the basic features of a Mongo DB API-driven Cosmos DB database locally. I'll start by working locally, getting a feel for MongoDB and then switching to an Azure database.

You can find installation instructions for all supported platforms in the official MongoDB documents at bit.ly/2S96ywj. Alternatively, you can pull a Docker image for running MongoDB on macOS, Linux or Windows from hub.docker.com/_/mongo.

Once it's installed, you start the process using the mongod.exe command. Unless otherwise specified, MongoDB expects you to have the directory /data/db created and to have permission to access. You can create this in the default location (for example, c:\data\db in Windows or in macOS, in the root folder where the Application, Library and Users folders live), or specify the location as a parameter of the "mongod" command. Because I'm just testing things out on my dev machine, I used "sudo mongod" to ensure that the service had the needed permissions. The Community Edition will run by default at localhost, port 27017, that is, 127.0.0.1:27017.

There are quite a few ways to interact with MongoDB. While I'm most interested in the C# API, I find it useful to start by working as close to "metal" as possible and then graduate to one of the APIs. If you're totally new to MongoDB, you might want to start by using its shell (installed along with the service) to do some work at the command line. You can start the shell just by typing "mongo." MongoDB installs three system databases: local, admin and config. In the shell, type "show dbs" to see them listed.

MongoDB doesn't have an explicit command to create a new database in the shell or other APIs. Instead, you use a database and the first time you insert data into it, if it doesn't exist, it will be created. Try this out with:

```
use myNewDatabase
```

If you call "show dbs" right afterward, you won't see myNewDatabase yet.

The shell uses a shortcut "db" to work with the current database object even if the actual database doesn't yet exist.

Now you need to insert a document. You create documents as JSON and the API will store them into MongoDB in its binary BSON format. Here's a sample document:

```
{
  firstname : "Julie",
  lastname : "Lerman"
}
```

But documents don't go directly in the database; they need to be stored in a collection. MongoDB follows the same behavior with collections as it does for the database: If you refer to a collection that doesn't yet exist, it will create it for you when inserting data into that collection. Therefore, you can reference a new collection while inserting a new document. Note that everything is case-sensitive. I'll use the insertOne method on a new collection called MyCollection to insert the document. Another document is returned that acknowledges that my document was inserted, and MongoDB provides a unique id key to my inserted document using its own data type, ObjectId:

```
> db.MyCollection.insertOne({firstname:"Julie",lastname:"Lerman"})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5c169d4f603846f26944937f")
}
```

Now "show dbs" will include myNewDatabase in its list and I can query the database with the "find" command of the collection object. I won't pass any filtering or sorting parameters, so it will return every document (in my case, only one):

```
>db.MyCollection.find()
{ "_id" : ObjectId("5c169d4f603846f26944937f"), "firstname" : "Julie",
  "lastname" : "Lerman" }
```

I've barely scratched the surface of the capabilities, but it's time to move on. Exit the mongo shell by typing "exit" at the prompt.

Using the Visual Studio Code Cosmos DB Extension

Now that you've created a new database and done a little work in the shell, let's benefit from some tooling. First, I'll use the Azure Cosmos DB extension in Visual Studio Code. If you don't have VS Code, install it from code.visualstudio.com. You can then install the extension from the VS Code extensions pane.

To open the local MongoDB instance in the extension, right-click on Attached Database Accounts and choose "Attach Database Account." You'll be prompted to choose from the Cosmos DB APIs. Select MongoDB. The next prompt is for the address where the database should be found. This will default to the MongoDB default: mongodb://127.0.0.1:27017. Once connected, you should be able to see the databases in the Cosmos DB explorer, including the one created in the shell with its collection and document, as shown in **Figure 1**.

If you edit an opened document, the extension will prompt you to update it to the server. This will happen whether you're connected to a local or to a cloud database.

MongoDB in a .NET Core App

Now let's step it up another notch and use the MongoDB database in a .NET Core app. One of the various drivers available is a .NET driver (bit.ly/2BvUEFl). The current version, 2.7, is compatible with .NET Framework 4.5 and 4.6, as well as .NET Core 1.0 and 2.0 (including minor versions). I'm using the latest .NET Core SDK (2.2) and will start by creating a new console project in a folder I named MongoTest, using the dotnet command-line interface (CLI) command:

```
dotnet new console
```

Next, I'll use the CLI to add a reference to the .NET driver for MongoDB, called mongocsharpdriver, into the project:

```
dotnet add package mongocsharpdriver
```

I'll create a simple app, leaning on last month's model with some classes related to the book and TV show, "The Expanse." I have two classes, Ship and Character:

```
public Ship()
{
  Crew=new List<Character>();
}
  public string Name { get; set; }
  public Guid Id { get; set; }
  public List<Character> Crew{ get; set;}
}
public class Character
{
  public string Name { get; set; }
  public string Bio {get;set;}
}
```

Notice that Ship has a Guid named Id. By convention, MongoDB will associate that with its required _id property. Character has no Id. I've made a modeling decision. In the context of interacting with ship data, I always want to see the characters on that ship. Storing them together makes retrieval easy. Perhaps, however, you're maintaining more details about characters elsewhere. You could embed an object that only has a reference to character Id's, for example:

```
public List<Guid> Crew{ get; set;}
```

But that means having to go find them whenever you want a ship with its list of characters. A hybrid alternative would be to add an Id property to Character, enabling me to cross-reference as needed. As a sub-document, the Character's Id would be just a random property. MongoDB requires only a root document to have an Id. But I've decided not to worry about the Id of Character for my first explorations.

There are many decisions to be made when modeling. Most importantly, if your brain defaults to relational database concepts, where you need to do a lot of translation between the relational data and your objects, you'll need to stop and consider the document database patterns and merits. I find this guidance on modeling docu-

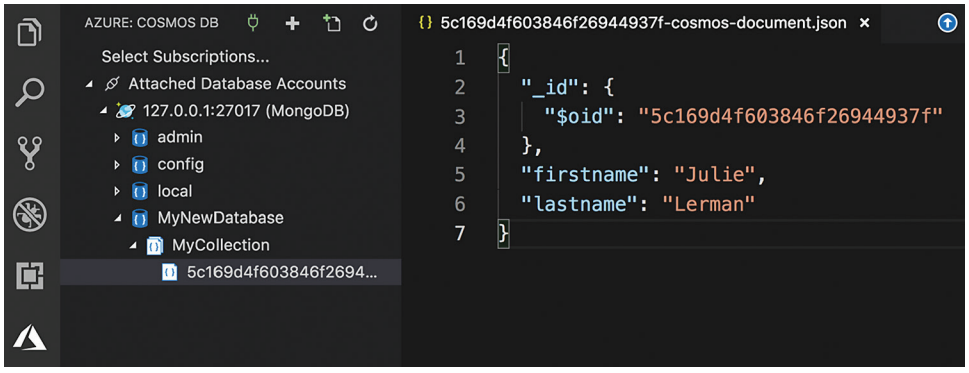


Figure 1 **Exploring the MongoDB Server, Databases and Data**

ment data for NoSQL database in the Azure Cosmos DB docs to be very helpful: bit.ly/2kpF46A.

There are a few more points to understand about the Ids. If you don't supply a value to the ship's Id property, MongoDB will create the value for you, just as SQL Server or other databases would. If the root document doesn't have any property that maps to the _id of the stored document (by convention or your own mapping rules), it will fail when attempting to serialize results where the _id is included.

Working with the mongocsharpdriver API

The .NET driver's API starts with a MongoClient instance and you work from there to interact with the database and collection and documents. The API reflects some of the concepts I already demonstrated with the shell. For example, a database and collection can be created on the fly by inserting data. Here's an example of that using the new API:

```
private static void InsertShip () {
  var mongoClient = new MongoClient ();
  var db = mongoClient.GetDatabase ("ExpanseDatabase");
  var coll = db.GetCollection<Ship> ("Ships");
  var ship = new Ship { Name = "Donnager" };
  ship.Characters.dd(new Character { Name = "Bobbie Draper",
    Bio="Fierce Marine"});
  coll.InsertOne (ship);
}
```

Where I used the command "use database" in the shell, I now call GetDatabase on the MongoClient object. The database object has a generic GetCollection<T> method. I'm specifying Ship as the type. The string "Ships" is the name of the collection in the database. Once that's defined, I can InsertOne or InsertMany, just like in the shell. The .NET API also provides asynchronous counterparts, such as InsertOneAsync.

The first time I ran the InsertShip method, the new database and collection were created along with the new document. If I hadn't inserted the new document and had only referenced the database and collection, they wouldn't have been created on the fly. As with the shell, there's no explicit command for creating a database.

Here's the document that was created in the database:

```
{
  "_id": {
    "$binary": "TbKPi3+tLUK9b68lJkGaww==",
    "$type": "3"
  },
  "Name": "Donnager",
  "Characters": [
    {
      "Name": "Bobbie Draper",
      Bio: "Fierce Marine"
    }
  ]
}
```

What's more interesting (to me), however, is the typed collection (GetCollection<Ship>). The MongoDB documentation describes a collection as "analogous to tables in relational databases" (bit.ly/2QZcOcD), which is an interesting description for a document database where you can store random, unrelated documents into a collection. Still, tying a collection to a particular type, as with the "ships" collection, does suggest that I'm enforcing the schema of the ship type in this collection. But this is for the Collection instance, not the physical collection in the database. It informs the particular instance how to serialize and deserialize objects, given that you can store data from any object into a single collection. As of version 3.2, MongoDB did add a feature that enforces schema validation rules, though that's not the default.

I can use the same Ships collection for other types:

```
var collChar = db.GetCollection<SomeOtherTypeWithAnId> ("Ships");
```

However, this would create a problem when it's time to retrieve data. You'd need a way to identify document types in the collection. If you read last month's article about the Cosmos DB provider for EF Core (which uses the SQL API), you may recall that when EF Core inserts documents into Cosmos DB, it adds a Discriminator property so you can always be sure what type a document aligns to. You could do the same for the MongoDB API, but that would be a bit of a hack because MongoDB uses type discriminators for specifying object inheritance (bit.ly/2sbHvgA). I've added a new class, DecommissionedShip, that inherits from Ship:

```
public class DecomissionedShip : Ship {
  public DateTime Date { get; set; }
}
```

The API has a class called BsonClassMap used to specify custom mappings, including its SetDiscriminatorIsRequired method. This will inject the class name by default. Because you'll be overriding the default mapping, you need to add in the Automap method, as well.

I've added a new method, ApplyMappings, into program.cs and am calling it as the first line of the Main method. This specifically instructs the API to add discriminators for Ship and DecommissionedShip:

```
private static void ApplyMappings () {
    BsonClassMap.RegisterClassMap<Ship> (cm => {
      cm.AutoMap ();
      cm.SetDiscriminatorIsRequired (true);
    });
    BsonClassMap.RegisterClassMap<DecommissionedShip> (cm => {
      cm.AutoMap ();
      cm.SetDiscriminatorIsRequired (true);
    });
  }
```

I've modified the InsertShip method to additionally create a new DecommissionedShip. Because it inherits from Ship, I can use a single Ship collection instance and its InsertMany command to add both ships to the database:

```
var decommissionedShip=new DecommissionedShip{Name="Canterbury",
  Date=new DateTime(2350,1,1)};
coll.InsertMany(new[]{ship,decommissionedShip});
```

Both documents are inserted into in the Ships collection and each has a discriminator added as property "_t." Here's the DecommissionedShip:

```
{
  "_id": {
    "$binary": "D1my7H9MrkmGzzJGSHOZfA==",
    "$type": "3"
  },
  "_t": "DecommissionedShip",
  "Name": "Canterbury",
  "Characters": [],
  "Date": {
    "$date": "2350-01-01T05:00:00.000Z"
  }
}
```

When retrieving the data from a collection typed to Ship, as in this GetShip method:

```
private static void GetShips ()
{
  var coll = db.GetCollection<Ship> ("Ships");
  var ships = coll.AsQueryable ().ToList ();
}
```

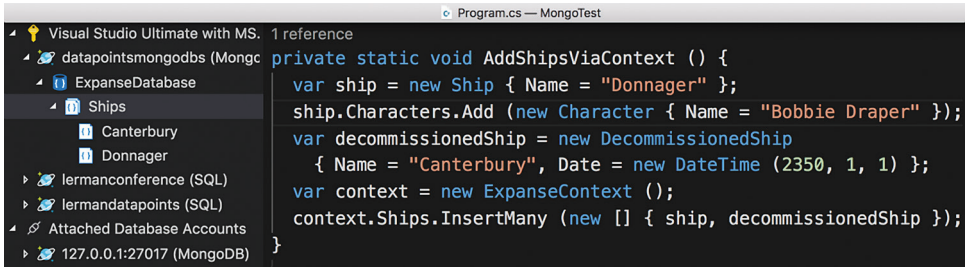


Figure 2 **The Newly Created Database, Collection and Documents in the Cosmos DB Database**

the API reads the discriminators and materializes both the Ship and DecommissionedShip objects with all of their data intact, including the Date assigned to the DecommissionedShip.

Another path for mapping is to use a BsonDocument typed collection object that isn't dependent on a particular type. Check my blog post, "A Few Coding Patterns with the MongoDB C# API" (bit.ly/2FeWcrK), to see how to use BsonDocuments, as well as how to encapsulate the MongoClient, Database and Collection for more readable code.

Use LINQ for Querying

You can retrieve documents with the .NET API using API methods or LINQ. The API uses a very rich Find method (similar to the shell's find method), which returns a cursor. You can pass in filters, project properties and return objects using one of its execution or aggregation methods—many of which look like LINQ methods. The .NET API Find method requires a filter, so to get any and all documents, you can filter on new (empty) BsonDocument, which is a filter matching any document. For LINQ, you first need to transform a collection to an IQueryable (using AsQueryable()) and then use the familiar LINQ methods to filter, sort and execute. If you didn't include or map the _id property in your classes, you'll need to use projection logic to take that into account as it *will* be returned from the query. You can refer to the documentation or to other articles (such as the great series by Peter Mbanugo at bit.ly/2Lqqw2J) to learn more of these details.

Switching to Azure Cosmos DB

After you've worked out your persistence logic locally against the MongoDB instance, eventually you'll want to move it to the cloud-based Cosmos DB. Visual Studio Code's Azure Cosmos DB extension makes it easy to create a new Cosmos DB account if you don't have one yet, although you'll likely want to tweak its settings in the portal. If you're using Visual Studio, the Cloud Explorer for VS2017 extension has features for browsing but not creating databases, so in that case you'll need to use the Azure CLI or work in the portal.

Here's how you can create a new instance with the Azure Cosmos DB API for MongoDB from scratch using the VS Code extension.

First, you'll need VS Code to be connected to your Azure subscription. This requires the Azure Account extension (bit.ly/2k1phdp), which, once installed, will help you connect. And once connected, the Cosmos DB extension will display your subscriptions and any existing databases. As with the MongoDB local connection shown in **Figure 1**, you can drill into your Cosmos DB accounts, databases, collections and documents (the Cosmos DB terms for these are *containers* and *items*). To create a brand-new account, right-click on the plus sign at the top of the extension's explorer. The workflow will be similar to creating a local MongoDB database, as I did earlier. You'll be prompted to enter an account name. I'll use datapointsmongodbs. Next, choose MongoDB from the available APIs and either create a new Azure resource or choose an existing one to tie to the account. I created a new one so that I can cleanly delete the resource and the test database as needed. After this, you have to select from among the datacenter regions where this one should be hosted. I live in the eastern United States so I'll pick the East US location. Given that Cosmos DB is a global database, you can control the use of regions in the portal or other apps, but I won't need that for my demo. At this point, you'll need to wait a few minutes while the account is created.

Once the new account is created, it will show up in the explorer. Right-click the account and select the "Copy Connection String" option. You can use this to change the MongoDB driver to point to the Azure Cosmos DB instance instead of pointing to the default local instance, as I've done here:

```
var connString=
  "mongodb://datapointsmongodbs:****.documents.azure.com:10255/?ssl=true";
ExpanseDb=new MongoClient(connString).GetDatabase("ExpanseDatabase");
```

I'll run a refactored version of the method that inserts a new Ship and a new DecommissionedShip into the Ships collection of the ExpanseDatabase. After refreshing the database in the explorer, the explorer displays the newly created database in my Azure account, collection and documents in the Cosmos DB database, as shown in **Figure 2**.

Not a Mongo DB Expert, but a Better Understanding of Multi-Model

The availability of this API is not meant to convince users like me with decades of experience with SQL to switch to using MongoDB for my Cosmos DB databases. There would be so much for me to learn. The real goal is to enable the myriad developers and teams who already use MongoDB to have a familiar experience while gaining from the many benefits that Azure Cosmos DB has to offer. I undertook my own exploration into the Azure Cosmos DB API for MongoDB to gain a better understanding of the Cosmos DB multi-model capability, as well as to have a little fun checking out a new database. And, hopefully, my experience here will provide some high-level guidance for other developers or clients in the future. ■

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

Thanks to the following technical expert for reviewing this article: *Nuri Halperin (Plus N Consulting)*

TED NEWARD

Coding Naked: Naked Speakers

In the last piece, I introduced you to the Naked Objects Framework (NOF), a framework that looks to hide away all of the work around UI and data persistence of objects, leaving developer focus solely on building domain objects the way you were always taught in school, before you learned about MVC and Web APIs and HTTP and SQL and database connection management and dependency injection and all the other things that consume the vast majority of our time and energy. It's a great idea, if it works, but the last article didn't exactly overwhelm with features. In this article, I'll take a deeper look into what you can define on a class with NOF and how that will translate into UI and database persistence.

Naked Concepts

Before I get too far into the code, though, let's get comfortable with some of the NOF terminology, because it will provide a certain amount of insight into how the creators and maintainers see the world.

First and foremost, at the center of the whole experience are domain objects—the objects that represent the state and behavior of the system. In NOF, any "plain old C# object" can be a domain object, as long as it follows a few rules so that NOF can provide the support it needs: properties must be virtual; any collection members must be initialized; and NOF recommends against constructors because the object in memory may have a different lifecycle than you're used to. I'll talk about this more later.

Not everything fits well into the object paradigm, however. Often there are behaviors that operate on objects, but don't belong particularly well inside of one. Rather than try and force behaviors into objects in an arbitrary way, NOF provides support for *services*, which provide behaviors "outside" of objects, such as creating, retrieving or other operations on domain objects. Services can also serve as a gateway to external system services like e-mail servers.

From a user perspective, the UI needs to provide behaviors that users can trigger, and these are collectively called actions, which often correspond to menus in the UI. Actions can be discovered (or "contributed") from a variety of sources, but most often they'll be found via Reflection against domain objects or services, and displayed in the UI as seen. You can use custom attributes to tailor much of this discovery and display, when and where necessary, and I'll examine more of this as I go.

This will all become clearer via code, so … shall we?

Naked Conference

In the MEAN series, I used a simple domain example (speakers delivering talks at a conference) as a backdrop for building code, and it seems reasonable to use the same example here, if for no other reason than to compare apples to apples. As domain models go, it'll be pretty simple: Speakers consist of first name, last name, age, subjects (C#, VB, Java, JavaScript, TypeScript and C++), and average rating (from zero to five), to start. Later I can add Presentations, which will have a title and description, once you see how to set up relationships between objects, but let's keep it simple for now.

As mentioned in the last article, I'll start by using the NOF Template .zip as the seed from which to start, so download that from the link in the NOF GitHub project README (bit.ly/2PMojiQ), unzip it into a freshly minted subdirectory, and open the solution in Visual Studio. Now open the Template.Model project node and delete the Student.cs and ExampleService.cs files; you don't want to store Students, after all. What you do want to store are Speakers (and later Presentations, but I'll leave that for refactoring later, just to see how well NakedObjects will support refactoring).

To be clear, I need to do a couple of things to support speakers as a domain type in a Naked Objects project: I need to create the domain type; I need to make sure I have a Repository object to obtain and create them; I need to make sure that a repository is made available to the Naked Objects framework to generate menus from; and, finally, I need (or, to be honest, want) to seed the database with some speakers.

Speaker Objects

Building out the Speaker class is actually the simplest of the four steps. In the Template.Model project, create a Speaker.cs file, and in that file, inside the Template.Model namespace, create a public class called Speaker. Speakers are defined (for the moment) by first name, last name and age, so create three properties for them, but mark them "virtual"; this is so that at runtime, NakedObjects can slip some additional behavior into place (via a subclass) that does the desired magic.

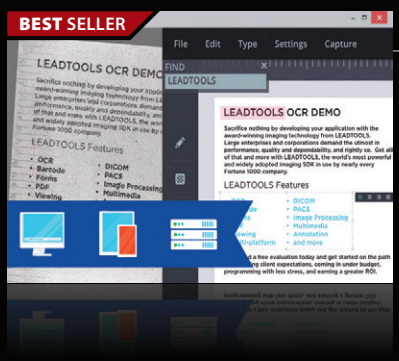
Let's add a little complexity to Speaker, however. First of all, you often want a non-domain-related identifier for objects stored in a database that can be used as a primary key, without allowing users to use or modify that identifier. In the NakedObjects system, I can

Figure 1 **The SpeakerRepository Class**

```
public class SpeakerRepository
{
  public IDomainObjectContainer Container { set; protected get; }

  public Speaker CreateNewSpeaker()
  {
    // 'Transient' means 'unsaved' - returned to the user
    // for fields to be filled in and the object saved.
    return Container.NewTransientInstance<Speaker>();
  }

  public IQueryable<Speaker> AllSpeakers()
  {
    return Container.Instances<Speaker>();
  }
}
```

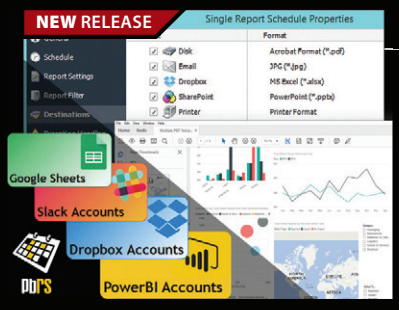


LEADTOOLS Document Imaging SDKs V20 | from $2,995.00 SRP

LEADTOOLS
THE WORLD LEADER IN IMAGING SDKS

Add powerful document imaging functionality to desktop, tablet, mobile & web applications.

- Universal document viewer & conversion framework for PDF, Office, CAD, TIFF & more
- OCR, MICR, OMR, ICR and Forms Recognition supporting structured & unstructured forms
- PDF SDK with text edit, hyperlinks, bookmarks, digital signature, forms, metadata
- Barcode Detect, Read, Write for UPC, EAN, Code 39, Code 128, QR, Data Matrix, PDF417
- Zero-footprint HTML5/JavaScript UI Controls & Web Services



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

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

flag any property with a "NakedObjectsIgnore" custom attribute so that it won't show up in the UI, so let's add an integer "Id" property (again virtual) that's flagged with this attribute. Additionally, it's not uncommon to want to create a property that's "computed" using other data elements, so let's add a "FullName" read-only property that concatenates first and last name. Because NakedObjects isn't going to be responsible for editing the UI on this, it doesn't need to be virtual.

When that's done, you should be left with something like this:

```
public class Speaker
{
  [NakedObjectsIgnore]
  public virtual int Id { get; set; }

  public virtual string FirstName { get; set; }
  public virtual string LastName { get; set; }
  public virtual int Age { get; set; }

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

So far, so good.

Speaker Repository

A repository object, for those not familiar with the pattern, is essentially an object that represents access to the database, providing easy method calls for standard database requests. In the NakedObjects system, a repository object is a form of service, which is an object specifically recognized by NakedObjects as an object that isn't a domain object but can provide UI elements and behavior on behalf of the system. In this case, the repository will provide some menu items for getting speakers out of the database, sorted (for example) by last name.

In practical terms, a NakedObject service is just another class, but it will have something the Speaker class doesn't: a reference to an IDomainObjectContainer object, which is an object provided by the NakedObjects framework that in turn provides access to the rest of the system. The list of system services available is given in the NakedObjects documentation, but specifically I'm interested in two elements: the NewTransientInstance method, which will construct a new Speaker object and wire it up against the rest of the NakedObjects infrastructure for use; and the Instances method, which will return the complete set of Speaker objects from the database. Thus, a minimal SpeakerRepository looks like what's shown in **Figure 1**.

Notice that both the NewTransientInstance and the Instances methods are generic functions, taking the type of object they're working with as a type parameter to the function call. (This is necessary because the DomainObjectContainer can and will be used for a variety of different domain object types, not just Speakers.)

A repository that just creates a new Speaker and gets a list of all of them isn't particularly useful, however; you'll often want to retrieve objects with certain ones filtered out, or sorted in a particular order. Here's where the power of LINQ and EntityFramework inside the NakedObjects framework makes its presence felt. Because the AllSpeakers method returns an IQueryable, I can use LINQ methods to filter and/or sort the objects returned, so that if I want to find Speakers by last name, I can simply add that method to the repository class, using LINQ as necessary to filter the results:

```
public class SpeakerRepository
{
  // ... as before

  public IQueryable<Speaker> FindSpeakerByLastName(string name)
  {
    return AllSpeakers().
      Where(c => c.LastName.ToUpper().Contains(name.ToUpper()));
  }
}
```

Anything LINQ can do, you can express as a method on the repository.

Figure 2 **Adding Speaker Objects to the DbContext**

```
public class ExampleDbInitializer :
  DropCreateDatabaseIfModelChanges<ExampleDbContext>
{
  private ExampleDbContext Context;
  protected override void Seed(ExampleDbContext context)
    {
    this.Context = context;

    Context.Speakers.Add(new Speaker() {
      FirstName = "Ted", LastName = "Neward", Age = 47
    });
  }
}
```

Speaker Context

Two more bookkeeping details I need to take care of: I need to set up the Entity Framework DbContext that will be used by the NakedObject persistence engine, and I need to hook the SpeakerRepository into the UI so I can have menu items to create new Speakers, find existing Speakers, and so on.

The first is found in the Template.DataBase project, in the ExampleDbContext.cs file (whose name you should feel free to change if it matters to you), in the ExampleDbContext class. As a public property on that class, a DbSet needs to be parameterized to the Speaker type called "Speakers," like so:

```
namespace Template.DataBase
{
  public class ExampleDbContext : DbContext
  {
    public ExampleDbContext(string dbName,
      IDatabaseInitializer<ExampleDbContext> initializer) : base(dbName)
    {
      Database.SetInitializer(initializer);
    }

    public DbSet<Speaker> Speakers { get; set; }
  }
}
```

As more top-level domain object types are added to the system, I'll need to add more properties here that return a DbSet, but for now, this is all I need.

If you want the database to be seeded with some speakers to start, you need to crack open the Template.SeedData project and find the ExampleDbInitializer.cs file and write the Seed method (which is already defined in that file) to add some Speaker objects to the ExampleDbContext (the object I just refactored in the previous paragraph) via the Speakers DbSet, as shown in **Figure 2**.

Note that this is straight Entity Framework code, so any further discussion about how to change the drop/create policy of the database would fall into the Entity Framework documentation set.

The last place I need to put code is in the Template.Server project, in the NakedObjectsRunSettings.cs file; this code is a collection of registration "hooks" or "hints" to the NakedObjects

Figure 3 **NakedObjects Run Settings**

```
public class NakedObjectsRunSettings
{
  // ...

  private static Type[] Services
  {
    get
    {
      return new Type[] {
        typeof(SpeakerRepository)
      };
    }
  }

  // ...

  public static IMenu[] MainMenus(IMenuFactory factory)
  {
    return new IMenu[] {
      factory.NewMenu<SpeakerRepository>(true, "Menu")
    };
  }
}
}
```

system, describing what the NakedObjectsFramework needs to know. For example, if you decide that "Template.Model" is a terrible namespace for the model objects for this conference app, then feel free to change it, so long as the ModelNamespaces of this NakedObjectsRunSettings also reflects the changed name.

With respect to the SpeakerRepository, I need to let NakedObjects know about it by adding it to the "Services" Type array. If I want the SpeakerRepository methods to be accessible via a menu option, I also need to tell NakedObjects to examine the SpeakerRepository for actions to use as menu items, so I need to include it as part of the array of menu items returned from the MainMenus method, as shown in **Figure 3**.

Once these changes are in place, I can fire up the projects, bring up the client and create a few speakers. Notice that the FullName property is displayed, built out of FirstName and LastName, but Id doesn't appear. Additionally, thanks to the FindSpeakersByLastName method on the SpeakerRepository, I can search for speakers by last name, and if more than one is returned, the client automatically knows how to present a list of them that can be selected for individual work.

Wrapping Up

Notice that at no point in this article did I do anything to the Template.Client project—that's the point! The UI is constructed out of the domain objects and services, with supplemental information provided by custom attributes (which you'll see more of in later columns). The persistence is, similarly, built out of the structure and metadata that the framework discovers from your types.

What you'll need to see, going forward, is how NOF handles more complex domain scenarios—like Speakers having a list of subjects on which they can speak and a collection of Presentations they can deliver—and how it presents a UI for them. In the next article, I'll look at some of these topics. In the meantime, however ... 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.*

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!

February 6-7, 2019

Microtek Training Center Dallas

Want to learn more on ASP.NET Core in the Microsoft Cloud? Check out Visual Studio Live!'s upcoming Training Seminar to expand your knowledge and accelerate your career.

REGISTER NOW!
vslive.com/dallas

March 3-8, 2019

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, 2019

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, 2019

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

August 12-16, 2019

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

September 29, 2019

Westin Gas Lamp

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.

DETAILS COMING SOON!

October 6-10, 2019

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.

DETAILS COMING SOON!

November 17-22, 2019

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!

FRANK LA VIGNE

A Closer Look at Neural Networks

Neural networks are an essential element of many advanced artificial intelligence (AI) solutions. However, few people understand the core mathematical or structural underpinnings of this concept. While initial research into neural networks dates back decades, it wasn't until recently that the computing power and size of training datasets made them practical for general use.

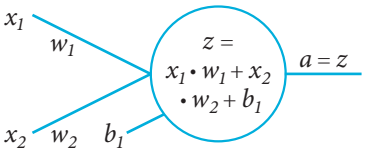


Figure 1 **Forward Propagation in a Neural Network**

Neural networks, or more specifically, artificial neural networks, are loosely based on biological neural networks in the brains of animals. While not an algorithm per se, a neural network is a kind of framework for algorithms to process input and produce a "learned" output. Neural networks have proven themselves useful at performing tasks that traditional programming methods have severe difficulty solving. Although there are several different variations of neural networks, they all share the same core structure and concepts. There exist frameworks, like Keras, designed to make them easier to implement and to hide many implementation details. However, I never fully grasped the power and the beauty of neural networks until I had to program one manually. That will be the aim of this column: to build out a simple neural network from scratch with Python.

Neurons and Neural Networks

Before building a neural network from scratch, it's important to understand its core components. Every neural network consists of a series of connected nodes called neurons. In turn, each neuron is part of a network of neurons arranged in layers. Every neuron in each layer is connected to every neuron in the next layer. Each artificial neuron takes in a series of inputs and computes a weighted sum value. The neuron will activate or not based on the output of the activation function, which takes in all the input values and weights along with a bias and computes a number. This number is a value between -1 and 1 or 0 and 1, depending on the type of activation function. This value is then passed on to other connected neurons into the next layer in a process called forward propagation.

As for the layers, there are three basic kinds: input layers, hidden layers and output layers. Input layers represent the input data while the output layer contains the output. The hidden layers determine the depth of the neural network—this is where the term "deep learning" comes from. In practice, neural networks can have hundreds of hidden levels, with the only upper limit being available processing power.

Forward Propagation

Forward propagation is the process by which data flows through a neural network from the input layer to the output layer. It involves computing a weighted sum of all inputs and factoring in a bias. Once the weighted sum is computed, it is then run through an activation function. Using the neuron in **Figure 1** as an example, the neuron has two inputs, x1 and x2, along with two weights, w1 and w2. The weighted sum is represented by z, while the value a represents the value computed by the activation function when it's passed the value of the weighted sum. Recall that the goal of the activation function is to compress the output of the neuron into a value between a range. Bias is added to fine-tune the sensitivity of the neuron.

To illustrate this, it may be best to go through the code. To keep things clear, I'll use variable names that match up with those in **Figure 1**. Start by launching a Jupyter Notebook (for more on that, go here: msdn.com/magazine/mt829269), then enter then following into a blank cell and execute:

```
x1 = .5
w1 = .2
x2 = 4
w2 = .5
b = .03

z = x1 * w1 + x2 * w2 + b

print (z)
```

The value of z, the weighted sum, is 2.13. Recall that the next step is to run this value through an activation function. Next, enter the following code into a new cell to create an activation function and execute it:

```
import numpy as np
def sigmoid_activation(weighted_sum):
  return 1.0 / (1.0 + np.exp(-1 * weighted_sum))
a = sigmoid_activation(z)
print(a)
```

The output should show that a is equal to 0.8937850083248244. In a multi-layer neural network, the value of a is passed on to the next layer. Therefore, activations in one layer cascade to the next and, eventually, through the entire network.

The sigmoid_activation returns values between 0 and 1 no matter how large or small the number is. Enter the following code to test it:

```
print(sigmoid_activation(1000000))
print(sigmoid_activation(.000001))
```

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

The output should read 1.0 and roughly 0.50, respectively.

For more information about the mathematical constant e, please refer to the Wikipedia article on the subject (bit.ly/2s8VL9Q).

How Does a Neural Network Work?

Given the relative simplicity of the structure and mathematics of a neural network, it's natural to wonder how it can be applied to a wide array of AI problems. The power is in the network, not necessarily the neurons themselves. Each neuron in a neural network represents a combination of input values, weights and biases. Through training, the appropriate weights and values can be determined.

By now, you've undoubtedly heard about the MNIST dataset, which is commonly used as a sort of "Hello World" for neural networks. I had seen it dozens of times before the notion of how neural networks functioned finally clicked for me. If you're not familiar with the problem, there are plenty of examples online that break it down (see varianceexplained.org/r/digit-eda). The MNIST dataset challenge nicely illustrates how easy it is for neural networks to take on tasks that have baffled traditional algorithmic approaches for decades.

Here's the problem summarized, given a 28x28 pixel grayscale image of a handwritten digit that a neural network must learn to read as the correct value. That 28x28 pixel image consists of 784 individual numerical values between zero and 255, making it easy to imagine the structure of the input layer. The input layer consists of 784 neurons, with values passed through an activation function that ensures that values are between zero and one. Therefore, lighter pixels will have a value closer to one and darker pixels will have a value closer to zero. The output layer consists of 10 neurons, one for each digit. The neuron with the highest value represents the answer. For instance, if the neuron for eight has the highest activation function value, then the neural network has determined that eight is the output value.

Figure 2 **Building a Neural Network**

```
def initialize_neural_network(num_inputs, num_hidden_layers,
  num_nodes_hidden, num_nodes_output):

  num_nodes_previous = num_inputs # number of nodes in the previous layer

  network = {}

  # Loop through each layer and randomly initialize
  # the weights and biases associated with each layer.
  for layer in range(num_hidden_layers + 1):

    if layer == num_hidden_layers:
      layer_name = 'output'
      num_nodes = num_nodes_output
    else:
      layer_name = 'layer_{}'.format(layer + 1)
      num_nodes = num_nodes_hidden[layer]

    # Initialize weights and bias for each node.
    network[layer_name] = {}
    for node in range(num_nodes):
      node_name = 'node_{}'.format(node+1)
      network[layer_name][node_name] = {
        'weights': np.around(np.random.uniform(size=num_nodes_previous),
          decimals=2),
        'bias': np.around(np.random.uniform(size=1), decimals=2),
      }

    num_nodes_previous = num_nodes

  return network
```

Just adding two hidden layers, with 32 neurons each, will have a dramatic effect. How so? Recall that each neuron is connected to every neuron in the previous layer and the next layer. That means that there are 784x32 weights in the first layer, 32x32 weights in the second layer, and 32x10 weights in the third layer. There are also 32 + 32 + 10 biases that need to be added, as well. That yields a grand total of 26,506 adjustable values, and in this relatively simple example of a three-layer neural network, just over 26,506 weights and biases. Effectively, this means that there are 26,506 parameters to adjust to achieve an ideal output. For an excellent visualization and explanation of this structure and the power behind it, watch the "But What *Is* a Neural Network? | Deep Learning, Chapter 1" video on YouTube at bit.ly/2RziJVW. And for interactive experimentation with neural networks, be sure to check out playground.tensorflow.org.

Keep in mind that a real-world neural network may have hundreds of thousands of neurons across hundreds of layers. It is this substantial quantity of parameters that gives neural networks their uncanny ability to perform tasks that have traditionally been beyond the capability of computer programs. With all these "knobs and dials" it's little wonder how these relatively simple structures can tackle so many tasks. This is also why training a neural network requires so much processing power and makes GPUs ideal for this kind of massively parallel computation.

Building a Neural Network

With the structure and mathematics explained, it's time to build out a neural network. Enter the code in **Figure 2** into a new cell and execute it. The initialize neural network function simplifies the creation of multilayer neural networks. To create a neural network with 10 inputs, two outputs, and five hidden layers with 32 nodes each, enter the following code into a blank cell and execute it:

```
network1 = initialize_neural_network(10, 5, [32, 32, 32, 32, 32], 2)
```

To create a network that has a structure to match the previous neural network described to solve the MNIST challenge, adjust the parameters as follows:

```
mnist_network = initialize_neural_network(784, 2, [32, 32], 10)
print(network1)
```

This code creates a neural network with 784 input nodes, two hidden layers with 32 nodes each, and an output layer of 10 nodes. Note that the output displays the networks in JSON format. Also note that the weights and biases are initialized to random values.

Exploring the Neural Network

So far, we have the structure of a neural network but we haven't done anything with it. Now, let's create some input for network1, which has 10 input nodes, like so:

```
from random import seed
np.random.seed(2019)
input_values = np.around(np.random.uniform(size=10), decimals=2)

print('Input values = {}'.format(input_values))
```

The output will be a numpy array of 10 random values that will serve as the input values for the neural network. Next, to view the weights and biases for the first node of the first layer, enter the following code:

```
node_weights = network1['layer_1']['node_1']['weights']
node_bias = network1['layer_1']['node_1']['bias']

print(node_weights)
print(node_bias)
```

Note that there are 10 values for the weights and one value for the bias. Next, enter the following code to create a function that calculates the weighted sum:

```
def calculate_weighted_sum(inputs, weights, bias):
  return np.sum(inputs * weights) + bias
```

Now enter the following code to compute and display the weighted sum (z) for this node:

```
weighted_sum_for_node = calculate_weighted_sum(inputs, node_weights, node_bias)
print('Weighted sum for layer1, node1 = {}'.format(
  np.around(weighted_sum_for_node[0], decimals=2)))
```

The value returned should be 3.15. Next, use the sigmoid_activation function to compute a value for this node, as follows:

```
node_output_value  = sigmoid_activation(weighted_sum_for_node)
print('Output value for layer1, node1 =
  {}'.format(np.around(node_output_value[0], decimals=2)))
```

The final output value for this node is 0.96. It's this value that will be passed on to all the neurons in the next layer.

Feel free to experiment and iterate though any of the 5,600 or so nodes in this network. Alternatively, you could repeat the steps I described for each of these nodes to get an appreciation for the sheer volume of calculations in a neural network. Naturally, this is a task better performed programmatically. I will cover this and how to train a neural network in the next column.

Wrapping Up

Neural networks have led to incredible advances in AI and have been applied to difficult real-world problems such as speech recognition and computer vision with great success. While their structures may be complex, they're made of relatively simple building blocks. Neurons are arranged in layers in a neural network and each neuron passes on values. Input values therefore cascade through the entire network and influence the output.

Neurons themselves are simple and perform basic mathematical functions. They become powerful, however, when they're connected to each other. The sheer number of tweakable values even in simple neural networks provides a great deal of control over the output and can prove useful in training.

While this article focused on building a neural network in Python, just about any programming language could be used to create a neural network. There are examples online of this being done in JavaScript, C#, Java or any number of modern languages. Where Python excels, however, is in the availability of widely supported frameworks, such as Keras, to make the creation of neural networks simpler. ■

Frank La Vigne *works at Microsoft as an AI Technology Solutions Professional where he helps companies achieve more by getting the most out of their data with analytics and AI. He also co-hosts the DataDriven podcast. He blogs regularly at FranksWorld.com and you can watch him on his YouTube channel, "Frank's World TV" (FranksWorld.TV).*

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

Minimize Complexity in Multithreaded C# Code

Thomas Hansen

Forks, or multithreaded programming, are among the most difficult things to get right when programming. This is due to their parallel nature, which requires a completely different mindset than linear programming with a single thread. A good analogy for the problem is a juggler, who must keep multiple balls in the air without having them negatively interfere with each other. It's a major challenge. However, with the right tools and the right mindset, it's manageable.

In this article, I dive into some of the tools I've crafted to simplify multithreaded programming, and to avoid problems such as race conditions, deadlocks and other issues. The toolchain is based, arguably, on syntactic sugar and magic delegates. However, to quote the great jazz musician Miles Davis, "In music, silence is more important than sound." The magic happens between the noise.

Put another way, it's not necessarily about what you *can* code, but rather what you can but choose *not* to, because you'd rather create a bit of magic between the lines. A quote from Bill Gates comes to mind: "To measure quality of work according to the number of lines of code, is like measuring the quality of an airplane by its weight." So, instead of teaching you how to code more, I hope to help you code less.

This article discusses:

- Managing the complexity of multithreaded code
- Using delegates to enable loosely coupled, weakly typed solutions
- Producing concise code that adheres to don't repeat yourself (DRY) principles

Technologies discussed:

C#, Delegates, Multithreading

The Synchronization Challenge

The first problem you'll encounter with multithreaded programming is synchronizing access to a shared resource. Problems occur when two or more threads share access to an object, and both might potentially try to modify the object at the same time. When C# was first released, the lock statement implemented a basic way to ensure that only one thread could access a specified resource, such as a data file, and it worked well. The lock keyword in C# is so easily understood, that it single-handedly revolutionized the way we thought about this problem.

However, a simple lock suffers from a major flaw: It doesn't discriminate read-only access from write access. For instance, you might have 10 different threads that want to read from a shared object, and these threads can be given simultaneous access to your instance without causing problems via the ReaderWriterLockSlim class in the System.Threading namespace. Unlike the lock statement, this class allows you to specify if your code is writing to the object or simply reading from the object. This enables multiple readers entrance at the same time, but denies any write code access until all other read and write threads are done doing their stuff.

Now the problem: The syntax when consuming the ReaderWriterLock class becomes tedious, with lots of repetitive code that reduces readability and complicates maintenance over time, and your code often becomes scattered with multiple try and finally blocks. A simple typo can also produce disastrous effects that are sometimes extremely difficult to spot later.

By encapsulating the ReaderWriterLockSlim into a simple class, all of a sudden it solves the problem without repetitive code, while reducing the risk that a minor typo will spoil your day. The class, as shown in **Figure 1**, is entirely based on lambda trickery. It's arguably just syntactic sugar around some delegates, assuming the existence of a couple interfaces. Most important, it can help make your code much more DRY (as in, "Don't Repeat Yourself").

There are just 27 lines of code in **Figure 1**, providing an elegant and concise way to ensure that objects are synchronized across multiple threads. The class assumes you have a read interface and a write interface on your type. You can also use it by repeating the template class itself three times, if for some reason you can't change the implementation of the underlying class to which you need to synchronize access. Basic usage might be something like that shown in **Figure 2**.

In the code in **Figure 2**, regardless of how many threads are executing your Foo method, no Write method will be invoked as long as another Read or Write method is being executed. However, multiple Read methods can be invoked simultaneously, without having to scatter your code with multiple try/catch/finally statements, or repeating the same code over and over. For the record, consuming it with a simple string is meaningless, because System.String is immutable. I use a simple string object here to simplify the example.

The basic idea is that all methods that can modify the state of your instance must be added to the IWriteToShared interface. At the same time, all methods that only read from your instance should be added to the IReadFromShared interface. By separating your concerns like this into two distinct interfaces, and implementing both interfaces on your underlying type, you can then use the Synchronizer class to synchronize access to your instance. Just like that, the art of synchronizing access to your code becomes much simpler, and you can do it for the most part in a much more declarative manner.

When it comes to multithreaded programming, syntactic sugar might be the difference between success and failure. Debugging multithreaded code is often extremely difficult, and creating unit tests for synchronization objects can be an exercise in futility.

If you want, you can create an overloaded type with only one generic argument, inheriting from the original Synchronizer class and passing on its single generic argument as the type argument three times to its base class. Doing this, you won't need the read or write interfaces, since you can simply use the concrete implementation of your type. However, this approach requires that you manually take care of those parts that need to use either the Write or Read method. It's also slightly less safe, but does allow you to wrap classes you cannot change into a Synchronizer instance.

Lambda Collections for Your Forks

Once you've taken the first steps into the magic of lambdas (or delegates, as they're called in C#), it's not difficult to imagine that you can do more with them. For instance, a common recurring theme in multithreading is to have multiple threads reach out to other servers to fetch data and return the data back to the caller.

The most basic example would be an application that reads data from 20 Web pages, and when complete returns the HTML back to a single thread that creates some sort of aggregated result based on the content of all the pages. Unless you create one thread for each of your retrieval methods, this code will be much slower than desired—99 percent of all execution time would likely be spent waiting for the HTTP request to return.

Running this code on a single thread is inefficient, and the syntax for creating a thread is difficult to get right. The challenge compounds as you support multiple threads and their attendant

Figure 1 **Encapsulating ReaderWriterLockSlim**

```
public class Synchronizer<TImpl, TIRead, TIWrite> where TImpl : TIWrite, TIRead
{
  ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();
  TImpl _shared;

  public Synchronizer(TImpl shared)
  {
    _shared = shared;
  }

  public void Read(Action<TIRead> functor)
  {
    _lock.EnterReadLock();
    try {
      functor(_shared);
    } finally {
      _lock.ExitReadLock();
    }
  }

  public void Write(Action<TIWrite> functor)
  {
    _lock.EnterWriteLock();
    try {
      functor(_shared);
    } finally {
      _lock.ExitWriteLock();
    }
  }
}
```

Figure 2 **Using the Synchronizer Class**

```
interface IReadFromShared
{
  string GetValue();
}

interface IWriteToShared
{
  void SetValue(string value);
}

class MySharedClass : IReadFromShared, IWriteToShared
{
  string _foo;

  public string GetValue()
  {
    return _foo;
  }

  public void SetValue(string value)
  {
    _foo = value;
  }
}

void Foo(Synchronizer<MySharedClass, IReadFromShared, IWriteToShared> sync)
{
  sync.Write(x => {
    x.SetValue("new value");
  });
  sync.Read(x => {
    Console.WriteLine(x.GetValue());
  })
}
```

objects, forcing developers to repeat themselves as they write the code. Once you realize that you can create a collection of delegates, and a class to wrap them, you can then create all your threads with a single method invocation. Just like that, creating threads becomes much less painful.

In **Figure 3** you'll find a piece of code that creates two such lambdas that run in parallel. Notice that this code is actually from the unit tests of my first release of the Lizzie scripting language, which you can find at bit.ly/2FfH5y8.

> The more classes you add to a hierarchy, the less elegant it becomes, until it collapses under its own weight.

If you look carefully at this code, you'll notice that the result of the evaluation doesn't assume that any one of my lambdas is being executed before the other. The order of execution is not explicitly specified, and these lambdas are being executed on separate threads. This is because the Actions class in **Figure 3** lets you add delegates to it, so you can later decide if you want to execute the delegates in parallel or sequentially.

To do this, you must create a bunch of lambdas and execute them using your preferred mechanism. You can see the previously mentioned Synchronizer class in **Figure 3**, synchronizing access to the shared string resource. However, it uses a new method on the Synchronizer, called Assign, which I didn't include in the listing in **Figure 1** for my Synchronizer class. The Assign method uses the same "lambda trickery" that I described in the Write and Read methods earlier.

If you'd like to study the implementation of the Actions class, note that it's important to download version 0.1 of Lizzie, as I completely rewrote the code to become a standalone programming language in later versions.

Functional Programming in C#

Most developers tend to think of C# as being nearly synonymous with, or least closely related to, object-oriented programming (OOP)—and obviously it is. However, by rethinking how you consume C#, and by diving into its functional aspects, it becomes much easier to solve some problems. OOP in its current form is simply not very reuse friendly, and a lot of the reason for this is that it's strongly typed.

Figure 3 **Creating Lambdas**

```
public void ExecuteParallel_1()
{
  var sync = new Synchronizer<string, string, string>("initial_");

  var actions = new Actions();
  actions.Add(() => sync.Assign((res) => res + "foo"));
  actions.Add(() => sync.Assign((res) => res + "bar"));

  actions.ExecuteParallel();

  string result = null;
  sync.Read(delegate (string val) { result = val; });
  Assert.AreEqual(true, "initial_foobar" == result || result == "initial_barfoo");
}
```

For instance, reusing a single class forces you to reuse every single class that the initial class references—both those used through composition and through inheritance. In addition, class reuse forces you to reuse all classes that these third-party classes reference, and so on. And if these classes are implemented in different assemblies, you must include a whole range of assemblies simply to gain access to a single method on a single type.

I once read an analogy that illustrates this problem: "You want a banana, but you end up with a gorilla, holding a banana, and the rainforest where the gorilla lives." Compare this situation with reuse in a more dynamic language, such as JavaScript, which doesn't care about your type, as long as it implements the functions your functions are themselves consuming. A slightly more loosely typed approach yields code that is both more flexible and more easily reused. Delegates allow you to do that.

You can work with C# in a way that improves reuse of code across multiple projects. You just have to realize that a function or a delegate can also be an object, and that you can manipulate collections of these objects in a weakly typed manner.

The ideas around delegates present in this article build on those articulated in an earlier article I wrote, "Create Your Own Script Language with Symbolic Delegates," in the November 2018 issue of *MSDN Magazine* (msdn.com/magazine/mt830373). This article also introduced Lizzie, my homebrew scripting language that owes its existence to this delegate-centric mindset. If I had created Lizzie using OOP rules, my opinion is that it would probably be at least an order of magnitude larger in size.

Of course, OOP and strongly typing is in such a dominant position today that it's virtually impossible to find a job description that doesn't mention it as its primary required skill. For the record, I've created OOP code for more than 25 years, so I've been as guilty as anyone of a strongly typed bias. Today, however, I'm more pragmatic in my approach to coding, and less interested in how my class hierarchy ends up looking.

It's not that I don't appreciate a beautiful class hierarchy, but there are diminishing returns. The more classes you add to a hierarchy, the less elegant it becomes, until it collapses under its own weight. Sometimes, the superior design has few methods, fewer classes and mostly loosely coupled functions, allowing the code to be easily extended, without having to "bring in the gorilla and the rainforest."

I return to the recurring theme of this article, inspired by Miles Davis' approach to music, where less is more and "silence is more important than sound." Code is like that, too. The magic often lives between the lines, and the best solutions can be measured more by what you don't code, rather than what you do. Any idiot can blow a trumpet and make noise, but few can create music from it. And fewer still can make magic, the way Miles did. ■

Thomas Hansen *works in the FinTech and ForEx industry as a software developer and lives in Cyprus.*

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

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

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MARCH 3-8, 2019 ➤ BALLY'S HOTEL AND CASINO

AN OASIS OF EDUCATION DAZZLING IN THE DESERT

Las Vegas

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

Save $300 When You Register by February 8!

Use Promo Code MSDN

Your Adventure STARTS HERE!

vslive.com/lasvegas

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360

AGENDA AT-A-GLANCE

Visual Studio LIVE! EXPERT SOLUTIONS FOR .NET DEVELOPERS — LAS VEGAS

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				
		Full Day Hands-On Labs: Sunday, March 3, 2019 *(Separate entry fee required)*			
8:00 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries			
9:00 AM	6:00 PM	**HOL01** Full Day Hands-On Lab: Modern Security Architecture for ASP.NET Core (Part 1) - *Brock Allen*	**HOL02** Full Day Hands-On Lab: Xamarin and Azure: Build the Mobile Apps of Tomorrow - *Laurent Bugnion & Brandon Minnick*	**HOL03** Full Day Hands-On Lab: Develop An ASP.NET Core 2.X and EF Core 2.X App In a Day - *Philip Japikse*	
4:00 PM	6:00 PM	Conference Registration Open			
START TIME	**END TIME**	**Pre-Conference Workshops: Monday, March 4, 2019** *(Separate entry fee required)*			
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries			
9:00 AM	06:00 PM	**HOL01** Full Day Hands-On Lab: Modern Security Architecture for ASP.NET Core (Part 2) - *Brock Allen*	**M02** Workshop: ASP.NET Core with Azure DevOps - *Brian Randell*	**M03** Workshop: Cross-Platform C# Using .NET Core and WebAssembly - *Rockford Lhotka & Jason Bock*	
7:00 PM	09:00 PM	Dine-A-Round - Carlos & Charlies in the Flamingo, meet at conference registration desk at 6:45pm to walk over with the group.			
START TIME	**END TIME**	**Day 1: Tuesday, March 5, 2019**			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**T01** Take Advantage of HTML5 Features - *Paul Sheriff*	**T02** Azure 101 - *Laurent Bugnion*	**T03** Visual Studio Productivity Tips and Tricks - *Allison Buchholtz-Au*	**T04** How Microsoft Does DevOps - *Tiago Pascoal*
9:30 AM	10:45 AM	**T05** Cool Features in CSS 3 - *Paul Sheriff*	**T06** Microservices with ACS (Managed Kubernetes) - *Vishwas Lele*	**T07** C# 7.x Like a Boss! - *Adam Tuliper*	**T08** DevSlop: DevSecOps with Azure DevOps Pipelines - *Tanya Janca*
11:00 AM	12:00 PM	Keynote: To Be Announced			
12:00 PM	1:00 PM	Lunch - *Platinum*			
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - *Grand Salon*			
1:30 PM	2:45 PM	**T09** Xamarin.Forms Takes You Places! - *Sam Basu*	**T10** Busy Developer's Guide to NoSQL - *Ted Neward*	**T11** Improving Performance in .NET Applications - *Jason Bock*	**T12** To Be Announced
3:00 PM	4:15 PM	**T13** Bringing History to Life with the HoloLens - *Adam Tuliper*	**T14** Busy .NET Developer's Guide to Python - *Ted Neward*	**T15** Concurrent Programming in .NET - *Jason Bock*	**T16** SQL Server Database DevOps - *Brian Randell*
4:15 PM	5:30 PM	Welcome Reception - Grand Salon			
START TIME	**END TIME**	**Day 2: Wednesday, March 6, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**W01** Choosing a Front-End JavaScript Framework - *Ben Hoelting*	**W02** The Next Frontier - Conversational Bots - *Sam Basu*	**W03** Introduction to Windows Containers and Docker - *Marcel de Vries*	**W04** The Visible Developer: Why You Shouldn't Blend In - *Heather Downing*
9:30 AM	10:45 AM	**W05** JavaScript Patterns for the C# Developer - *Ben Hoelting*	**W06** Getting Started with Artificial Intelligence - *Jen Stirrup*	**W07** Agile Failures: Stories From The Trenches - *Philip Japikse*	**W08** DevOps on Azure with Containers, K8s, and Azure DevOps - *Brian Randell*
11:00 AM	12:00 PM	General Session: .NET Today and Tomorrow - *Brady Gaster, Senior Program Manager & Beth Massi, Product Marketing Manager, .NET Team, Microsoft*			
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch - *Platinum*			
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:10pm (Must be present to win) - *Grand Salon*			
1:30 PM	2:45 PM	**W09** Securing Web APIs from JavaScript/SPA Applications - *Brock Allen*	**W10** Taking Advantage of AI easily with Azure Cognitive Services - *Laurent Bugnion*	**W11** Go Serverless with Azure Functions - *Eric D. Boyd*	**W12** .NET Development Using Azure Dev Spaces - *Lisa Guthrie*
3:00 PM	4:15 PM	**W13** What's New in ASP.NET Core 2.2 to Secure Web Applications and APIs - *Brock Allen*	**W14** Applying ML to Software Development - *Vishwas Lele*	**W15** Secure Your App with Azure AD B2C - *Oren Novotny*	**W16** Bolting Security Into Your Development Process - *Tiago Pascoal*
4:30 PM	5:45 PM	**W17** Cross-Platform C# Using .NET Core and WebAssembly - *Rockford Lhotka*	**W18** Solving the 80% Problem. Data Preparation for Microsoft AI - *Euan Garden*	**W19** Monitor Your Applications and Infrastructure - *Eric D. Boyd*	**W20** Signing Your Code The Easy Way - *Oren Novotny*
7:00 PM	8:30 PM	Evening Out TBD			
START TIME	**END TIME**	**Day 3: Thursday, March 7, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**TH01** Science of Great UI - Part 1 (Efficiency in Thought & Motion) - *Mark Miller*	**TH02** Data Visualization Principles for Artificial Intelligence in Business - *Jen Stirrup*	**TH03** Building Resilient Microservices with .NET on Azure Service Fabric - *Mark Fussell*	**TH04** C# 7, Roslyn and You - *Jim Wooley*
9:30 AM	10:45 AM	**TH05** Science of Great UI - Part 2 (Design Like a Pro) - *Mark Miller*	**TH06** Understanding, Selecting, Visualizing and Finalizing Data Science Models for Data Professionals - *Jen Stirrup*	**TH07** Microservice Architecture in .NET - *Rockford Lhotka*	**TH08** (WPF + WinForms) * .NET Core = Modern Desktop - *Oren Novotny*
11:00 AM	12:15 PM	**TH09** Building Cross Device Experiences with Project Rome - *Tony Champion*	**TH10** Introduction to the CNTK Neural Network Library - *James McCaffrey*	**TH11** How to Observe and Talk to Users - *Billy Hollis*	**TH12** Entity Framework Core Performance Monitoring and Tuning - *Jim Wooley*
12:15 PM	1:15 PM	Lunch - Platinium			
1:15 PM	2:30 PM	**TH13** Building UWP Apps for Multiple Devices - *Tony Champion*	**TH14** Modern SQL Server Security Features for Developers - *Leonard Lobel*	**TH15** The Most Important Lessons I've Learned in Forty Years of Developing Software - *Billy Hollis*	**TH16** Architecting Systems for DevOps and Continuous Delivery - *Marcel de Vries*
2:45 PM	4:00 PM	**TH17** Google vs. Alexa: Battle of the Bots - *Heather Downing*	**TH18** Introduction to Cosmos DB - *Leonard Lobel*	**TH19** SignalR: Real-time for All the Things - *Brady Gaster*	**TH20** Security in Your Pipelines. The shift to Rugged DevOps - *René van Osnabrugge*
START TIME	**END TIME**	**Post-Conference Workshops: Friday, March 8, 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: Building, Running, & Continuously Deploying Microservices with Docker Containers on Azure - *Marcel de Vries and René van Osnabrugge*		**F02** Workshop: Developer Dive into SQL Server - *Leonard Lobel*	

Speakers and sessions subject to change

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

Machine Learning on the Edge: Azure ML and Azure IoT Edge Integration

Ted Way and Emmanuel Bertrand

The edge is defined as any compute platform that isn't the cloud. It could range from Azure Stack (a cluster of machines that serves as mini-Azure), to a powerful gateway server, to a smaller device such as a Raspberry Pi or sensor. Even refrigerators and other appliances have enough processing power nowadays to be considered edge devices. The challenge is how to apply artificial intelligence (AI) and machine learning (ML) to data that can't make it to the cloud, whether that's due to data sovereignty, privacy, bandwidth or other issues.

James McCaffrey had a great article—"Machine Learning with IoT Devices on the Edge" (msdn.com/magazine/mt847186), in the July 2018 issue of *MSDN Magazine* on training ML models and using the Microsoft Embedded Learning Library (ELL) to deploy an optimized model to small edge devices. In this article, our focus is on enterprise tools and services to train your model, manage it and containerize it with Azure Machine Learning. With Azure IoT Edge, you'll learn how to configure a data pipeline utilizing the model and deploying it to the edge device. We'll walk through an example where the Custom Vision service can be used to train a model that's deployed with Azure IoT Edge. Iterations of the model can be registered with Azure ML.

This article discusses:

- Training an image classification model with CustomVision.ai
- Registering the model or train your own custom model with Azure Machine Learning
- Managing and deploying ML models to edge devices with Azure IoT Edge

Technologies discussed:

CustomVision.ai, Azure Machine Learning, Azure IoT Edge, Visual Studio Code

Snow Leopard Scenario

The Snow Leopard Trust (snowleopard.org) is dedicated to "better understand the endangered snow leopard, and to protect the cat in partnership with communities that share its habitat." It does this by setting up cameras across 200 square miles to track which snow leopards are in the area. The cameras are deployed and then data is collected from the camera after a set time. These cameras are motion-activated, which can produce some surprises—such as the time years ago when researchers developed a roll of film only to find photos of a tree swaying in the wind.

With digital cameras, more images can be captured, but even then, only 5 percent of the images contain snow leopards. It could take a human researcher more than 200 hours to sift through the pictures to identify those containing snow leopards. A far better use of a researcher's time would be to train an ML algorithm to identify snow leopards.

ML algorithms are said to "learn," rather than be explicitly programmed. For example, if you were creating an app to predict the

price of a house based on factors such as the number of bedrooms, square footage and other factors, you could explicitly program it by adding rules or looking it up in some database that already has the answer. If the user enters the number of bedrooms between two and four, and the house had a certain square footage, you would return a certain price.

Instead of programming that response, you could use a very simple equation, such as:

$$y = w1*x1 + w2*x2 + b$$

In this example, x1 is the number of bedrooms and x2 is the square footage. Based on your data where you have the last sale price of a number of houses, you can "train" the model by finding w1, w2 and b that fits the data best. After training, the ML model is static, in that if you give it the same input twice, it will respond with the same output each time. However, you could continue to evolve the algorithm by training it on new data and updating the algorithm, which is essentially finding new values for w1, w2 and b.

Deep learning is a subset of ML. Instead of a simple linear equation like the one just described, we use artificial neural networks or deep neural networks (DNNs) that consist of many neurons or nodes across many layers. The goal is to mimic the way our brains work. Each neuron performs a calculation based on data, such as parts of images, and if the result meets a certain threshold, the neuron is activated.

This input is then fed into the next layer. For example, let's say we had a picture of a handwritten number, and we have two neurons. The first neuron looks at the top half of the picture, and the second looks at the bottom half. If the first neuron sees a circle, it activates. If the bottom one sees a vertical line, it activates. For the number eight, the first neuron will activate, but the second one will not. However, for the number nine, the first neuron will activate, and the second one will also activate. You can then take that information and conclude that it's the number nine.

Azure Components

Before we go any further, it's important to be aware of the components of both the Azure and ML stacks. These include:

Custom Vision Service: This service is part of the Cognitive Services platform and lets you easily train an AI model by uploading images and manually tagging them. The model can be exported to many platforms—such as iOS (CoreML), ONNX or TensorFlow—and saved in the .pb format, based on Protocol Buffers, or protobufs. Protobufs are described by Google as its "language-neutral, platform-neutral, extensible mechanism for serializing structured data—think XML, but smaller, faster and simpler" (developers.google.com/protocol-buffers).

Azure Machine Learning: This service is an end-to-end data science platform as a cloud service that you can use to develop and deploy ML models. Using the Azure Machine Learning service, you can track your models as you build, train, deploy and manage them, all at the broad scale that the cloud provides.

Azure IoT Hub and IoT Edge: Azure IoT Hub is used to connect, monitor and control billions of Internet of Things (IoT) assets. Azure IoT Edge is a platform that's remotely managed from an IoT Hub. It enables analysis of data on devices instead of in the cloud by moving parts of your workload to the edge. Your devices can spend less time sending messages to the cloud, and they can react autonomously and quickly to changes.

Using these components the Snow Leopard Trust can deploy 30 motion-activated cameras, each attached to relatively inexpensive PCs. We'll use Azure IoT Edge to deploy the snow leopard model to a PC that simulates one of the devices.

Image Classification on the Edge

Now let's walk through the steps involved in provisioning the required Azure resources and for building the solution. We'll provide more details about what's provisioned and what's happening in the code in the following section. To get started you'll need:

- An Azure subscription. Don't have one? Create a free account at azure.microsoft.com/free.
- Visual Studio Code (code.visualstudio.com).
- Ubuntu 16.04 Linux x64 VM to serve as your IoT Edge device.
- Azure resources, including an Azure Machine Learning workspace, an Azure IoT Hub and a CustomVision.ai account.
- Model deployment sample at aka.ms/visedge.

Creating an Azure ML workspace also creates a storage account and Azure Container Registry, which you can use to store your Docker images for deployment to IoT Edge devices. Start by downloading and installing VS Code Tools for AI from (aka.ms/vscodetoolsforai). After installation, click the Azure symbol in the VS Code activity bar on the left.

Now install the Azure ML SDK by opening the command palette using Ctrl+Shift+P and typing "Install Azure ML SDK." In the integrated terminal window, specify the Python interpreter to use or you can hit Enter to use your default Python interpreter.

Next, create an Azure ML workspace by clicking the Azure icon in the Visual Studio Code activity bar. The Azure: Machine Learning sidebar appears and then you right-click your Azure subscription and select Create Workspace. In the list that appears, select an existing resource group from the list, or create a new one using the wizard in the command palette. In this example, we're using "iot-camera-group."

Now in the field, type a unique and clear name for your new workspace. Press enter and the new workspace will be created. It appears in the tree below the subscription name.

Let's create an IoT Hub by opening VS Code and clicking on the Extensions button on the left pane (Ctrl+Shift+X). Search for "Azure IoT Toolkit" and install the extension. You'll need a Linux PC to act as your edge device. You can use a VM on your PC, on Azure or anywhere else you have access.

Start by clicking the Explorer button on the activities bar on the left and find the "Azure IoT Hub Devices" entry. Click "More actions" (the three dots) and select "Create IoT Hub."

Now choose a subscription, then choose the resource group ("iot-camera-group") from earlier or create a new one. Next, choose a region and choose a tier. You may use the free F1 tier, but if an F1 tier IoT Hub already exists, then you'll need to create an S1 tier. Finally, give the IoT Hub a name, such as "iot-camera-hub."

It's time to register an IoT Edge device by adding an entry to your IoT Hub. This will generate a connection string that you'll enter in your IoT Edge device (in this case, your Linux VM) so it can connect to your IoT Hub.

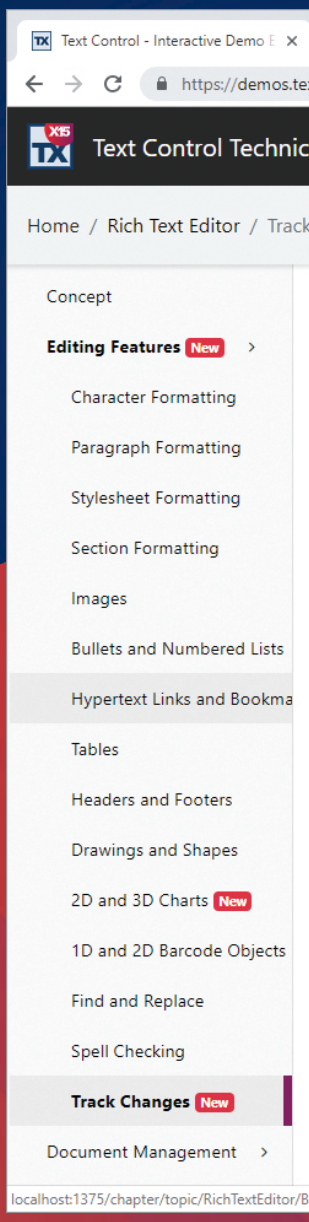
TEXT CONTROL

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

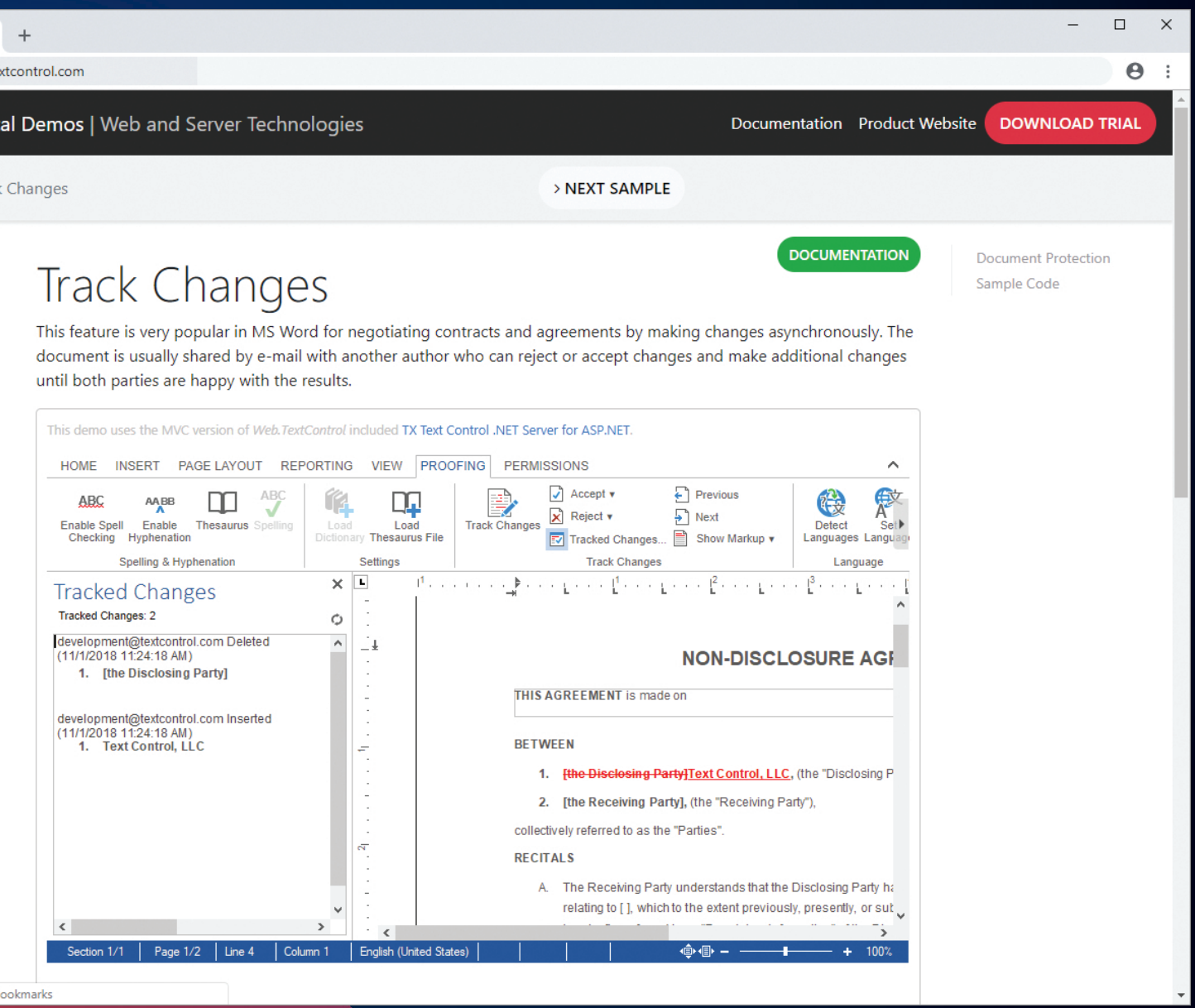


WE ARE CHANGING THE WAY YOU LOOK AT REPORTING

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!



To do this, first click the "More actions" menu (the three dots) on "Azure IoT Hub Devices" and choose "Create IoT Edge device." Don't click "Create Device." Then enter a name for the edge device, such as camera-pc-01. Finally, copy the connectionString value to the entry you just created. You'll need to enter this in your edge device.

In the future, you can click "Azure IoT Hub Devices" and then "Select IoT Hub." You'll have to sign into Azure, choose the subscription you provisioned the IoT Hub in, and then select your IoT Hub, but you can view all the connection strings associated with your devices.

Set up IoT Edge Device

The easiest way to simulate an edge device is to use a preconfigured VM (aka.ms/edgevm). When the VM starts, it installs the latest Azure IoT Edge runtime and dependencies. After you provision the VM, log into it and run this command on the command line with the connection string for the device you registered with IoT Edge (the connection string is the connectionString .value you copied earlier):

```
/etc/iotedge/configedge.sh "<your_iothub_edge_connection_string>"
```

If you want to set up your own Ubuntu 16.04 x64 VM as the edge device, check out this information: aka.ms/edgevm.

Image Classification Solution

Now it's time to build the image classification solution. Clone the Git sample (aka.ms/visedge). We'll use the Linux x64 PC as the edge machine. Follow these steps:

- Update the .env file with the values for your container registry and make sure that your Docker engine has access to it.
- Rename the deployment.template.json file to deployment.template.RPI.json.
- Rename the deployment.template.test-amd64.json file to deployment.template.json.

Figure 1 **Prediction Code**

```
from urllib.request import urlopen

import tensorflow as tf

from PIL import Image
import numpy as np
# import scipy
# from scipy import misc
import sys
import os

filename = 'model.pb'
labels_filename = 'labels.txt'

network_input_size = 227
mean_values_b_g_r = (0,0,0)

size = (256, 256)
output_layer = 'loss:0'
input_node = 'Placeholder:0'

graph_def = tf.GraphDef()
labels = []

def initialize():
  print('Loading model...',end=''),
  with tf.gfile.FastGFile(filename, 'rb') as f:
    graph_def.ParseFromString(f.read())
    tf.import_graph_def(graph_def, name='')
  print('Success!')
  print('Loading labels...', end='')
  with open(labels_filename, 'rt') as lf:
    for l in lf:
      l = l[:-1]
      labels.append(l)
  print(len(labels), 'found. Success!')

def crop_center(img,cropx,cropy):
  y,x,z = img.shape
  startx = x//2-(cropx//2)
  starty = y//2-(cropy//2)
  print('crop_center: ', x, 'x', y, 'to', cropx, 'x', cropy)
  return img[starty:starty+cropy,startx:startx+cropx]

def predict_url(imageUrl):
  print('Predicting from url: ',imageUrl)
  with urlopen(imageUrl) as testImage:
    # image = scipy.misc.imread(testImage)
    image = Image.open(testImage)
    return predict_image(image)

def predict_image(image):
  print('Predicting image')
  tf.reset_default_graph()
  tf.import_graph_def(graph_def, name='')

with tf.Session() as sess:
  prob_tensor = sess.graph.get_tensor_by_name(output_layer)

  # w = image.shape[0]
  # h = image.shape[1]
  w, h = image.size
  print('Image size',w,'x',h)

  # scaling
  if w > h:
    new_size = (int((float(size[1]) / h) * w), size[1], 3)
  else:
    new_size = (size[0], int((float(size[0]) / w) * h), 3)

  # resize
  if  not (new_size[0] == w and new_size[0] == h):
    print('Resizing to', new_size[0],'x',new_size[1])
    #augmented_image = scipy.misc.imresize(image, new_size)
    augmented_image = np.asarray(image.resize((new_size[0], new_size[1])))
  else:
    augmented_image = np.asarray(image)

  # crop center
  try:
    augmented_image = crop_center(augmented_image, network_input_size,
      network_input_size)
  except:
    return 'error: crop_center'

  augmented_image = augmented_image.astype(float)

  # RGB -> BGR
  red, green, blue = tf.split(axis=2, num_or_size_splits=3, value=augmented_image)

  image_normalized = tf.concat(axis=2, values=[
    blue - mean_values_b_g_r[0],
    green - mean_values_b_g_r[1],
    red - mean_values_b_g_r[2],
  ])

  image_normalized = image_normalized.eval()
  image_normalized = np.expand_dims(image_normalized, axis=0)

  predictions, = sess.run(prob_tensor, {input_node: image_normalized})

  result = []
  idx = 0
  for p in predictions:
    truncated_probablity = np.float64(round(p,8))
    if (truncated_probablity > 1e-8):
      result.append({'Tag': labels[idx], 'Probability': truncated_probablity })
    idx += 1
  print('Results: ', str(result))
  return result
```

- Build the entire solution by right-clicking the file deployment.template.json and selecting Build IoT Edge Solution (this can take a while, especially to build numpy and pillow).
- Deploy the solution to your device by right-clicking the config/deployment.json file, selecting Create Deployment for IoT Edge device and choosing your targeted device.
- Monitor the messages being sent to the Cloud by right-clicking your device from the VS Code IoT Edge Extension and selecting Start Monitoring D2C Message.

For the x64 version of the solution, these steps deploy two modules:

- **Camera Capture:** This captures the video stream from a USB camera and sends the frames to the image classification module. The result is sent to edgeHub, which can relay it as a message to IoT Hub.
- **Custom Vision:** This Web service runs locally over HTTP and takes in an image and classifies it. It was built using the Custom Vision (customvision.ai) Web site. The model in the solution classifies whether an image contains an apple or a banana.

The Camera Capture module can send messages to the edgeHub using the Azure IoT Edge SDK. You can build your own modules as Docker containers and use the SDK to communicate with edgeHub.

For the Custom Vision Service you can work with the existing model in the solution, or you can go to customvision.ai and train your own model. To create your snow leopard classifier, upload pictures of snow leopards to the Custom Vision Service. You can also upload pictures of other animals you may find in the snow leopard habitat. Just tag the images and click the Train button to train the model. The model and label files are exported as .pb and .txt files, respectively, which you can use to replace what's in the solution.

Custom Vision uses transfer learning, which takes an existing deep neural network and extracts features from your image. Then it trains a classifier on your data (in this case, the images you uploaded and the tags you assigned to them).

To illustrate transfer learning, imagine you want to train a bomb-sniffing dog—a German Shepherd, for example. If you take a 2-year-old German Shepherd, it won't know anything about bombs. It does have the "infrastructure" to detect bombs, with its powerful sense of smell. You can start training your German Shepherd by having it sniff a bomb and telling it that this is a bomb, then give it another smell, such as that of a flower, and say that this is not a bomb. You repeat this for various types of bombs and other smells, and after a few months and lots of kibble, you have yourself a bomb-sniffing dog. Training the German Shepherd isn't that difficult. Making a German Shepherd is the hard part.

DNNs such as ResNet 50, Inception v3, YOLO and VGG are essentially German Shepherds. They're complex AI models that take in an image and extract features from it, much the way a German Shepherd extracts signals from a smell. Just like the number of bedrooms and the square footage of a house are features that can be used to predict a price, the features extracted are sent to a classifier. This classifier is what you need to train, which will then do the prediction of whether a snow leopard is in the image or not. The classifier can be a simple logistic regression, support vector machine or any other classifier. The beauty in transfer learning is the feature extraction with the DNN, which results in useful features that can be used to classify images very accurately.

Figure 1 provides the code to take an image and predict what's in it using the AI model.

Azure Machine Learning

You can use Custom Vision to create your AI model, but what if you want to train multiple models or collaborate with a team of data scientists? The rigors of software development, especially around source control, are missing in the data science workflow. Azure ML brings this all together with data preparation, experimentation, model management and operationalization.

For this example, if you're doing various experiments with Custom Vision, you can register the .pb model and other files so you can keep track of them. Here's that code:

```
from azureml.core.model import Model
model = Model.register(model_path = "custom-vision-model/",
                       model_name = "model.pb",
                       tags = {"area": "vision", "type": "classification",
                         "version": "1.0"},
                       description = "Snow leopard classification model",
                       workspace = ws)
```

If you want to see all the models you've registered, enter this code:

```
models = Model.list(workspace=ws)
for m in models:
  print("Name:", m.name,"\tVersion:", m.version, "\tDescription:",
  m.description, m.tags)
```

If you want to train your own models, check out the Azure Machine Learning sample notebooks (github.com/Azure/MachineLearningNotebooks). You can train your model, register it and create Docker images with the model. The Docker images can be deployed to the cloud in an Azure Container Instance or Azure Kubernetes Service. A REST API is exposed in the container to call when scoring data. The Docker image also contains the Azure IoT Edge SDK, which will broker messages with the IoT Edge agent running on the edge device. This lets you create the container and deploy it to the cloud or edge.

Wrapping Up

There are a lot of components when building an enterprise solution, starting with Custom Vision to train the model, Azure ML to manage the model (and to train other types of models) and Azure IoT Edge to deploy the models. Setting up these services will take some ramping up time, but you'll reap the benefits of easily training new models and updating them on multiple edge devices. ■

Ted Way *is a senior program manager on the Azure ML team working to accelerate AI in the cloud and the edge on specialized hardware. His Ph.D. from the University of Michigan was on "spell check for radiologists," a computer-aided diagnosis system for estimating malignancy on thoracic CT scans.*

Emmanuel Bertrand *is a senior program manager on the Azure IoT Edge team, where he helps enable easy deployment of IoT modules through the Azure Marketplace. Bertrand joined Microsoft as a customer and user experience expert, and prior to that had digitized the processes of several industrial companies using traditional tools.*

Thanks to the following technical experts for reviewing this article: *Rakesh Kelkar, Chipalo Street*

Build and Deploy Highly Available and Resilient Solutions on Azure

Srikantan Sankaran

Organizations these days have a global presence and they expect their mission-critical applications to reach users across multiple geographies. IT staff in such organizations are under pressure to deliver on those expectations, and their success depends on how well their applications and platforms are architected and built to tackle the complexities inherent in that goal. Microsoft Azure offers numerous services and features that provide an almost turnkey implementation to address the relevant requirements, whether they involve enabling geo-replication of services and data, latency-sensitive routing of user requests to a service closer to end users, or providing seamless application failover to other regions when disaster strikes. In this article I'll take a look at a few of these objectives and discuss how Azure can be used to implement highly available and resilient global applications.

I'm going to use a three-tier solution consisting of an MVC and Web API application to illustrate the key aspects of building and deploying highly available and resilient applications in Azure. The application is accessed by a user to capture basic information pertaining to his family, which is passed on to the API tier, which in turn persists the data into a geo-replicated Cosmos DB database. The application is deployed Active-Active across multiple regions to ensure resiliency against regional outages. Within a region, it's deployed across Availability Zones to protect against datacenter failures (bit.ly/2zXlRll). The database has a multi-region write feature enabled (bit.ly/2zV6OI0), ensuring that users can write to and read data from a database that's close to their point of access, to minimize latency (bit.ly/2QRr1Il).

Some of the technologies discussed in this article are in preview. All information is subject to change.

This article discusses:

- The solution architecture
- Developing a cross-platform MVC and Web API application using ASP.NET Core
- Deploying the application to Azure Service Fabric
- Using Availability Zones to protect against datacenter failures
- Configuring Azure Front Door Service for global reach and access
- Using Azure DevOps and Jenkins for CI/CD

Technologies discussed:

Microsoft Azure (Cosmos DB, DevOps, Front Door, Service Fabric, Availability Zones, Virtual Network), Docker, Git Bash, Yeoman, Jenkins, ASP.NET Core

Code download available at:

bit.ly/2Lra9Dm

Architecture of the Solution

Figure 1 represents the architecture of the solution. Azure Service Fabric hosts the application components packaged as Docker containers for Linux, provides orchestration capabilities, and ensures

the availability and reliability of the application. There are two zonal Service Fabric clusters deployed to each of the two Azure regions, in South East Asia and East US 2. A zone-redundant Azure Load Balancer is used to round-robin the requests between the two Availability Zones in each region.

Azure Front Door Service (still in public preview) directs user requests to one of the load-balanced endpoints across the two regions, based on the path of least latency. It also provides resiliency in the event of the application or an entire datacenter going down in one of the regions, by directing the requests to the endpoint in the other region.

The Azure Cosmos DB database used in the application is geo-replicated across the two Azure Regions and configured for multi-region write. Thus, data writes originating from the API Service in each region are written to a Cosmos DB collection local to that region.

Azure DevOps is the source code repository for the application. Azure Pipelines is used to build the applications, package them as Docker containers and upload them to a Docker Hub Registry.

The IT policies at the organization hosting this solution forbid administrative access to the Service Fabric cluster over the public Internet. A continuous delivery (CD) pipeline is implemented using Jenkins deployed inside an Azure Virtual Network. It pulls the deployment packages and containers from the Git repository in Azure DevOps and Docker Hub, respectively, and deploys the application to the Service Fabric clusters in the two regions.

The Service Fabric cluster deployed to each region consists of two node types. The primary node type runs the Service Fabric platform services and exposes the management endpoints using an internal load balancer. The secondary node type runs the MVC and Web API applications packaged using Docker containers. A public-facing load balancer routes requests from end users accessing the application over the Internet.

A third Azure Load Balancer (not depicted in the diagram) is used in this architecture to permit outbound calls to the Internet from the cluster, to download the Service Fabric platform components. The internal load balancer configured doesn't have a public-facing endpoint and can't reach the Internet. Without creating this additional public load balancer for outbound connectivity, the Service Fabric cluster can't be set up (bit.ly/2AOVBpp).

> Azure Service Fabric leverages the ability of the underlying Azure Virtual Machine Scale Sets (VMSS) resources for zonal deployments.

Note that the standard SKU of Azure Load Balancer and Public IP address resources alone support deployment to Azure Availability Zones, ruling out the basic SKU, which does not.

Azure Service Fabric leverages the ability of the underlying Azure Virtual Machine Scale Sets (VMSS) resources for zonal deployment.

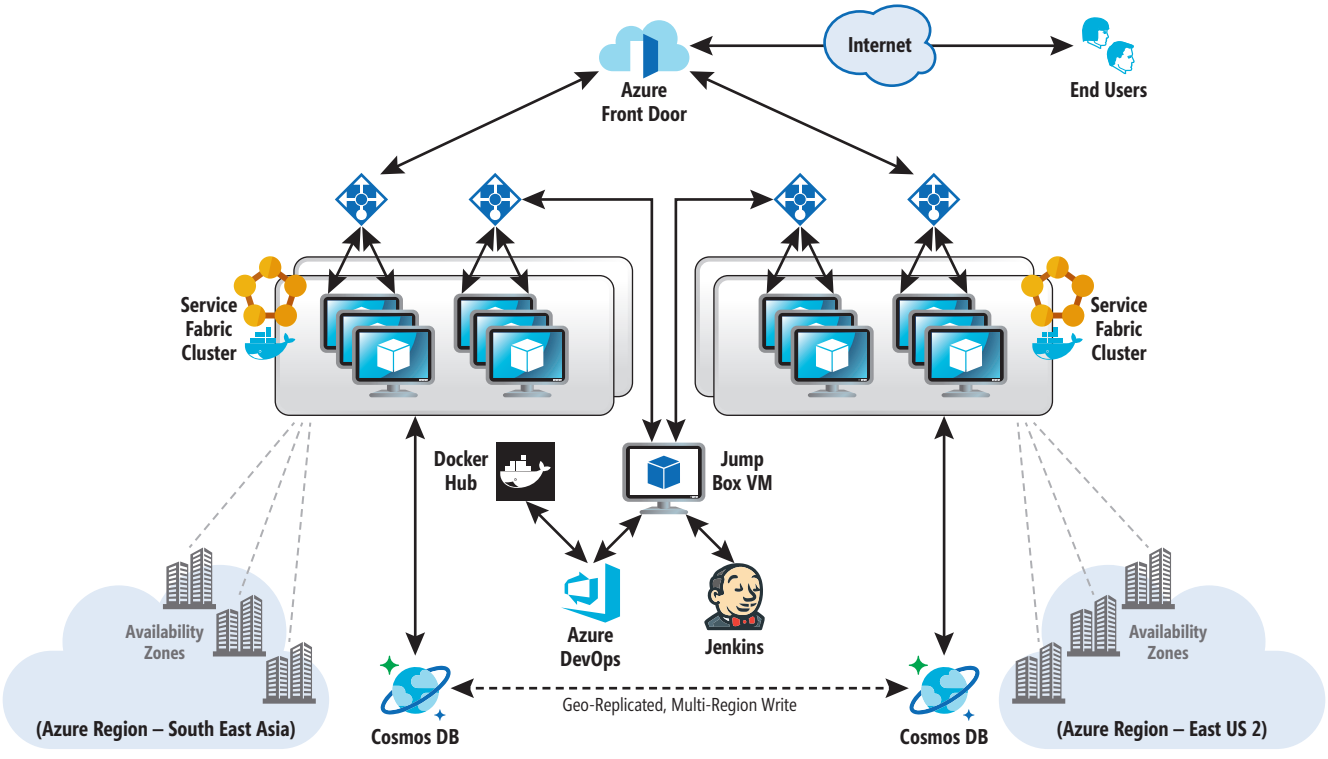


Figure 1 **Solution Architecture**

Developing the Applications

There are two Visual Studio 2017 solution files shared with this article:

- censusapp.sln: The ASP.NET Core 2.0 MVC application.
- censusapi.sln: The ASP.NET Core 2.0 Web API application.

Keep in mind that though this sample application has been built using ASP.NET Core 2.0, it could very well be a Web application created using other technologies, such as Node.js, PHP and so forth. Also, the sample application shared with this article doesn't implement any coding best practices. It's intended only to illustrate select design implementation detail.

The MVC application implements the UI that lets users submit census data and displays that data. It uses the service discovery feature in Service Fabric to call the REST API that persists the data in Azure Cosmos DB. The API project uses the Cosmos DB SDK to implement a prioritized list of database locations to connect to in order to read and write data. The application that's deployed to the South East Asia Region would add this region as "priority 1" and East US 2 as "priority 2." For deployment to the East US 2 Region, it would be the reverse. Hence, there would be two separate container images for the Web API Project, one each for deployment to the respective regions. The container image for the MVC project would be the same, regardless of the Azure Region to which it's deployed.

The following snippet shows how the prioritized list of locations is added to the Cosmos DB connection, and how to configure multi-region writes in the application:

```
ConnectionPolicy policy = new ConnectionPolicy
  {
    ConnectionMode = ConnectionMode.Direct,
    ConnectionProtocol = Protocol.Tcp,
    UseMultipleWriteLocations = true,
  };
policy.PreferredLocations.Add("East US 2");
policy.PreferredLocations.Add("South East Asia");
DocumentClient client = new DocumentClient(new Uri(this.accountEndpoint),
  this.accountKey, policy);
```

The Web API project uses the "Bogus" NuGet package to generate fictitious data.

The Cosmos DB connection string and access keys are stored in the appsettings.json file of the Web API project for simplicity. For production use, they could instead be stored in Azure Key Vault. Refer to my article, "Secure Your Sensitive Business Information with Azure Key Vault" for guidance (msdn.com/magazine/mt845653).

Enabling the Web Applications for Docker Containers Visual Studio 2017 Tools for Docker provide a turnkey implementation to enable an ASP.NET 2.0 Web application for Windows and Linux containers. Selecting the right-click option on the project in Visual Studio 2017 to "enable Docker Support" adds a Docker file to it. Selecting the right-click option on the project to "enable orchestration support" creates a Docker compose file.

The applications in this solution were packaged for Docker Linux containers. Minor edits were done to the Docker compose file to add the port mapping to be used when deploying the application (port 80 on the VMSS nodes to access the MVC application and port 5002 for the Web API).

Continuous Integration (CI) The Visual Studio 2017 solution files are checked into an Azure DevOps Git repository. An Azure pipeline is created that would run the Docker compose files for each of the MVC and Web API projects created in the previous steps. This builds the ASP.NET Core 2.0 applications and creates the Docker container images.

In the next step in the pipeline, a Bash script is used to tag the container images and push them to Docker Hub. This step needs to be performed once for deployment to the Azure Region in South East Asia and once for deployment to Azure Region East US 2. Again, the MVC container is the same no matter to which Azure Region it's deployed. **Figure 2** shows a snapshot of the CI pipeline created to accomplish this step.

Packaging the Applications for Deployment to Service Fabric I used the Yeoman generator for Windows to generate the application and service manifest files and package the solution. You'll find guidance in using this tool at bit.ly/2zZ334n.

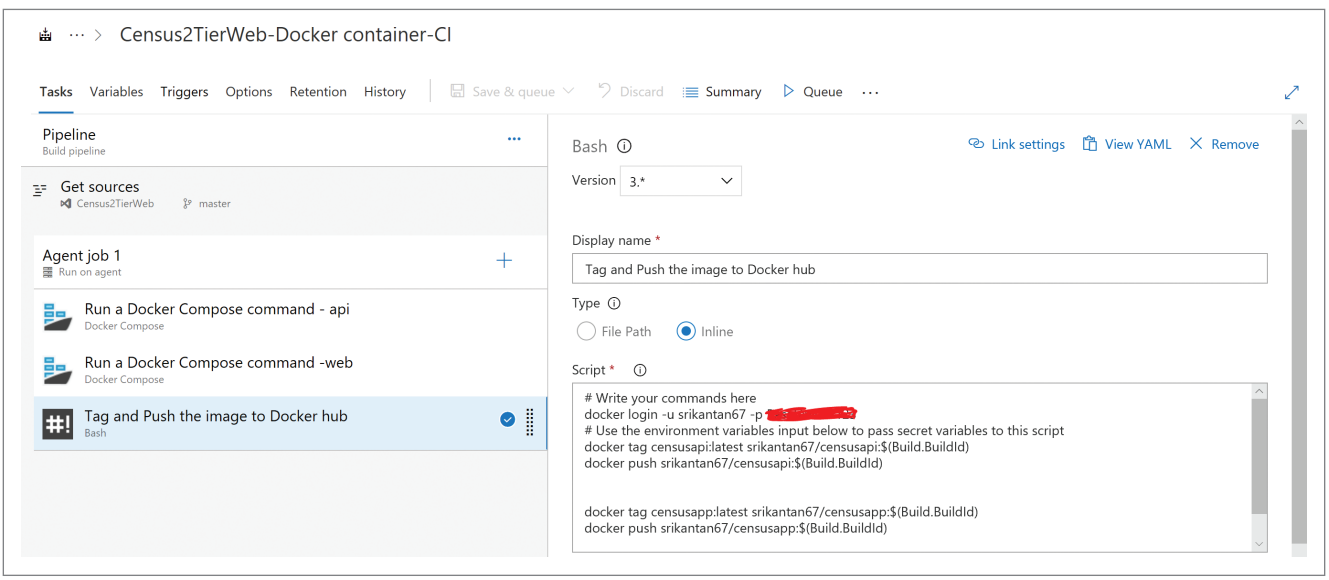


Figure 2 **Azure DevOps Pipeline**

DocuVieware

Universal HTML5 and Document Management Kit

Easy integration

Full support for custom snap-in

Zero-footprint solution

Fully customizable UI

Mobile devices optimization

Fast & crystal-clear rendering

Check the New Features and the Online Demos
60-day Free Trial Support Included at **www.docuvieware.com**

Powered by **GdPicture**.NET

DocuVieware is an ORPALIS imaging technologies product

A single application package is created that bundles the MVC and API apps as Service Types. In the service manifest files, enter the container names uploaded to Docker Hub in the previous steps.

Note that the Web API project's service manifest defines a service DNS name that should match the value in the MVC project's appsettings.json file, to be used in service discovery.

The number of instances of each service type (that is, the Web and API projects) in the manifest is set to two. This instructs Service Fabric to have two instances of the container of that service type always running in the cluster. This number can be modified to suit the deployment, or can be set to auto-scale within a min and max range.

The service manifest supports the notion of a placement constraint. In the following snippet, a placement constraint is defined on the node type name in the Service Fabric cluster to which the service should be deployed. If not specified, the code package could get deployed across all node types, including the primary node type, in the Service Fabric cluster. However, I intend to host only the Service Fabric Platform Services in the primary node type.

```
<ServiceTypes>
  <StatelessServiceType ServiceTypeName="censusapiType"
    UseImplicitHost="true">
<PlacementConstraints>(NodeTypeName==nt-sfazvm0) </PlacementConstraints>
  </StatelessServiceType>
</ServiceTypes>
```

The node type name configured in the Azure Resource Manager (ARM) template (discussed later) to deploy the Service Fabric Cluster should match that in the placement constraint of the service manifest.

Deploying the Applications to Service Fabric

Now let's create the Service Fabric cluster and then deploy the application packages to it.

The first step is to provision a Cosmos DB database, which can be performed from the Azure Portal. Start by creating a database in South East Asia and enable geo-replication to East US 2 Region. Select the "enable Multi-Region Write" option in the wizard.

I retained the Session Consistency configuration, which is the default for the Cosmos DB database, and the default setting that indexes all the properties in the schema. You could select only specific properties to index if you prefer, based on the need to query the data.

To handle any potential conflicts that arise out of enabling the multi-master write, I used the default, "last-writer-wins conflict resolution" policy.

Next, you provision the Service Fabric cluster using an ARM template (bit.ly/2swVkI5 and bit.ly/2rBvFfi). Template SF-Std-ELB-ZonalDeployment.Json deploys two zonal clusters into an existing virtual network. There are certain prerequisites to run this template.

First, use a certificate stored in Azure Key Vault for node-to-node security in the cluster. The thumbprint, Key Vault URL and certificate URL from this step need to be entered in the Parameters section of the ARM template, as shown in **Figure 3**. This must be done for each of the Regions separately, because both the Service Fabric cluster and the Key Vault used by it should reside in the same region.

Figure 3 **Storing the Thumbprint, Azure Key Vault URL and Certificate URL**

```
"certificateThumbprint": {
      "type": "string",
      "defaultValue": "<Certificate Thumb print>
    },
    "sourceVaultValue": {
        "type": "string",
        "defaultValue": "/subscriptions/<SubscriptionId>/resourceGroups/
          lpvmvmssrg/providers/Microsoft.KeyVault/vaults/<vaultname>"
    },
    "certificateUrlValue": {
        "type": "string",
        "defaultValue": "https://<vaultname>.vault.azure.net/secrets/
          soloclustercert/<GUID>
        }
  }
```

Second, a self-signed certificate is generated using Open SSL. The thumbprint of this certificate is entered in the Parameters section of the ARM template:

```
"clientCertificateStoreValue": {
      "type": "string",
      "defaultValue": "<Client certificate thumbprint
    },
```

This certificate is used in Jenkins to connect to the Service Fabric cluster to deploy the application. See bit.ly/2GfLHG0 for more information.

Note that this article uses a self-signed certificate for illustration. For production deployments, you'd use Azure Credentials to connect to the management endpoint securely and to deploy the application.

For more on Service Fabric security, see bit.ly/2CeEzpi and bit.ly/2SR1E73.

The health probes used in the load balancer for the MVC application and Web API need to be configured for HTTP, with the right port numbers and the request path, as shown in **Figure 4**.

Here are the other salient configurations in the ARM template, specific to zonal deployments:

- The standard SKU is chosen for all load balancers used in the cluster.
- The standard SKU is chosen for the Public IP address resource.
- The Availability Zone property needs to be specified. Because this is a zonal deployment, a single zone is specified in a property value. For zonal SF Cluster 1 in South East Asia, this value would be "1" and for zonal SF Cluster 2 in South East Asia, the value would be "2."

Figure 4 **Configuring the Ports and Request Paths**

```
{
    "name": "SFContainerProbe1",
    "properties": {
      "protocol": "Http",
      "port": 5002,
      "requestPath": "/api/family",
      "intervalInSeconds": 5,
      "numberOfProbes": 2
    }
  },
    {
    "name": "SFContainerProbe2",
    "properties": {
      "intervalInSeconds": 5,
      "numberOfProbes": 2,
      "port": 80,
      "requestPath": "/",
      "protocol": " Http "
    }
  }
```

- The property "SinglePlacementGroup" is set to "true."
- To enable service discovery in the cluster, the Service Fabric DNS must be enabled.

Figure 5 shows how these are configured in the ARM template. Verify that the template has deployed successfully and that the services shown in **Figure 5** are visible in the Azure Portal.

Note that Azure Availability Zones are not available in all Azure Regions yet. Also, the feature has reached general availability (GA) in some regions but is only in preview in others.

Now let's connect to the Service Fabric cluster. The Service Fabric management endpoint is deployed behind an internal load balancer. Hence, to access the Service Fabric Explorer to deploy the application, deploy a jump box virtual machine (VM) running Windows Server 2016, either in the same virtual network as the Service Fabric cluster, or to a different virtual network that's peered to that of the cluster.

Copy the certificate for the Service Fabric admin client (the self-signed certificate that was created earlier) to the user certificate store on the jump box VM. When launching the Service Fabric Explorer, select this certificate when prompted.

Next, provision Jenkins to deploy the application. For the scenario in this article, I'll use a Docker container image running Jenkins and the Service Fabric plug-in. This image is deployed to an Ubuntu VM in Azure running in the same virtual network as the Service Fabric cluster.

You'll find more information on the steps required to download, install and configure the container image at bit.ly/2RRRt1Q.

To prepare Jenkins to connect to the Service Fabric cluster, I performed the following additional steps:

- Copy the certificate for the Service Fabric admin client

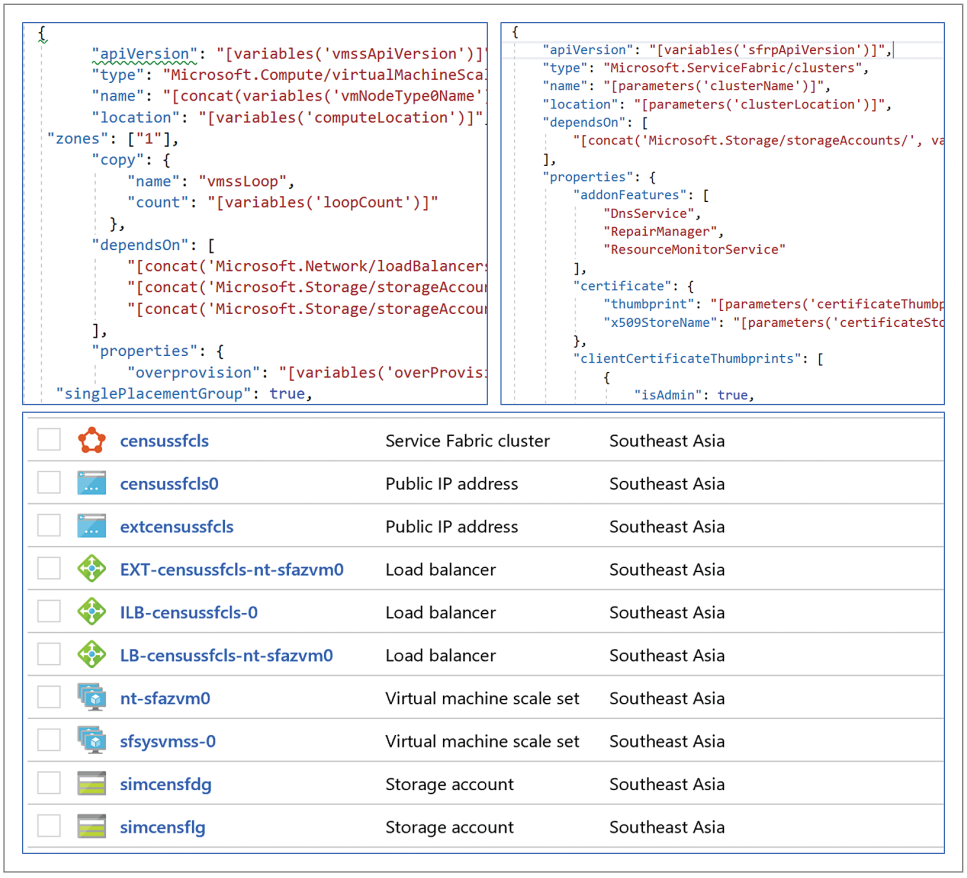


Figure 5 **VMSS Resources Used in the ARM Template for Zonal Deployment**

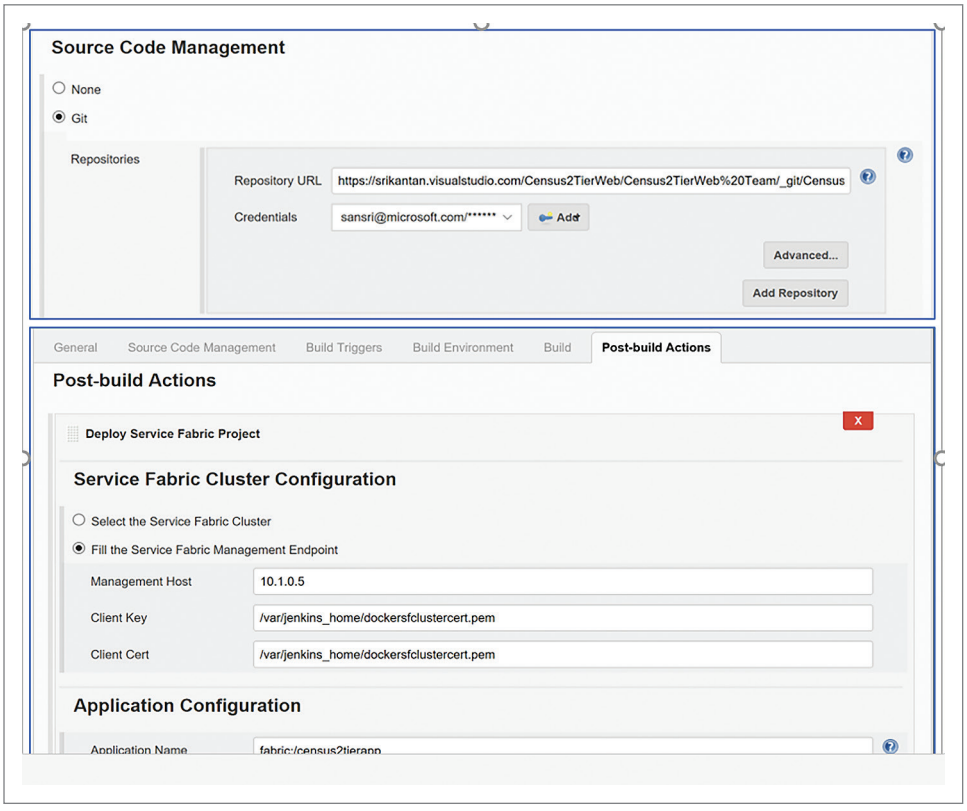


Figure 6 **Service Fabric Plug-in for Jenkins**

Recall that the two container images of the API Service each have a separate region priority when connecting to Cosmos DB, and to implement multi-region writes.

(the self-signed certificate created earlier) to the home directory on the Jenkins VM, using tools like WinSCP.

- Start the Jenkins container on this VM and run the "docker exec" command to copy this certificate to the Jenkins home directory inside the container.

Launch Jenkins and log in to http://<Public-IP-of-Jenkins-VM>:8080 to configure the Service Fabric plug-in.

The steps to configure the Service Fabric plug-in for Jenkins are described at bit.ly/2UBVhWV. The "Post Build Actions" configuration that was used to run the deployment of the application to one of the zonal clusters is shown in **Figure 6**. Add an additional Post Build Action for the second zonal cluster, along the same lines.

Trigger the Build Job option in Jenkins manually, and ensure that the post-trigger action completes successfully. From the jump box VM, launch the Service Fabric Explorer to verify that the application is deployed to both zonal clusters. **Figure 7** shows the Service Fabric Explorer after the application is deployed.

Repeat these steps to deploy to the second Azure Region. Ensure that the service manifest file of the API project is modified so it points to the right container image of the API Service meant for this region. Recall that the two container images of the API Service each have a separate region priority when connecting to Cosmos DB, and to implement multi-region writes.

Running the Application

To bring up the application deployed to Region 1, launch the following URLs:

- http://<Public-IP-Address-LB Region1> starts the MVC Application
- http://<Public-IP-Address-LB Region1>:5002/api/family starts the Web API directly. The MVC project calls the API internally, using Service discovery.

Another set of URLs would be available for Region 2.

Figure 8 shows the application page accessed through the end-

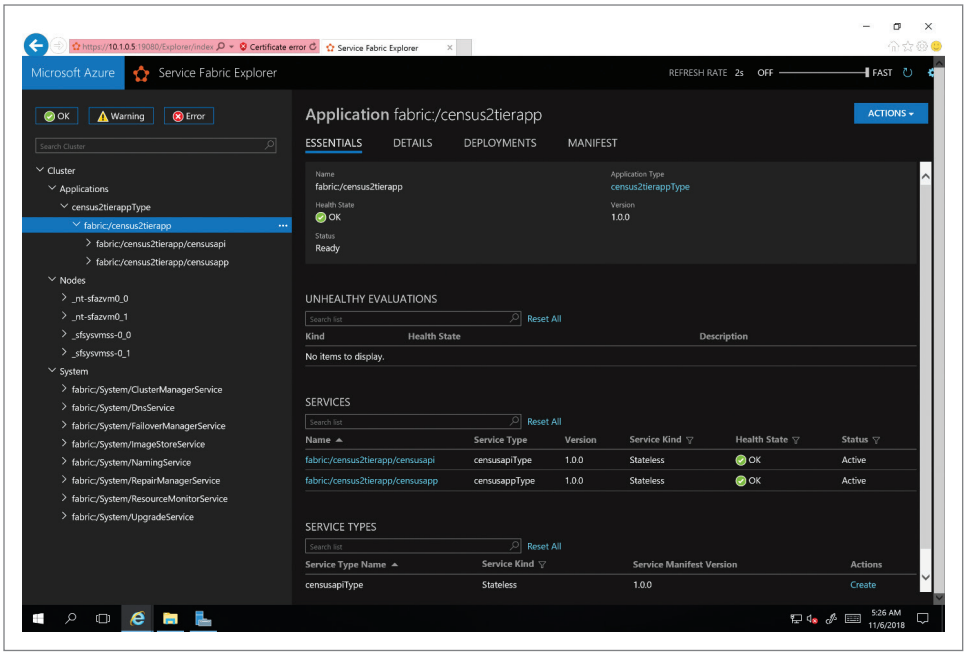


Figure 7 **The Management Endpoint in Service Fabric Explorer**

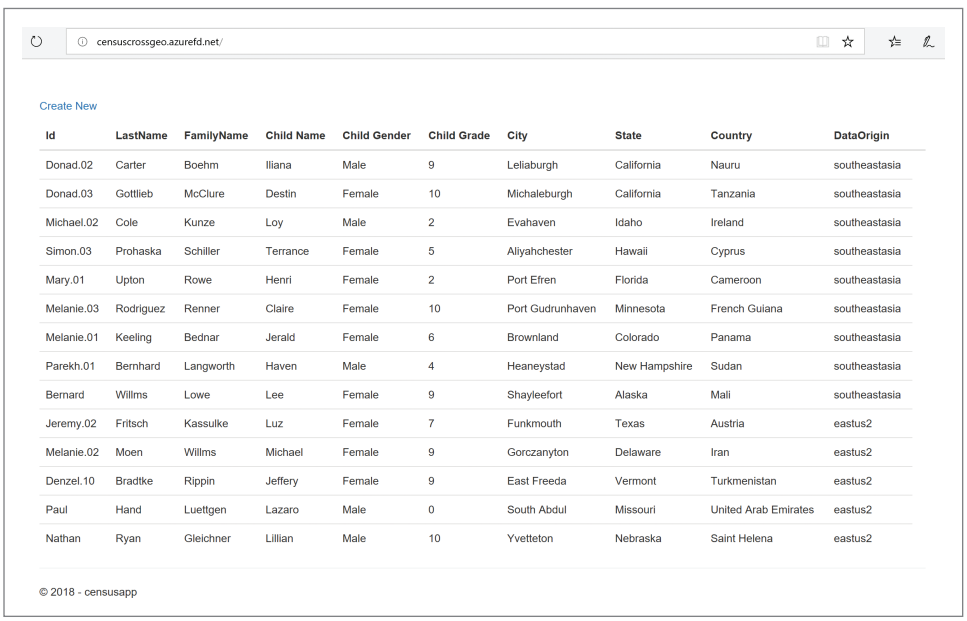


Figure 8 **Sample Census Application**

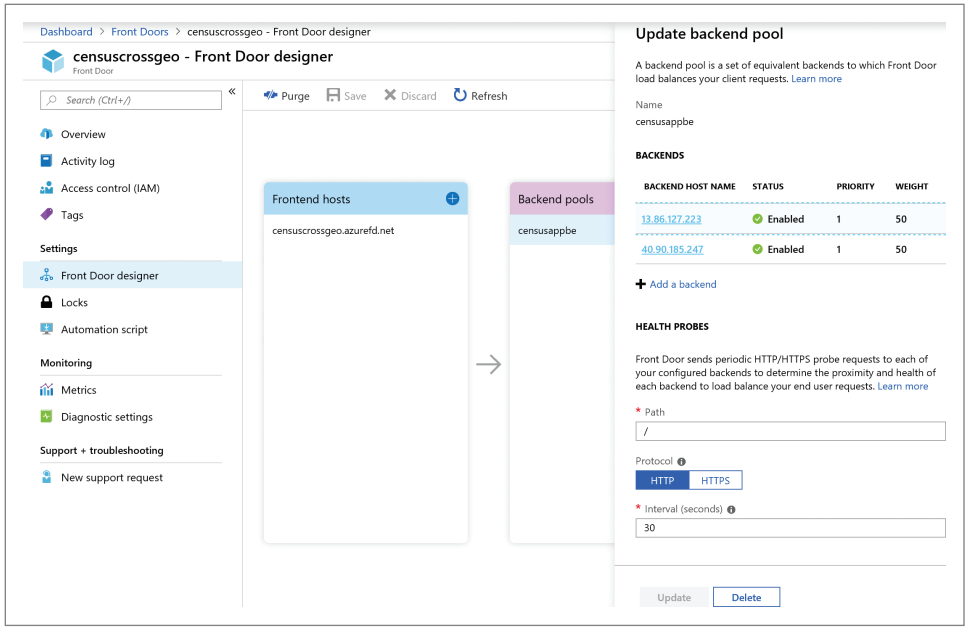


Figure 9 **Azure Front Door Configuration**

points exposed by the Azure Front Door Service. You'll notice a column called DataOrigin that indicates the name of the write region of the Cosmos DB database from which the record was inserted, showcasing the multi-region write capability of Cosmos DB.

Provision Azure Front Door Service

Use the Azure Portal to provision Azure Front Door Service and add the public-facing endpoints of the MVC and Web API applications as back ends to the Front Door Service.

Health probes configured in the Front Door Service ensure that when one of the back-end endpoints isn't accessible, due either to an outage in that region or when the application gets unresponsive, subsequent requests are directed to the other healthy one. The configuration settings in **Figure 9** show how the application endpoints in the two regions have been mapped to a single endpoint exposed by the Front Door Service. While Azure Traffic Manager could have been used as an alternative to Azure Front Door, I chose to implement the latter here as it provides Layer 7 Reverse Proxy, SSL termination and request routing, which are required in such applications.

You'll find more information on creating a Front Door for global Web apps at bit.ly/2QXRkwu.

Software Prerequisites

Here's what you need to implement the sample application:

- Azure Service Fabric CLI (bit.ly/2Cd2zsZ)
- Azure Front Door Service (bit.ly/2QXRkwu)
- Docker for Windows (dockr.ly/2oiWxxH)
- Git Bash for Windows (gitforwindows.org)
- Service Fabric plug-in for Jenkins (bit.ly/2GfLHG0)
- Service Fabric SDK and tools for Visual Studio 2017 (bit.ly/2PDlxMN)
- Yeoman generator for Windows (bit.ly/2zZ334n)

Wrapping Up

In this article I discussed how Azure Service Fabric can be used to package and deploy Docker container-enabled applications, and implement orchestration and service discovery features that are fundamental to a microservices architecture. With the support for Availability Zones in Azure, I deployed 2 Zonal Service Fabric clusters that span multiple datacenters in a region in order to eliminate the effects of datacenter failures.

To increase the reach of the application, I deployed the application to multiple Azure regions, and leveraged the multi-region write support in Azure Cosmos DB, which ensures that not only are the stateless application tiers geo-replicated, but the database also is truly distributed and replicated. Finally, to ensure that users experience the least amount of latency when accessing the application, I implemented a single Azure Front Door Service that routes requests to the right application endpoint. To show how you can implement this architecture with the least amount of disruption to business, and to ensure that security policies are honored, I discussed how CI and CD practices can be implemented using Azure DevOps Service and Jenkins, respectively, and how these can be carried out within the confines of a corporate network.

> To ensure that users experience the least amount of latency when accessing the application, I implemented a single Azure Front Door Service that routes requests to the right application endpoint.

You can download the sample application at bit.ly/2Lra9Dm. See "Software Prerequisites" for information on the software required to implement the sample application. ■

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

Thanks to the following Microsoft technical experts for reviewing this article: *Sudhanva Huruli, Muni Pulipalyam*

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

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! New Orleans, 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 February 22 & Save Up To $400!

Use Promo Code MSDN

YOUR ADVENTURE STARTS HERE

vslive.com/neworleans

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

Protect Your Data with Azure Confidential Computing

Stefano Tempesta

Security is a key driver for accelerating the adoption of cloud computing, but it's also a major concern when you're moving extremely sensitive intellectual property (IP) and data to a public cloud. There are common ways to secure data at rest and in transit, but threats may also occur when data is being processed in memory. Confidential computing adds new data security capabilities to your applications by using trusted execution environments (TEEs) and encryption mechanisms to protect your data while in use. TEEs, also known as enclaves, are hardware or software implementations that safeguard data being processed from access from outside the enclave. An enclave provides a protected container by securing a portion of the processor and memory, as shown in **Figure 1**. Only authorized code is permitted to run and access data, so code and data are protected against viewing and modification from outside of the TEE.

A Matter of Trust

With the recent announcement of the public availability of confidential computing in Azure, Microsoft became the first cloud provider to offer protection for data in use (bit.ly/2OyyxaH and bit.ly/2BxQkpp). The official post on the Azure blog (bit.ly/2I692X1) describes it well: "… Azure confidential computing protects your data while it's in use. It is the final piece to enable data protection through its lifecycle whether at rest, in transit, or in use. It is the cornerstone of our 'Confidential Cloud' vision, which aims to make data and code opaque to the cloud provider."

The concept of "opaque data and code" is revolutionary. For the first time, you can have trust in the cloud that no one, including the cloud provider, can read your data. The data is encrypted at every stage, and only authorized applications have the key to decrypt it and access it. This is accomplished in two ways:

- Hardware: Thanks to a partnership with Intel, Azure can offer hardware-protected virtual machines that run on Intel Software Guard Extensions (SGX) technology. Intel SGX is a set of extensions to the Intel CPU architecture that aims to provide integrity and confidentiality guarantees to sensitive computation performed on a computer, where all the privileged software (kernel, hypervisor and so on) might potentially be compromised.
- Hypervisor: Virtualization-based security (VBS) is a software-based TEE that's implemented by Hyper-V in Windows 10 and Windows Server 2016. Hyper-V prevents administrator code from running on the computer or server, as well as local administrators and cloud service administrators from viewing the contents of the enclave or modifying its execution.

The potential applications for confidential computing are really unlimited. Every time there's a requirement for protecting sensitive data, trusted execution environments represent the building blocks on top of which you can enable new secure business scenarios and use cases. SQL Server Always Encrypted represents a typical application that provides increased data confidentiality and integrity (bit.ly/2zS7TPQ). Always Encrypted protects data in use from malicious insiders with administrative privilege and safeguards

This article discusses:

- Data protection in Azure
- Code and data confidentiality in trusted execution environments
- How to build an enclave and host app with the Open Enclave SDK

Technologies discussed:

Azure Confidential Computing, Open Enclave SDK

against hackers and malware that exploit bugs in the OS, application or hypervisor. With the use of confidential computing, SQL Always Encrypted protects sensitive data in use by providing in-place encryption while preserving SQL Server's rich querying capabilities. This is an enhancement of the current Always Encrypted capability in SQL Server, which now ensures that sensitive data within a SQL database can be encrypted at all times without compromising the functionality of SQL queries. Always Encrypted accomplishes this by delegating computations on sensitive data to an enclave, where the data is safely decrypted and processed.

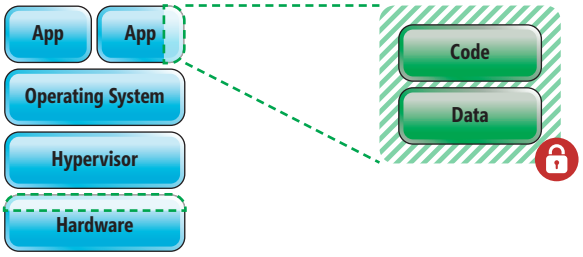


Figure 1 **Code and Data Running in a Protected Enclave**

In addition to SQL Server, many industries and technologies can benefit from Azure Confidential Computing. In finance, for example, personal portfolio data and wealth management strategies would no longer be visible outside of a TEE. Healthcare organizations can collaborate by sharing their private patient data, like genomic sequences, to gain deeper insights from machine learning across multiple data sets, without the risk of data being leaked to other organizations. Organizations could share their datasets confidentially in order to combine multiple data sources to support secure multi-party machine learning scenarios. Machine learning services can obtain a higher accuracy of prediction by working on a larger trained model, but organizations can still preserve their own customer information (data is shared in encrypted format, visible only to the machine learning service). In oil and gas and IoT scenarios, sensitive seismic data that represents the core intellectual property of a corporation can be moved to the cloud for processing, but with the protection of encrypted-in-use technology.

Another significant application is the creation of a trusted distributed network among a set of untrusted participants. The Confidential Consortium Blockchain Framework enables highly scalable and confidential blockchain networks to reside in a public cloud infrastructure and to reap the broad benefits of Azure. Permissioned blockchain networks that rely on trusted nodes called validators to certify transactions benefit from the Azure confidential compute platform to better verify the chain of trust in a decentralized network. This simplifies consensus and, eventually, transaction processing for high throughput and confidentiality.

For this to happen, applications running in an enclave need:

1. A common cross-platform API that's consistent across TEEs, both hardware and software-based, so that confidential application code is portable.
2. Attestation, which involves verifying the identity of code running in TEEs to establish trust with that code and determine whether to release protected data to it.

Let's Get Started

To get started with confidential computing in Azure, you access the Azure Marketplace, deploy and configure a virtual machine, and install the Open Enclave SDK (openenclave.io/sdk). The Open Enclave SDK is an open source project for creating a single unified enclaving abstraction for developers to build TEE-based applications in the C and C++ languages. It supports an API set that lets you build an application once and deploy it on multiple platforms (Linux and Windows) and environments, from cloud to hybrid to edge.

During the deployment of a virtual machine, many of the basic VM deployment configurations are supported through the Confidential Computing VM Deployment workflow in the Azure Portal, including selection of the supported platform, creation of a new or join of an existing resource group and VNet, choice of storage and disk type, enabling diagnostics, and other properties.

You can read how to install the Open Enclave SDK from the official GitHub repository at bit.ly/2AdKs4D.

Once the SDK is installed, you can start building your first application to run in an enclave. As illustrated in **Figure 2**, an enclave application partitions itself into two components: an untrusted component, called the host, and a trusted component, called the enclave. An enclave is a secure container whose memory is protected from access by outside entities, including the host OS, privileged users and even the hardware. All functionality that needs to be run in a TEE should be compiled into the enclave binary. A host is a normal user mode application that loads an enclave into its address space before starting to interact with the enclave.

The Enclave Application

The sample application that I'm going to build prints a message in the enclave before calling back to the host to print a message from there, too. The host initially creates an enclave, and then it calls the enclave_message function in the enclave to print a message. This function then calls back to the host to print an acknowledgment message before returning to the enclave. Once the enclave function returns back to the host, the process is terminated.

First, I define the functions that I want to call between the enclave and host. To do this, I create a functions.edl file, which holds the enclave and host function definitions:

```
enclave {
  trusted {
    public void enclave_message();
  };

  untrusted {
    void host_acknowledgment();
  };
};
```

The EDL file defines the functions that call into and out of enclaves, along with the parameters that are passed into these functions. The oeedger8r tool, available in the Open Enclave SDK, is used to generate the marshaling code necessary to call functions between the enclave and the host. Marshaling code from the host to the enclave is done for security purposes, in order to mitigate certain processor vulnerabilities (such as spectre). You'll find more information on using the Open Enclave oeedger8r tool at bit.ly/2Sg4Dpw.

Let's examine the two functions defined in the EDL file more in detail. The enclave_message function in **Figure 3** is implemented

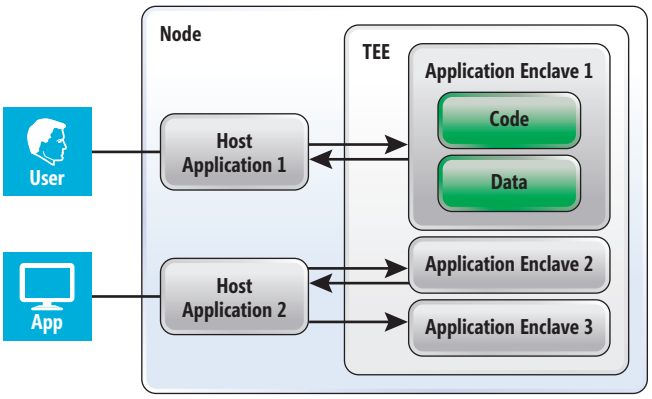


Figure 2 **The Enclave App Model**

inside the trusted enclave and is invoked by the untrusted host. For the host to be able to call this function, the host needs to call through the Open Enclave SDK to transition from the untrusted host into the trusted enclave. To help with this, the oeedger8r tool generates some marshaling code in the host directory with the same signature as the function in the enclave, with the addition of an enclave handle so the SDK knows which enclave will execute the code.

Please note that although enclave_message is using fprintf to print a message, this function has a dependency on the kernel to print a message on the screen, so this code can't execute within the enclave itself. Instead, this function marshals the call through to the host to carry out the call on the enclave's behalf.

The reverse is also true for functions defined in the untrusted host that the trusted enclave needs to call into. The untrusted host runs the host_acknowledgment function, shown in **Figure 4**, and the oeedger8r tool generates some marshaling code with the same signature as the function in the host.

The host process, a regular C-language executable with a standard main function that creates the enclave and calls into it, is what drives the enclave app. It's responsible for managing the lifetime of the enclave and invoking enclave methods. A host, though running in a cloud service, should always be considered an untrusted component that's never allowed to handle plain text data intended for the enclave.

It's worth noting the inclusion of the untrusted functions_u.h header that's generated during the build. This file is created by calling the SDK tool oeedger8r against the functions.edl file. I also include stdio.h for the fprintf function. Unlike the enclave implementation, which includes a special enclave version of the stdio library that marshals APIs to the host, the host isn't protected, so it uses all the normal C libraries and functions. This oe_create_functions_enclave function is generated by oeedger8r. This function creates an enclave for use in the host process and allocates the enclave address space. The code and data to protect are then loaded into the enclave at the allocated address. To execute the host application, which is called functionshost, you can simply invoke it from the command line as:

```
functionshost ./enc/functionsenc.signed
```

Instructions on how to build the application are available on the Open Enclave SDK repository (bit.ly/2CrTM6m). The first parameter of the host's main method identifies the path to the signed enclave library file. For testing purposes, you can also run the application in simulation model with the --simulate command:

```
functionshost ./enc/functionsenc.signed --simulate
```

The "Getting Started with Open Enclave in Simulator mode" guide describes how to set up the enclave simulator (bit.ly/2LvA2BT).

Figure 3 **The Enclave Function**

```
void enclave_message()
{
  // Print a message from the enclave.
  fprintf(stdout, "Hello from the enclave\n");

  // Call back into the host.
  oe_result_t result = host_acknowledgment();
  if (result != OE_OK)
  {
    fprintf(stderr, "Call to host failed: %u (%s)\n", result,
      oe_result_str(result));
  }
}
```

Attestation

Before an enclave can be trusted with confidential data, it needs to prove that it's an enclave running in a valid TEE and that it has the correct identity and runtime properties to be trusted. This process of proving its identity and trustworthiness to a challenger is known as attestation.

Intel SGX supports CPU-based attestation, enabling a remote system to cryptographically verify that specific software has been loaded within an enclave. The process also bootstraps an end-to-end encrypted channel with the enclave for sharing data in a

Figure 4 **The Host Application**

```
#include <openenclave/host.h>
#include <stdio.h>
#include "functions_u.h"

void host_acknowledgment()
{
  fprintf(stdout, "Call from enclave acknowledged.\n");
}

int main(int argc, const char* argv[])
{
  oe_result_t result;
  oe_enclave_t* enclave = NULL;

  // Create the enclave.
  result = oe_create_functions_enclave(argv[1], OE_ENCLAVE_TYPE_SGX,
    OE_ENCLAVE_FLAG_DEBUG, NULL, 0, &enclave);
  if (result != OE_OK)
  {
    fprintf(stderr, "Create enclave failed: %u (%s)\n", result,
      oe_result_str(result));
    return 1;
  }

  // Call into the enclave.
  result = enclave_message(enclave);
  if (result != OE_OK)
  {
    fprintf(stderr, "Call to enclave failed: %u (%s)\n", result,
      oe_result_str(result));
    // Clean up the enclave.
    oe_terminate_enclave(enclave);
    return 1;
  }

  return 0;
}
```

protected manner. During enclave creation, a secure hash, known as a measurement, defines the enclave's initial state. The enclave may later retrieve a report signed by the processor that proves its identity and communicates a unique value (such as a public key) to another local enclave. By using a trusted quoting enclave, this mechanism can be leveraged to obtain an attestation known as a quote, which proves to a remote system that the report comes from an enclave running on a genuine SGX implementation. Ultimately, the processor manufacturer (for example, Intel) is the root of trust for attestation.

The enclave being attested first needs to generate a cryptographically strong proof of its identity that the host can verify. This is done by asking the SGX platform to generate a report signed by the oe_get_report method in the Open Enclave SDK, whose signature is:

```
oe_result_t result = oe_get_report(OE_REPORT_OPTIONS_REMOTE_ATTESTATION,
  reportDashHash, sizeof(reportDashHash), NULL, 0, quoteBuffer, quoteBuffesSize);
```

You can toggle between local and remote forms of attestation by removing or adding the OE_REPORT_OPTIONS_REMOTE_ATTESTATION option, respectively. The local report can only be verified by another instance of this enclave on the same machine, whereas the remote report can be verified by the oe_verify_report method running on a different machine.

An important feature of oe_get_report is that you can pass in application-specific data to be signed into the report. Typically, you sign data (using the reportDashHash parameter) into the report by first hashing it before passing it to the oe_get_report call. This is useful for bootstrapping a secure communication channel between the enclave and the challenger.

Once the report is generated and passed to the challenger, the challenger can call oe_verify_report to validate that the report originated from a valid SGX platform. A local report is verified using the SGX report signing keys held by the platform, and a remote report is verified using the certificate chain issued by Intel only for valid SGX platforms. At this point, the challenger knows that the report originated from an enclave running in a valid SGX platform, and that the information in the report can be trusted:

```
oe_result_t result = oe_verify_report(quote, quoteSize, &parsedReport);
bool verified = memcmp(parsedReport.identity.authorID,
  g_MRSigner, sizeof(g_MRSigner)) == 0;
```

Finally, it's up to the enclave app to check that the identity and properties of the enclave reflected in the report matches its expectation. The Open Enclave SDK exposes a generalized identity model to support this process across TEE types, defined in the oe_identity_t structure:

```
typedef struct _oe_identity
{
uint32_t idVersion;
uint32_t idVersion;
uint32_t securityVersion;
uint64_t attributes;
uint8_t uniqueID[OE_UNIQUE_ID_SIZE];
uint8_t authorID[OE_AUTHOR_ID_SIZE];
uint8_t productID[OE_PRODUCT_ID_SIZE];
} oe_identity_t;
```

I normally test for productID and securityVersion to validate the expected enclave identity, as follows:

```
bool productVerified = parsedReport.identity.productID[0] == 1;
bool versionVerified = parsedReport.identity.securityVersion >= 1;
```

I'd also ensure that the identity of the enclave matches the expected value, by verifying the uniqueID value. Bear in mind that any patches to the enclave will change the uniqueID in the future.

Before You Go

Before the enclave can be run, the properties that define how the enclave should be loaded need to be specified. These properties, along with the signing key, define the enclave identity that's used for attestation and sealing operations. In the Open Enclave SDK, these properties can be attached to the enclave as part of the signing process. To do this, you'll need to use the oesign tool, which takes the following parameters:

```
oesign ENCLAVE CONFFILE KEYFILE
```

It's worth mentioning that, for testing purposes, you can run an enclave in debug mode without signing it first. However, as a word of warning, please be aware that enclaves running in debug mode are not confidential, and you should make sure that debug mode is disabled before deploying an enclave to production. Details on how to build and sign an enclave, and how to enable debug mode, are available on the Open Enclave SDK documentation (bit.ly/2EHOeJ7).

Finally, the host process is what drives the enclave app. It's responsible for managing the lifetime of the enclave and invoking enclave functions. Hosts, though they run in a cloud environment, should be considered an untrusted component that's never allowed to handle clear text or binary data intended for the enclave. As opposed to enclave functions, there are relatively fewer restrictions on building a host application. In general, you're free to link your choice of additional libraries into the host application. In contrast, enclave functions have limited support for external libraries, for security reasons. As the Open Enclave SDK evolves, its support for additional libraries continues to improve. For information on supported libraries, please consult the Open Enclave SDK Web site (openenclave.io).

It's Open Source!

As I mentioned, the Open Enclave SDK is open source! The intention is for it to be a non-vendor-specific solution that supports enclave applications on both the Linux and Windows platforms. The current implementation of Open Enclave is built on Intel SGX; other enclave architectures, such as solutions from AMD or ARM, will be added in the future.

This 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 the Open Enclave team the rights to use your contribution. If you would like to contribute to the Open Enclave SDK, please refer to the guidelines for contribution at bit.ly/2TbQ4DJ. ■

Stefano Tempesta *is a Microsoft Regional Director, MVP on AI and Business Applications, and member of 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" (gallery.azure.ai).*

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

DINO ESPOSITO

Dealing with Forms in Blazor

ASP.NET Core Blazor is a C#-based, client-side framework to build single-page applications (SPAs). In this regard, it's not much different from Angular, in that it can interact with whatever back end you may have that exposes HTTP-reachable endpoints. At this time, however, Blazor is a work in progress and far from go-live.

That said, some parts of Blazor are shipping as part of ASP.NET Core 3.0. Razor Components is a special configuration of a Blazor application that runs entirely on the server, side-by-side with a classic ASP.NET Core application that might constitute its back end. The coexistence of a classic Web application mostly used for back-end tasks and a distinct Blazor-based presentation layer on the ASP.NET server simplifies some programming scenarios, and establishes a sort of middle ground between a pure SPA approach and a pure server approach. In this article, I'll discuss how to deal with input forms and client-to-server communication. Let's start with a quick survey of Razor Components.

Server-Side Blazor

Blazor is a core framework designed to receive and handle events regardless of the surrounding environment. Today, the only fully supported scenarios are running Blazor within the UI thread of a hosting browser and within the ASP.NET Core runtime. Other realistic scenarios yet to come are running Blazor within a Web worker or even within some desktop-enabling platforms such as Electron. Hosting in a Web worker would make Blazor a suitable platform for progressive and offline Web applications, while hosting in a platform like Electron is also reasonable. As an example, let's briefly compare the instructions for the very basic Electron startup (see github.com/electron/electron-quick-start) with what happens when a new server-side Blazor project runs.

The Electron's quick-start source code contains the following method:

```
function createWindow () {
  mainWindow = new BrowserWindow({width:
    800, height: 600})
  mainWindow.loadFile('index.html')
}
```

The method is responsible for creating the browser window and for loading a separate HTML file into it. All that happens next is a continuous exchange of messages between the outside world (the browser) and the innermost shell (the HTML pages). The framework that sits in between just manages the burden of proxying messages appropriately. Let's see what happens when a server-side Blazor application runs, as depicted in **Figure 1**.

First, the browser loads the application-level index.html file that defines the main page layout (mostly the HEAD tag) and, as part of the download, also loads a small JavaScript file named blazor.server.js. The _blazor/negotiate step in **Figure 1** indicates the moment an ASP.NET Core SignalR connection is established between the browser's host environment and the Blazor app. Finally, the connection is upgraded to Web Sockets. This happens in the final step captured by the Fiddler screen in **Figure 1**.

In both cases, the host environment loads up an initializer file. Once the connection is established, the application logic runs and produces any required HTML. In server-side Blazor, the HTML produced server side is compared to the currently rendered DOM and only the necessary fragments of the DOM are actually updated. All updates run over the SignalR connection.

User clicks captured within the browser go the other way around. Related events are packaged up and sent server side, where they're processed by the Blazor JavaScript interop layer and processed in C# code.

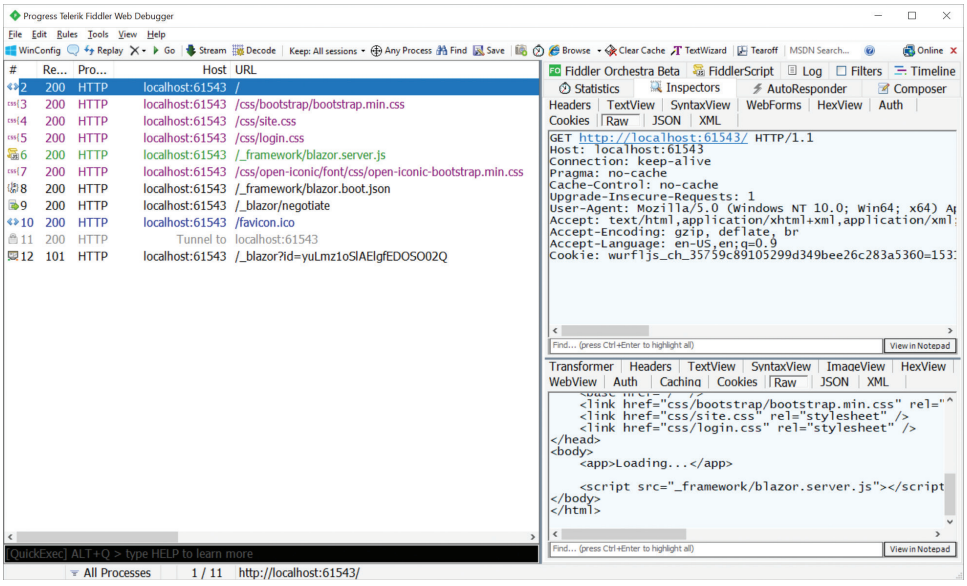


Figure 1 **The Loading Phase of a Server-Side Blazor Application**

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

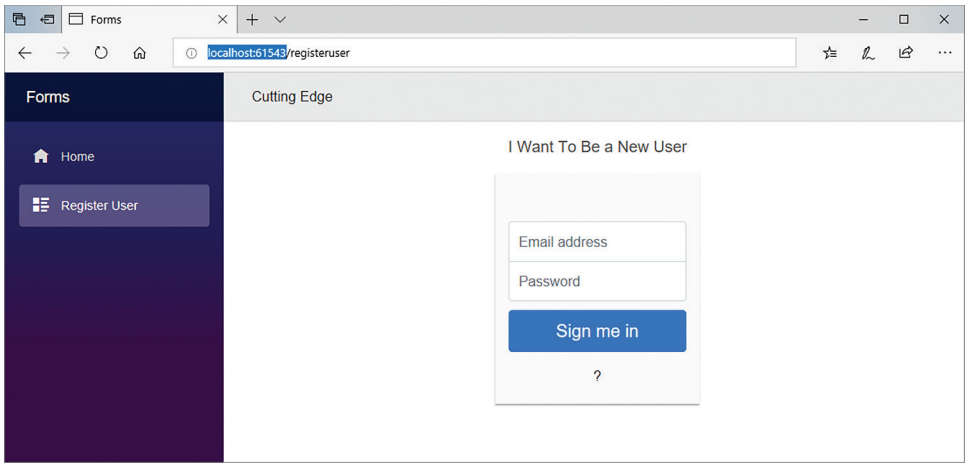


Figure 2 **The Sample Form**

Visual Studio comes with an ad hoc project template for server-side Blazor applications. The typical project includes a client project and an ASP.NET Core project. Most likely, you'll want to add a .NET Standard library project to hold classes that must be shared between the front end and the back end.

Setting Up the Client App

The startup class of the client application is nearly empty:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
  }

  public void Configure(IBlazorApplicationBuilder app)
  {
    app.AddComponent<App>("app");
  }
}
```

The only necessary change is injecting a reference to HttpClient so that any of the Blazor pages will be able to place HTTP remote calls:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddScoped<HttpClient>();
}
```

The most relevant addition to the client application is a component that allows users to fill out and post a form. In **Figure 2** you see the application UI produced by the Register User button. It's a plain simple HTML form that collects an e-mail address and password from the user and passes it back to a remote back end.

The interesting thing here is the markup you need to generate the container for the input fields in **Figure 2**. Even though the container walks and quacks like an HTML form, it's not a true HTML form, as shown in **Figure 3**.

As you can see, the markup is pure HTML, but there's no need for a FORM element. In the end, the C# code running on the client will take care of collecting the input values and will forward them to a remote endpoint that might (or might not) be an ASP.NET back end.

The bind attribute you see associated with the INPUT elements guarantees bidirectional binding between the pseudo-form and the properties of the Blazor client page. The TryRegister function is responsible for posting the content to the back end. **Figure 4** shows the code of the client page's @functions block.

The content of the input fields are gathered in a fresh instance of a data-transfer object—RegisterUserViewModel—and serialized as JSON through the PostJsonAsync method of HttpClient. The back end receives an HTTP POST request, in which the body is set as follows:

```
{"email":"...","password":"..."}
```

In a server-side Blazor application, the back end is usually an ASP.NET Core application. In this case, the request is forwarded to a controller endpoint where model binding applies. As a result, the posted data will be captured by a server-side instance of the same RegisterUserViewModel class. Note, though, that the back end doesn't have to necessarily be an ASP.NET Core application. The URL you specify in the call via HttpClient must be an absolute URL that can post data to virtually anywhere, including some legacy Visual Basic-based back end.

Figure 3 **Markup to Produce a (Non-HTML) Form**

```
<div class="col-sm-6 col-md-4 offset-md-4">
  <h3 class="text-center login-title">I Want To Be a New User</h3>
  <div class="account-wall">
    <div class="form-signin">
      <input type="text"
                  class="form-control"
                  name="email" bind="@Email" />
      <input type="password"
                  class="form-control"
                  name="password" bind="@Password" />
      <button class="btn btn-primary
                     onclick="@TryRegister">
        Sign me in
      </button>
    </div>
    <div class="text-center">@Message</div>
  </div>
</div>
```

Setting Up the Server App

In the sample application, the server application is a plain ASP.NET Core Web API, secured as you would secure any such Web API. For example, it might be protected against unauthorized access via a JWT token that would work well also with non-Web clients, including desktop and mobile clients. For now, let's just assume that the Web API allows anonymous access. Here's the code you need to successfully receive and process any Blazor-posted data:

```
public class AccountController : Controller
{
  [HttpPost]
  public IActionResult Register(
    [FromBody] RegisterUserViewModel input)
  {
    // Some work here
      ...
    return Json(CommandResponse.Ok.AddMessage("Done"));
  }
}
```

The FromBody attribute plays a key role here. Without it any call made toward the endpoint, the way the Blazor client does it, would throw a model-binding exception.

The FromBody attribute instructs the ASP.NET Core runtime to bind the parameter to the data found in the body of the incoming HTTP request. Note that there can be at most one parameter per action decorated with the FromBody attribute.

ASP.NET Core delegates the responsibility of processing the request stream to a dedicated formatter class. The default class is JsonInputFormatter. Note also that once the request stream has been read for a parameter, it can't be read again for another parameter. The stream pointer, in fact, has advanced to the end of the body and can't be moved back.

The response from the Web API is another JSON response that the Blazor client receives and deserializes, like so:

```
var response = await
  Http.PostJsonAsync<CommandResponse>(absoluteUrl, input);
```

In this code snippet, the response is deserialized to an instance of the CommandResponse type and an exception is thrown if it turns out to be impossible.

A Word on Security and Authentication

The client part of Blazor runs in a sandbox in the same way that JavaScript code does. The server back end is completely disconnected from the client and reachable only at the conditions the back end defines. The Blazor client app must comply with whatever security layer you have in place around the back end. In this regard, Blazor is a plain Web-based client that calls into a Web API. It may authenticate through a cookie, but more likely it will need to authenticate through a JWT bearer token.

The server-side Blazor scenario may be a different story, in the sense that it may enable alternate scenarios for authentication and authorization. No decision has been made yet on this point, though you can get up to speed on the discussion at bit.ly/2CdS74c. Chances are that some built-in API will be provided to streamline the creation of login and logout forms for server-side Blazor applications, as well as for other common authentication operations such as recovering a password and interacting with the internal membership system. It's possible that Blazor will in some way incorporate the built-in UI of ASP.NET Core Identity. But again, nothing is decided at the time of this writing.

Figure 4 **The @functions Block**

```
@inject HttpClient Http
...
@functions
{
  public string Email { get; set; }
  public string Password { get; set; }
  public string Message = "?";

  public async Task TryRegister()
  {
    var input = new RegisterUserViewModel(Email, Password);
    try
    {
      var response = await Http.PostJsonAsync<CommandResponse>(
        ".../account/register", input);
      if (response.Success)
        Message = response.Message;
    }
    catch (Exception e)
    {
      Console.WriteLine(e);
      throw;
    }
  }
}
```

Pros and Cons of Razor Components

Server-side Blazor, or Razor components, will be the first part of Blazor to reach go-live status when it ships as part of ASP.NET Core 3.0. Some developers may balk at the drawbacks. Used to set up a pure SPA solution, Blazor may seem too distant from the programming experience of classic ASP.NET Web developers. Also, WebAssembly is required to run C# within the browser, which imposes a cost in terms of download and initial startup, while debugging can be problematic. With a server-side solution, the development experience is smoother, with improved debugging, JIT compilation and a larger API. What's more, all browsers are supported, regardless of their native support for WebAssembly. For a classic ASP.NET developer, refreshing only parts of the UI becomes nearly as easy as in a desktop application.

With a server-side solution, the development experience is smoother, with improved debugging, JIT compilation and a larger API.

The issue is that all this magic relies on a SignalR connection. This brings up a few potential problems. First, there will be quite a number of requests going over the wire. Any updates, in fact, require a roundtrip. Second, the server is forced to track every client and its state. ASP.NET Core SignalR is also offered as an Azure service—and this provides a great foundation for scalability—but nobody has tried it out yet in the real world. Finally, there's no built-in mechanism yet to restore the application state in case the SignalR connection drops. Any persistent solution is good, but it's all left to developers for now.

All in all, Blazor promotes the SPA pattern and the SPA pattern is a bit disruptive for many developers. Server-side Blazor (aka, Razor Components) is a smart, non-disruptive way to move piecemeal toward the SPA architecture. The source for this article is based on Blazor 0.7.0 and is available at bit.ly/2EpGF8d. ■

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 technical expert for reviewing this article: *Jonathan Miller*

JAMES MCCAFFREY

Rating Competitors Using Infer.NET

Infer.NET is an open source code library that can be used to create probabilistic programming systems. I tend to think of an ordinary computer program as one that's largely based on variables that have a value of a specified type, such as a char variable that has value "Q." Probabilistic programming is largely based on probability distributions, such as a Gaussian distribution that has mean 0.0 and standard deviation 1.0.

In this article, I show you how to get started with the Infer.NET library by computing ratings for a set of competitors using head-to-head results. A good way to get an idea of probabilistic programming and see where this article is headed is to take a look at the demo program in **Figure 1**. The demo sets up six competitors (think sports teams): Angels, Bruins, Comets, Demons, Eagles and Flyers.

The six teams play against each other. Each team plays three times so there are a total of nine games. The demo program defines a probabilistic model that assumes each team has an intrinsic strength and that each strength can be described by a Gaussian (also called normal or bell-shaped) distribution with a mean of 2000 and a standard deviation of 200 arbitrary units.

Using the win-loss data, the demo program infers the strengths of the six teams. The Angels won three games and lost none and have an inferred strength of 2256.8 units, which is about 1.25 standard deviation units better than the assumed average strength of 2000 units. The Flyers lost all three of their games and have an inferred strength of 1739.7 units.

I use the term strengths here but you can think of the inferred numeric values as ratings. Notice that if you can infer the ratings of a set of items, you automatically get a ranking of the items. The final ranking, from best to worst, is Angels, Bruins, Comets, Eagles, Demons, Flyers.

This article assumes you have intermediate or better programming skills with C#, but doesn't assume you know anything about Infer.NET or probabilistic programming. Infer.NET supports only C# and F#, so you could refactor the demo to F# if you wish. Once you understand the basics of probabilistic programming, you should be able to rewrite the demo program using one of the many other probabilistic programming frameworks, such as Stan or Edward.

Code download available at msdn.com/magazine/0219magcode.

The complete source code for the demo program is presented in this article. The source code is also available in the accompanying file download. All normal error checking has been removed to keep the main ideas as clear as possible.

Understanding Random Variables

The demo program assumes that each team's strength is a Gaussian-distributed random variable with a specified mean (average) and standard deviation. What exactly does this mean and from where does this assumption come?

There are many kinds of random variable distributions. Each has one or more characteristic parameters. For example, a random variable that follows a uniform distribution with a = 2.0 and b = 5.0 can be any value between 2.0 and 5.0, where each possible value is equally likely. For the demo problem it would be possible to assume

```
C:\InferStrengths\bin\Debug\InferStrengths.exe

Begin Infer.NET demo

Data:

game: 0   winning team: Angels   losing team: Bruins
game: 1   winning team: Bruins   losing team: Comets
game: 2   winning team: Comets   losing team: Demons
game: 3   winning team: Angels   losing team: Eagles
game: 4   winning team: Bruins   losing team: Demons
game: 5   winning team: Demons   losing team: Flyers
game: 6   winning team: Angels   losing team: Flyers
game: 7   winning team: Comets   losing team: Eagles
game: 8   winning team: Eagles   losing team: Flyers

Defining Gaussian model with mean = 2000.0 and sd = 200.0

Inferring strengths from win-lose data

Compiling model...done.
Iterating:
.........|.........|.........|.........| 40

Inference complete. Inferred strengths:

Angels: 2256.8
Bruins: 2135.3
Comets: 2045.5
Demons: 1908.0
Eagles: 1914.6
Flyers: 1739.7

End demo
```

Figure 1 **Infer.NET Rating in Action**

Figure 2 **Complete Source Code**

```
using System;
using Microsoft.ML.Probabilistic.Models;
using Microsoft.ML.Probabilistic.Algorithms;
using Microsoft.ML.Probabilistic.Distributions;
// VS2017 (Framework 4.7) Infer.NET 0.3.1810.501

namespace InferStrengths
{
  class InferStrengthsProgram
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Begin Infer.NET demo ");

      // ===== Set up teams and win-loss data ====================

      string[] teamNames = new string[] { "Angels", "Bruins",
        "Comets", "Demons", "Eagles", "Flyers" };
      int N = teamNames.Length;

      int[] winTeamIDs =  new int[] { 0, 2, 1, 0, 1, 3, 0, 2, 4 };
      int[] loseTeamIDs = new int[] { 1, 3, 2, 4, 3, 5, 5, 4, 5 };

      Console.WriteLine("Data: \n");
      for (int i = 0; i < winTeamIDs.Length; ++i) {
        Console.WriteLine("game: " + i + "   winning team: " +
          teamNames[winTeamIDs[i]] + "   losing team: " +
          teamNames[loseTeamIDs[i]]);
      }

      // ===== Define a probabilistic model ====================

      Range teamIDsRange = new Range(N).Named("teamsIDRange");
      Range gameIDsRange =
        new Range(winTeamIDs.Length).Named("gameIDsRange");

      double mean = 2000.0;
      double sd = 200.0;
      double vrnc = sd * sd;

      Console.WriteLine("\nDefining Gaussian model mean = " +
        mean.ToString("F1") + " and sd = " + sd.ToString("F1"));
      VariableArray<double> strengths =
        Variable.Array<double>(teamIDsRange).Named("strengths");

      strengths[teamIDsRange] =
        Variable.GaussianFromMeanAndVariance(mean,
        vrnc).ForEach(teamIDsRange);

      VariableArray<int> winners =
        Variable.Array<int>(gameIDsRange).Named("winners");
      VariableArray<int> losers =
        Variable.Array<int>(gameIDsRange).Named("losers");

      winners.ObservedValue = winTeamIDs;
      losers.ObservedValue = loseTeamIDs;

      using (Variable.ForEach(gameIDsRange))
      {
        var ws = strengths[winners[gameIDsRange]];
        var ls = strengths[losers[gameIDsRange]];
        Variable<double> winnerPerf =
      Variable.GaussianFromMeanAndVariance(ws, 400).Named("winPerf");
        Variable<double> loserPerf =
      Variable.GaussianFromMeanAndVariance(ls, 400).
        Named("losePerf");

        Variable.ConstrainTrue(winnerPerf > loserPerf);
      }

      // ===== Infer team strengths using win-loss data ==========

      Console.WriteLine("\nInferring strengths from win-loss data");
      var iengine = new InferenceEngine();
      iengine.Algorithm = new ExpectationPropagation();
      iengine.NumberOfIterations = 40;
      // iengine.ShowFactorGraph = true;  // needs Graphviz install

      Gaussian[] inferredStrengths =
        iengine.Infer<Gaussian[]>(strengths);
      Console.WriteLine("Inference complete. Inferred strengths: ");

      // ===== Show results ======================================

      for (int i = 0; i < N; ++i) {
        double strength = inferredStrengths[i].GetMean();
        Console.WriteLine(teamNames[i] + ": " +
          strength.ToString("F1"));
      }

      Console.WriteLine("\nEnd demo ");
      Console.ReadLine();
    } // Main
  }
} // ns
```

that team strength is uniformly distributed with a = 1000 and b = 3000, which would imply that many such strengths would have a mean of 0.5 * (1000 + 3000) = 2000.0 units and a standard deviation of sqrt((1/12) * (3000 - 1000)^2) = 577.35 units.

The demo assumes that team strengths are Gaussian with mean = 2000 and standard deviation = 200, and therefore the average strength of many such teams would be about 2000 and about 99 percent of many such teams would have strengths between 2000 - (3 * 200) = 1400 and 2000 + (3 * 200) = 2600.

At this point, you may be thinking, "Great, so I have to have a Ph.D. in statistics distribution to use Infer.NET and create probabilistic programs." This isn't entirely true. Infer.NET supports many distributions, but in practice you usually need to understand just a few. The ones I use most frequently are Gaussian, uniform, beta, binomial, multinomial, gamma and Poisson. An analogy is the many types of data structures supported by the System.Collections namespace of the .NET Framework. Even though there are many, you typically use only a few (stacks, queues, dictionaries), and you learn about new ones as you encounter problems that need them.

The mean is the average value of the data and standard deviation is a measure of how spread out the data is. The variance is the square of the standard deviation, and the precision is the inverse of the variance. The reason there are three different terms that contain the exact same information is that sometimes one form is more convenient than the others when using it in a math equation of some sort. The Infer.NET library tends to use variance and precision rather than standard deviation.

> Infer.NET supports many distributions, but in practice you usually need to understand just a few.

The Demo Program Structure

The complete source code for the demo program, with a few minor edits to save space, is presented in **Figure 2**.

To create the demo program, I launched Visual Studio 2017 Community Edition and selected the console application template

project with .NET Framework version 4.7 and named it InferStrengths. Infer.NET can run in a .NET Core application, but I prefer the classic .NET Framework.

After the template code loaded, I right-clicked on file Program.cs in the Solution Explorer window and renamed the file to InferStrengthsProgram.cs and then allowed Visual Studio to automatically rename class Program for me.

I right-clicked on the project name in the Solution Explorer window and selected the Manage NuGet Packages option. In the NuGet window, I selected the Browse tab and then searched for "Infer.NET." In the results list I selected the Microsoft.ML.Probabilistic.Compiler package (version 0.3.1810.501) and clicked the install button. Microsoft plans to migrate Infer.NET into the ML.NET library at some point, so if you can't find Infer.NET as a standalone package, look for it inside the ML.NET package.

Setting up the Data

The demo sets up the six sports teams like so:

```
string[] teamNames = new string[] { "Angels", "Bruins",
  "Comets", "Demons", "Eagles", "Flyers" };
int N = teamNames.Length;
```

Because Infer.NET works primarily with random variables, in many programs all data is strictly numeric. Here, the team names are specified as strings rather than integer indexes just for readability. The win-loss data is:

```
int[] winTeamIDs =
  new int[]  { 0, 2, 1, 0, 1, 3, 0, 2, 4 };
int[] loseTeamIDs =
  new int[]  { 1, 3, 2, 4, 3, 5, 5, 4, 5 };
```

The data means in game [0], team 0 (Angels) beat team 1 (Bruins). In game [1], team 2 (Comets) beat team 3 (Demons), and so on through game [8]. With numerical programming, using parallel arrays like this tends to be a more common pattern than placing the data into a class or struct object.

Notice that at this point the demo is using only native .NET data types. However, Infer.NET has its own type system and the data will be converted into types useable by Infer.NET shortly. The demo displays the source data:

```
for (int i = 0; i < winTeamIDs.Length; ++i) {
  Console.WriteLine("game: " + i + " winning team: " +
    teamNames[winTeamIDs[i]] + " losing team: " +
    teamNames[loseTeamIDs[i]]);
}
```

There is very little data in the demo. The ability to work with limited data is a strength of probabilistic programming compared to some machine learning techniques, such as neural networks or reinforcement learning, that often require very large amounts of training data to be effective.

Defining the Probabilistic Model

The demo prepares the Gaussian model with these statements:

```
Range teamIDsRange = new Range(N).Named("teamsIDRange");
Range gameIDsRange =
  new Range(winTeamIDs.Length).Named("gameIDsRange");
double mean = 2000.0;
double sd = 200.0;
double vrnc = sd * sd;
```

The idea of a Range object is important in Infer.NET. Unlike standard procedural programming where it's common to explicitly iterate using a for loop or a foreach loop, in Infer.NET it's more common to apply meta operations via a Range object. This is a coding paradigm that can be a bit difficult to get used to.

Almost all Infer.NET objects can be supplied with an optional string name by using the Named method chained to the constructor call. The object string name doesn't need to match the object identifier name, but it's good practice to make them the same. As you'll see shortly, using the Named method is useful because Infer.NET has a built-in way to display a model's computational graph, and the graph will use a string name if available rather than a library-generated name like vdouble23.

The team strengths to infer are set up with these two statements:

```
VariableArray<double> strengths =
  Variable.Array<double>(teamIDsRange).Named("strengths");
strengths[teamIDsRange] =
  Variable.GaussianFromMeanAndVariance(mean,
  vrnc).ForEach(teamIDsRange);
```

The first statement sets up a single object as a special VariableArray type that consists of six random Variable objects. The second statement initializes each random variable as a Gaussian with mean = 2000 and variance = 4000 (which is equivalent to standard deviation = 200). It's possible to set up an array of individual Variable objects along the lines of:

```
Variable<double>[] arr = new Variable<double>[6];
for (int i = 0; i < 6; ++i)
  arr[i] = Variable.GaussianFromMeanAndVariance(2000, 4000);
```

However, the Infer.NET documentation recommends using the VariableArray approach for performance reasons. Next, random variables of type int are created to hold the indexes of the winning and losing teams, and then the native .NET int data is transformed into Infer.NET objects:

```
VariableArray<int> winners =
  Variable.Array<int>(gameIDsRange).Named("winners");
VariableArray<int> losers =
  Variable.Array<int>(gameIDsRange).Named("losers");
winners.ObservedValue = winTeamIDs;
losers.ObservedValue = loseTeamIDs;
```

In this situation, the random variables don't follow a probability distribution; instead, they're essentially just ordinary integer values. The ObservedValue property is used to set fixed values in this situation.

The key statements that define the characteristics of the probabilistic model are:

```
using (Variable.ForEach(gameIDsRange)) {
  var ws = strengths[winners[gameIDsRange]];
  var ls = strengths[losers[gameIDsRange]];
  Variable<double> winnerPerf =
    Variable.GaussianFromMeanAndVariance(ws,
    400.0).Named("winPerf");
  Variable<double> loserPerf =
    Variable.GaussianFromMeanAndVariance(ls,
    400.0).Named("losePerf");

  Variable.ConstrainTrue(winnerPerf > loserPerf);
}
```

This code is quite subtle. The code is inside a Variable.ForEach block so the operations are applied to each team and each game outcome in a meta fashion. Random variables that correspond to the performance of the winning and losing teams for each game are defined so that they're based on the team's strength, but they have a noise component that's Gaussian-distributed with variance = 400. One way to think of this is that when the team strengths are being inferred, there's a chance that one team may slightly underperform and the opponent may slightly outperform. The noise variance value of 400 must be determined by trial and error.

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

#VSLive

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! Boston, including everything Tuesday – Thursday at the conference.

Register by April 10 & Save Up To $400!

Your Adventure Starts Here!

vslive.com/boston

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360 An 1105 MEDIA company

The ConstrainTrue statement is critical and adds the logic that allows the inference engine to compute each team's strength. The inference engine uses a sophisticated algorithm to examine different values of means and variances for each of the six teams and then determines how likely the observed win-loss outcomes are, given the assumed means and variances. The inference algorithm finds the six means and variances that best match the observed data. Clever!

Inferring the Team Strengths

Once the probabilistic model is defined, an inference engine is created with these statements:

```
var iengine = new InferenceEngine();
iengine.Algorithm = new ExpectationPropagation();
iengine.NumberOfIterations = 40;
// iengine.ShowFactorGraph = true;
```

Infer.NET supports three different algorithms: expectation propagation, variational message passing and Gibbs sampling. Different algorithms apply to different types of probabilistic models. For the demo model, only expectation propagation is valid. Expectation propagation is unique to Infer.NET and minimizes the Kullback-Liebler divergence metric to approximate the probability distribution for a set of observed data. The value of the NumberOfIterations property must be determined by trial and error.

An interesting feature of Infer.NET is that you can automatically generate a visual representation of a model's computational graph, such as the one in **Figure 3** that corresponds to the demo program. This feature has a dependency on the open source Graphviz program. If you set ShowFactorGraph to true and Graphviz is installed, a graph will be saved in SVG format in the current working directory, and it can be displayed in a browser.

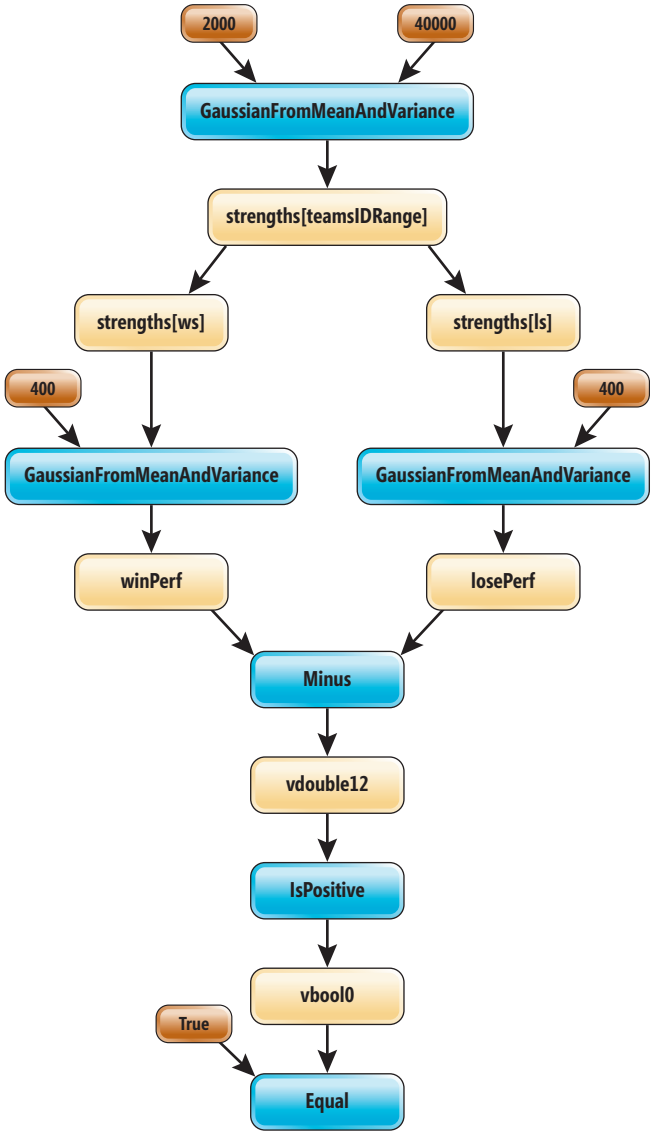


Figure 3 **Visual Representation of the Computational Graph**

> An interesting feature of Infer.NET is that you can automatically generate a visual representation of a model's computational graph.

After the inference engine is created, it's easy to compute and display the team strengths using the Infer method:

```
Gaussian[] inferredStrengths =
  iengine.Infer<Gaussian[]>(strengths);
for (int i = 0; i < N; ++i) {
  double strength = inferredStrengths[i].GetMean();
  Console.WriteLine(teamNames[i] + ": " + strength);
}
```

The strengths object is passed to Infer. Recall that strengths is type VariableArray, which is a collection of Gaussian random variable objects that are associated to the winners and losers VariableArray objects, which have the win-loss data. Notice that an Infer.NET model is a collection of loosely connected objects rather than a single top-level object. For me, at least, this idea took a while to get used to when I was new to Infer.NET.

Wrapping Up

Probabilistic programming using Infer.NET has a much different feel to it than programming using standard procedural languages, and there's a non-trivial learning curve involved. The example presented in this article is loosely based on the TrueSkill ranking system developed by Microsoft Research for use in Xbox matchmaking. But probabilistic programming can be applied to many problems other than rating and ranking. Somewhat unusually for a code library that has its origins in research, Infer.NET has excellent documentation. If you want to learn more about probabilistic programming, I recommend that you take a look at the tutorials at bit.ly/2rEr784. ■

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

TECHMENTOR

IN-DEPTH TRAINING FOR IT PROS

Training Conference for IT Pros at Microsoft HQ!

The FUTURE of TECH is HERE

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

DAVID S. PLATT

Our Event Horizon

This column is running in the February issue of *MSDN Magazine*, but I had to submit it before Christmas. The turn of the year, with darkness stubborn at the edges of the day, leads me to contemplate life, the universe and our shared profession. Today I'm pondering how this crazy computing business started—with Alan Turing, of course. I'm re-reading Charles Petzold's book, "The Annotated Turing" (Wiley, 2008), an excellent guide to our first guru and his foundational paper.

Turing wasn't originally thinking about real computing machinery. Though he later went on to help build some hardware, the concept of cat videos in everyone's pocket probably never crossed his mind. Rather, his Turing machines were thought experiments imagined in response to Kurt Gödel's First Incompleteness Theorem (1931). Gödel proved that if mathematics were consistent, theorems would have to exist that were true, but could never be proven—not because we weren't smart enough, but because of the basic structure of the universe, like Heisenberg's contemporaneous Uncertainty Principle. Turing wanted to find ways to identify these unprovable theorems. Could any finite algorithm do that? (Spoiler: No.)

> Taking inspiration from Kurt Gödel's First Incompleteness Theorem and Werner Heisenberg's Uncertainty Principle, intrepid columnist David Platt asks, "What kinds of questions can Google never answer, no matter how big or how good it gets?"

What does that have to do with today, I hear you wondering. Here's what. I was having dinner with my parents, and my mother asked me to check the score of the Patriots (American football) game. As I wrote last December, accessing this information is difficult for seniors, and shouldn't be (see msdn.com/magazine/mt832865). But I still have enough dexterity to Google "New England Patriots" on my phone, and enough visual acuity to identify them beating up on some hapless opponent. Then my father asked me to find the blessings over the Chanukah candles that we would start lighting that night. Again, a quick Google, and bang, right there were my answers, in Hebrew, English translation and transliteration; even videos of professional cantors singing them, with and without cats (youtu.be/AQvt5mMzJVA and youtu.be/YZDOQXPj_XI, respectively).

That encounter started me thinking: Just about any question of fact can be answered by Googling. But, similar to Turing, I pondered which questions cannot, and how we can identify them.

I'm not asking whether Google currently contains a particular item, as Turing wasn't investigating the proof of any particular statement. Google doesn't know where my car keys are right now (damn), but if I put a Tile tracker on my keyring, it could find out. What kinds of questions can Google never answer, no matter how big or how good it gets? Where is its quantum limit, its event horizon?

Google can't easily decipher the spin of human languages. So it can't tell me if a government payment for installing solar panels on my roof is an incentive (good) or a subsidy (bad). Am I foolishly taking out a second mortgage, or wisely adding a home equity line of credit to my comprehensive financial plan? Is this annoying software behavior an issue or a bug (see msdn.com/magazine/ff955613)? Am I a cranky old fart, or a noble silverback? Google can't say.

My daughter Lucy pointed out that it's hard to Google one's own medical symptoms, except in trivial cases. ("OK Google, my nose is bleeding!" "Hmm, I think you have a nosebleed.") The signal-to-noise ratio of the medical world is too low, and humans too suggestible.

Knowledge of horrible but obscure diseases was once limited to heavy, expensive, paper medical textbooks. Students would scare themselves silly with induced hypochondria so common that it earned a name: "medical student syndrome" (see bit.ly/2GVcRIZ). "Tingling in my left nostril, only on Tuesdays, hmm, must be McCloskey's disease. It says here my nose will fall off next week."

All of that information is now online, instantly searchable by any civilian. "Wait, does my pinky hurt too? Incipient Ebola, has to be." Plus, as a cancer-surviving reader pointed out to me, your search results are swamped by people trying to make money off you. Google can't help much here, either.

I've only been working this problem for about a month, so I haven't fully generalized these results. What do you think so far? Gödel, Heisenberg, Turing and Plattski? ■

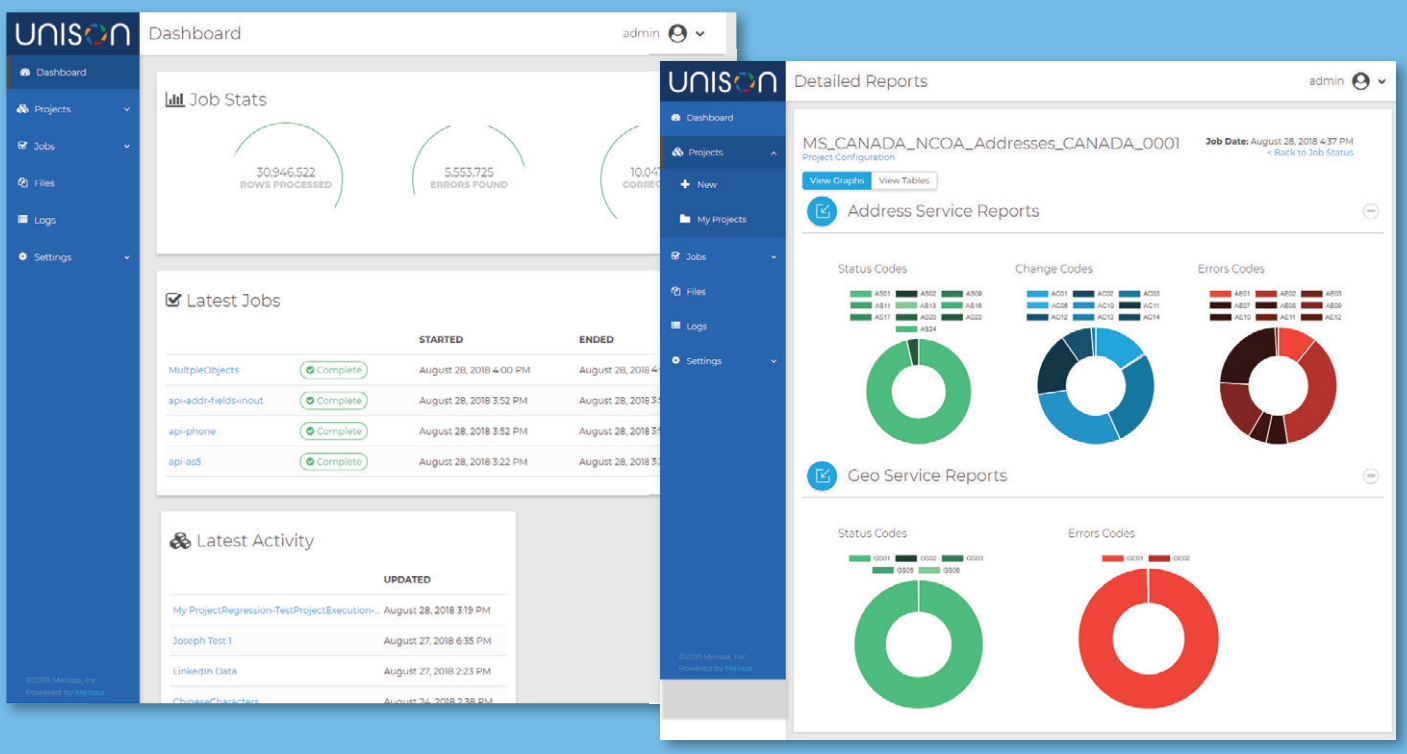
DAVID S. PLATT *teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him 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. It's the ideal multiplatform solution to establish and maintain contact data quality at higher speeds – processing 30+ 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, as data never leaves your organization. Streamline data prep workflows, reduce analytics busy work, gain more insights and increase efficiency!

- Verify and standardize names, addresses, emails and phone numbers, plus append lat/long coordinates and Census data.
- On-Premise platform requiring no coding or programming with automatic updates from Melissa.
- Works offline, scalable across multiple servers and allows users to script batch jobs with various levels of data access.
- Access, manage and visually analyze the quality of your data over time, enjoy project collaboration and more!



Request a Demo

Melissa.com/msdn-unison

1-800-MELISSA

melissa®