THE MICROSOFT JOURNAL FOR DEVELOPERS

JUNE 2019 VOL 34 NO 6

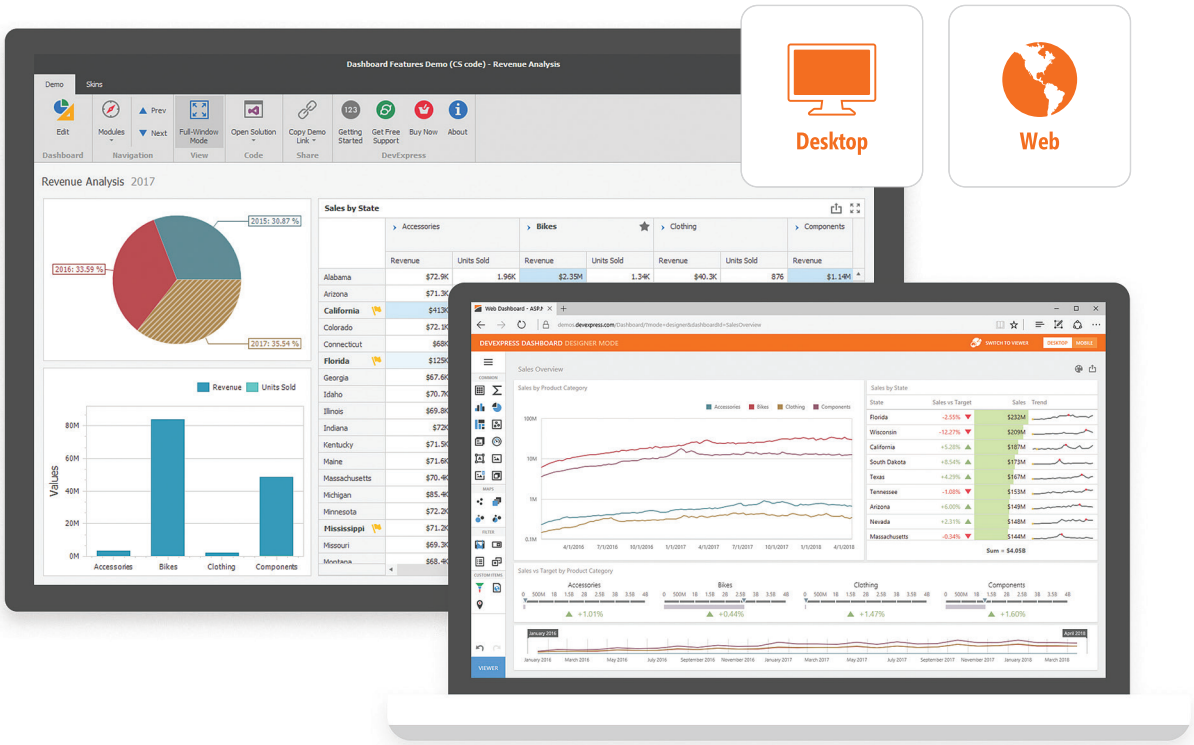
msdn magazine

AI and Biometric Security..18

DevExpress®

DevExpress Dashboard for .NET

Create and distribute royalty-free decision support systems and effortlessly share business intelligence across your entire enterprise.



Data Grid · Pivot Grid · Card View · Charts · Gauges · Treemap · Map · Range Control · Filter Control · Rich Text Editor · Image

With DevExpress Dashboard for .NET, creating insightful and information-rich decision support systems for executives and business users across platforms and devices is a simple matter of selecting the appropriate UI element (Chart, Pivot Table, Data Card, Gauge, TreeMap, Map, Grid or simple Filter elements) and dropping data fields onto corresponding arguments, values, and series. And because DevExpress Dashboard automatically provides the best data visualization option for you, results are immediate, accurate and always relevant.

Learn More Today — Try DevExpress Dashboard Risk Free for 30 days

devexpress.com/dashboard

THE MICROSOFT JOURNAL FOR DEVELOPERS

JUNE 2019 VOL 34 NO 6

msdn magazine

AI and Biometric Security..18

COLUMNS

Microsoft

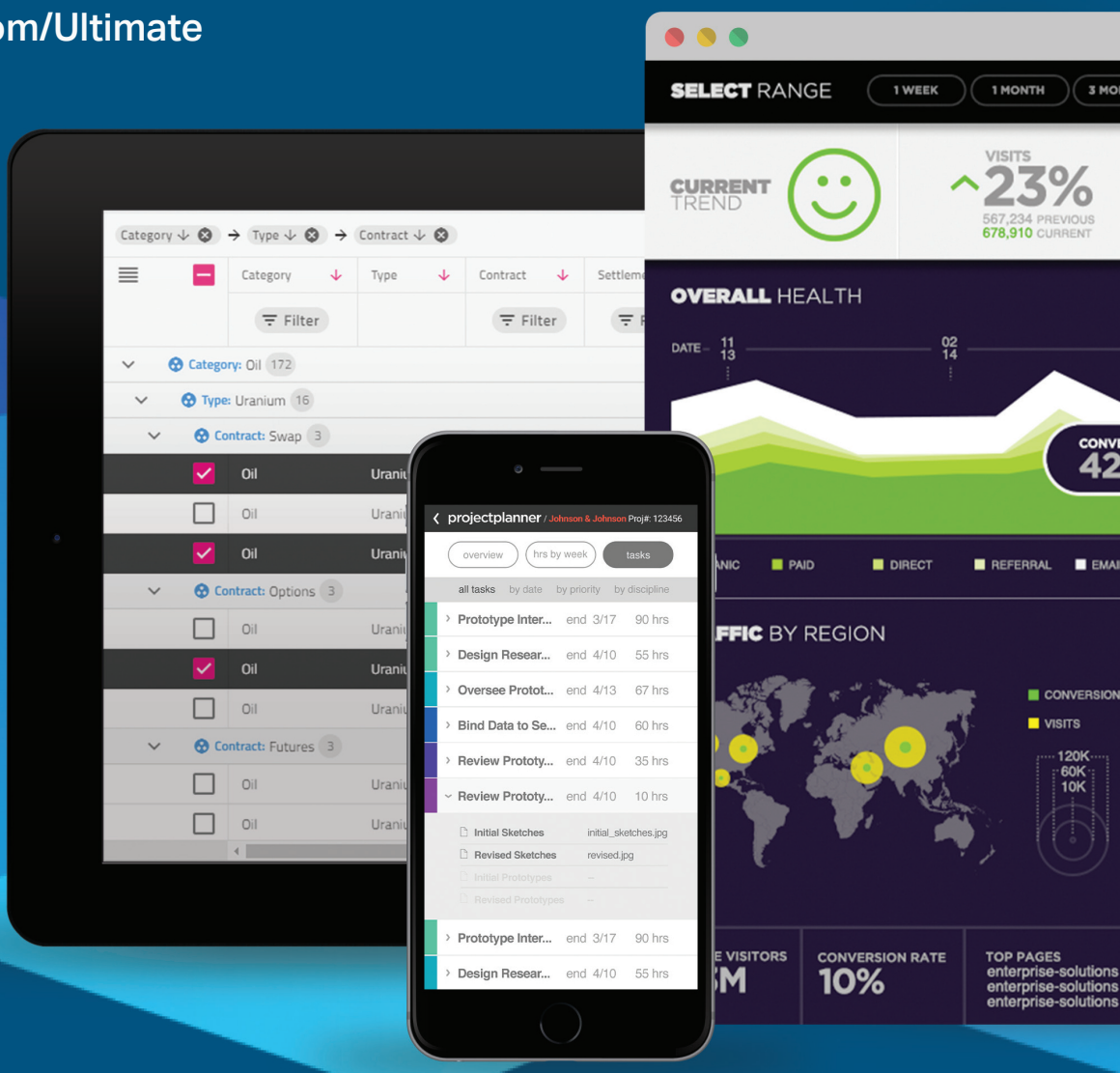
INFRAGISTICS

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 19.1

- Fastest **grids** & **charts** on the market – any device, any platform
- Build Spreadsheets with Charts in WPF, Windows Forms, Angular & JavaScript
- Get Angular code from Sketch designs with Indigo.Design
- 100% support for .NET Core 3

msdn magazine

JUNE 2019 VOLUME 34 NUMBER 6

ID STATEMENT MSDN Magazine (ISSN 1528-4859) is published 13 times a year, monthly with a special issue in November by 1105 Media, Inc., 6300 Canoga Avenue, Suite 1150, Woodland Hills, CA 91367. Periodicals postage paid at Woodland Hills, CA 91367 and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. $35.00, International $60.00. Annual digital subscription rates payable in U.S. funds are: U.S. $25.00, International $25.00. Single copies/back issues: U.S. $10, all others $12. Send orders with payment to: MSDN Magazine, File 2272, 1801 W.Olympic Blvd., Pasadena, CA 91199-2272, email MSDNmag@1105service.com or call 866-293-3194 or 847-513-6011 for U.S. & Canada; 00-1-847-513-6011 for International, fax 847-763-9564. POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: P.O. Box 201, Richmond Hill, ON L4B 4R5, Canada.

CORPORATE ADDRESS 1105 Media, Inc.
6300 Canoga Avenue, Suite 1150, Woodland Hills 91367
1105media.com

MEDIA KITS Direct your Media Kit requests to Chief Revenue Officer Dan LaBianca, 972-687-6702 (phone), 972-687-6799 (fax), dlabianca@Converge360.com

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

LIST RENTAL This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Jane Long, Merit Direct.
Phone: (913) 685-1301
Email: jlong@meritdirect.com
Web: meritdirect.com/1105

Reaching the Staff

Staff may be reached via e-mail, telephone, fax, or mail.

E-mail: To e-mail any member of the staff, please use the following form: FirstinitialLastname@1105media.com

Irvine Office (weekdays, 9:00 a.m.-5:00 p.m. PT)
Telephone 949-265-1520; Fax 949-265-1528
2121 Alton Pkwy., Suite 240, Irvine, CA 92606

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

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

Microsoft

BPA WORLDWIDE

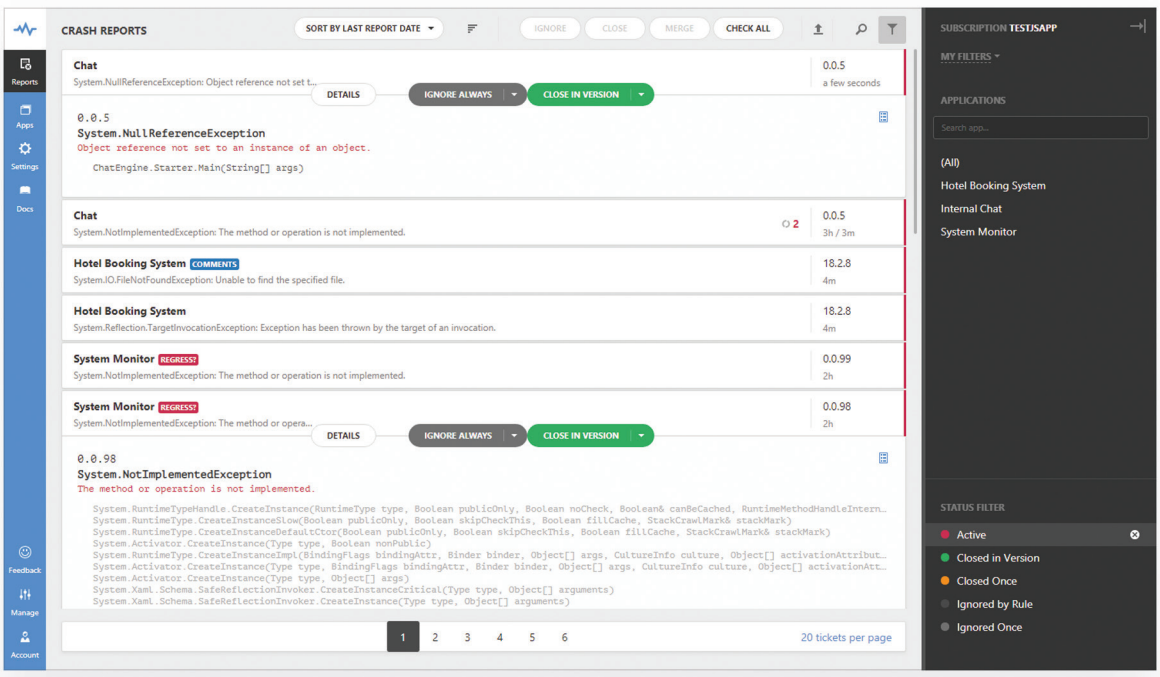
1906 2006 100 AMERICAN BUSINESS MEDIA

WPA Western Publications Association

logify™

Detect Errors and Exceptions
with 24/7 Application Monitoring

Logify's cloud-based application monitoring service allows you to automatically collect app crash events and runtime exceptions. With Logify, you will never have to deal with Visual Studio's® internal exception information. Logify presents all relevant exception data in an easy-to-understand, clutter-free manner. From loaded modules and cookies to browser info, OS build, user activity — Logify will organize results so you can address application issues in the shortest possible time.



Learn More Today — Try Logify Risk Free for 15 days

devexpress.com/logify

DevExpress®

MICHAEL DESMOND

A Failure of Process

Last month I wrote about the twin crashes of Boeing's new 737 MAX airliner and the automated Maneuvering Characteristics Augmentation System (MCAS) that precipitated both events. The investigations that followed point toward significant flaws with MCAS, which repeatedly commanded the aircraft nose down at low altitude as pilots struggled to diagnose the problem. ("Flight of Failure," msdn.com/magazine/mt833436).

Serious questions remain about MCAS and its behavior in the 737 MAX. For instance, MCAS was able to command 2.5 degrees of travel per increment on the rear stabilizer, yet the documentation submitted to the FAA showed a value of just 0.6 degrees. Likewise, MCAS was designed to activate based on data from a single angle of attack (AoA) sensor mounted on the fuselage of the plane. Yet, any system deemed a potential threat to controlled flight must activate based on multiple sensors.

> ARP-4754 informs the DO-178 process to determine the level of rigor required for software development around aircraft systems like MCAS.

What's concerning is that the integration of software into aircraft systems is strictly managed under guidelines such as DO-178B/DO-178C ("Software Considerations in Airborne Systems and Equipment Certification") and ARP-4754 ("Guidelines for Development of Civil Aircraft and Systems"). Both impose rigorous documentation and review requirements on manufacturers with an emphasis on safety. DO-178 focuses on software development across planning, development, verification, configuration management and quality assurance. ARP-4754 is a system-level process that addresses the integration of software and hardware systems and subsystems.

I spoke with a longtime aviation engineer. He says the layered levels of documentation, testing, review and validation defined in DO-178 and ARP-4754 should ensure that a subsystem like MCAS is implemented in a way that minimizes risk.

"It's all phases and gates," he says of the process. "There are checks and balances forward and backward. If a test fails we can go back up the line and see what the requirement was and see if the code was written badly or if the test case was designed badly. It's a very prescriptive, methodical process when we do this."

ARP-4754 informs the DO-178 process to determine the level of rigor required for software development around aircraft systems like MCAS. Failure hazard analysis performed under ARP-4754 enables engineers to define failure rates and risk factors for aircraft systems. This data is then used to calculate the Development Assurance Level (DAL) of systems like MCAS in DO-178. The five-point scale—from Catastrophic to No Effect—determines the severity of impact should a system fail in flight, and thus the required level of robustness in the code driving that system.

So what of the disparity between the documented and actual increment of the rear stabilizer? Perhaps field testing showed that MCAS required more authority. However, changing those specs should have kicked off another round of evaluation and testing to confirm MCAS behavior and failure states during flight, and mandated redundant sensor input to protect against inadvertent activation.

One of the emerging lessons of the 737 MAX catastrophe is that a process is only as good as the people and the institutions that execute it. Once undermined, it can lead to dangerous outcomes.

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

Intellifront BI | from $23,246.35

christiansteven

Cost-Effective Data Analytics & Business Intelligence.

- Design and serve visually stunning real-time reports using Grids, Charts, Gauges, Pies & more
- Add automatically updating KPI cards with aggregation, rounding, units, goals, colors & glyphs
- Create & assign Reporting, KPIs and Dashboards as consolidated "Canvases" to user groups
- Give your users a simple and intuitive self-service portal for their BI reports and documents
- Secure User Access with Active Directory Integration, Single Sign On & 2-Factor Authentication

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

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

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

LEADTOOLS
THE WORLD LEADER IN IMAGING SDKs

Powerful DICOM, PACS, and HL7 functionality.

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

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

JULIE LERMAN

EF Core in a Docker Containerized App, Part 3

In the past two articles, I've walked you through building a small ASP.NET Core API using Entity Framework encapsulated in a Docker container. In the first article, I included a SQLite database within the container. In the second, I targeted an Azure SQL Database that can be reached from anywhere. The bigger challenge in this case was how to keep secrets (such as database passwords) secret and configurations (such as connection strings to development or production databases) fluid. This involved learning how to leverage the ability of Docker Compose to read environment variables from the host.

There's another very interesting path for persisting data, which is to have both a database server and the data in containers. I've written about using SQL Server for Linux in Docker containers in earlier columns, such as the one at msdn.com/magazine/mt784660. I really like the way Microsoft's Steve Lasker expresses the perspective of using SQL Server in containers for dev and test:

> *Spinning up SQL in a container gives developers the ability to isolate/simulate an environment. Including a different version of SQL Server. Being able to start a test from the exact same state, each time, is goodness. Especially when testing a DB upgrade, instead of figuring out how to roll back the DB update, just restart the container.*

In the article on SQL Server for Linux, I interacted with the database using only sqlcmd at the command line. In a more recent article (msdn.com/magazine/mt832858), I used Azure Data Studio to interact with a containerized SQL Server and database. Now I want to integrate a dedicated, containerized SQL Server into my API development. I'll show you how to accomplish this in Visual Studio, continuing with the sample I used in the previous column.

Quick Review of the API

The API is simple with a single two-property Magazine class (Id and Name), a Magazines controller and an EF Core DbContext that uses migrations to seed new databases with four magazines.

The next iteration of the app, in Part 2 of the series (msdn.com/magazine/mt833438), introduced container orchestration via a docker-compose file. While that was a single-container solution that didn't truly need orchestration, docker-compose provided the ability to pass environment variables into the container without storing the values directly in the image. But even with single-container solutions, a compose file can be used to run the app on a Docker Swarm or Kubernetes cluster, giving you scale and self-healing.

Code download available at msdn.com/magazine/0619magcode.

The MagazinesContext class used the environment variables to inject a password to the database connection string.

Moving to a Containerized Database

The goal of this article is to target a SQL Server for Linux server and a database in containers on my development machine. This will mean updating the existing docker-compose file and updating connection strings within the app. In all, not a lot of effort. But if you've never done it before, it's certainly nice to have someone show you the way!

The biggest changes will be to the docker-compose.yml file. And what's super cool is that Docker will take care of the hardest pieces for you, simply by reading the instructions relayed in docker-compose. Because all of this starts in Visual Studio 2017, the included Tools for Docker will also participate when you run or debug from Visual Studio. If you're using Visual Studio 2019 (which, as I'm writing this, was officially released only yesterday, so I'll continue this series in 2017), the tooling is built in and the experience should be the same.

Here's the previous version of the docker-compose.yml file:

```
version: '3.4'

services:
  dataapidocker:
    image: ${DOCKER_REGISTRY-}dataapidocker
    build:
      context: .
      dockerfile: DataAPIDocker/Dockerfile
    environment:
      - DB_PW
```

This docker-compose file was only managing one image, defined in the dataapidocker service. The tooling will make sure

Figure 1 **The Docker-Compose File with Changes to Include an mssql/server Container**

```
version: '3.4'

services:
  dataapidocker:
    image: ${DOCKER_REGISTRY-}dataapidocker
    build:
      context: .
      dockerfile: DataAPIDocker/Dockerfile
    environment:
      - DB_PW
    depends_on:
      - db
  db:
    image: mcr.microsoft.com/mssql/server
    environment:
      SA_PASSWORD: "${DB_PW}"
      ACCEPT_EULA: "Y"
    ports:
      - "1433:1433"
```

that the DOCKER_REGISTRY variable is replaced at run time with the Docker engine running on my development machine. Then, using that image, it will find the Dockerfile (defined in both Part 1 and Part 2 of this series) to get further instructions about what to do with the image when the container instance is created. Because I didn't provide a value for the DB_PW environment variable directly in the docker-compose file, it allows me to pass a value from the shell where I'm running the container or from another source, such as a docker .env file. I used an .env file in Part 2 to store the key-value pair of DB_PW and my password, *eiluj*.

So now I'm going to tell docker-compose that I want to also have it spin up a SQL Server container. The SQL Server for Linux image is an official image in the Microsoft Container Registry (MCR), though it's also listed on Docker Hub for discoverability. By referencing it here, Docker will first look in the local registry (on the dev machine where I'm working) and if the image isn't found there, it will then pull the image from the MCR. See **Figure 1** for an example of these changes.

But the new service I added, which I named db, does more than just point to the mssql/server image. Let's, as they say, unpack the changes. Keep in mind that there's another modification coming after I work through this step.

The first change is within the dataapidocker service—the original one that describes the container for the API. I've added a mapping called *depends_on*, which contains what YAML refers to as a sequence item, named db. That means that before running the dataapidocker container, Docker will check the docker-compose file for another service named db and will need to use its details to instantiate the container defined by that service.

> The SQL Server for Linux image is an official image on Docker Hub.

The db service begins by pointing to the SQL Server for Linux image using its official Docker name. This image requires that you pass in two environment variables when running a container—SA_Password and ACCEPT_EULA—so the service description also contains that information. And, finally, you need to specify the port that the server will be available on: 1433:1433. The first value refers to the host's port and the second to the port inside the container. Exposing the server through the host's default port 1433 makes it easy to access the database from the host computer. I'll show you how that works after I've gotten this project up and running.

Figure 2 **Defining a Data Volume in Docker-Compose**

```
services:
  dataapidocker: [etc]
  db:
    image: mcr.microsoft.com/mssql/server
    volumes:
      - mssql-server-julie-data:/var/opt/mssql/data
    environment:
      SA_PASSWORD: "${DB_PW}"
      ACCEPT_EULA: "Y"
    ports:
      - "1433:1433"
volumes:
  mssql-server-julie-data: {}
```

When it's time to run this docker-compose *outside* Visual Studio, I'll also need to expose ports from the dataapidocker service. The Visual Studio tooling created a second docker-compose file, docker-compose.override.yml, that Visual Studio uses during development. In that file are a few additional mappings, including the ports mapping (ports: - 80) for the dataapidocker service. So for now I'll let the tooling take care of allowing me to browse to the Web site when debugging in Visual Studio.

> Exposing the server through the host's default port 1433 makes it easy to access the database from the host computer.

Defining a Separate Volume for Persistent Data

There's still more work to be done in docker-compose, however. With the existing description, any databases and data will be created inside the same container that's running the SQL Server. This is probably not a problem for testing if a clean database is desired for each test. It's better practice, however, to persist the data separately. There are a few ways to do this, but I'll be using what's called a data volume. Take a look at my blog post at bit.ly/2pZ7dDb, where I go into detail about data volumes and demonstrate how they persist even if you stop or remove the container running the SQL Server.

You can leave instructions in the docker-compose file to specify a volume for persisting the data separately from the SQL Server container, as I've done in **Figure 2**. A volume isn't the same as a container, so it isn't another service. Instead, you create a new key called volumes that's a sibling to services. Within the key, you provide your own name for the volume (I've called mine mssql-server-julie-data). There's no value associated with this key. Naming a volume this way allows you to reuse it with other containers if you need. You can read more about volumes in the Docker reference for docker-compose at docs.docker.com/compose/compose-file or, for more detail, check out Elton Stoneman's Pluralsight course on stateful data with Docker (pluralsight.pxf.io/yoLYv).

Notice that the db mapping also has a new mapping for volumes. This mapping contains a sequence item in which I've mapped the named volume to a target path in the source container where the data and log files will be stored.

Setting Up the Connection String for the Containerized SQL Server

As a reminder, in the previous version of the app, I left a connection string to SQL Server LocalDB in appsettings.Development.json for times I want to test the API running in the Kestrel or local IIS server. The connection string to the Azure SQL database, named ConnectionStrings:MagsConnectionMssql, is stored in an environment variable in the API's Dockerfile, just above the section that defines the build image. Here are the first few lines of that Dockerfile. The connection string has placeholders for the user id

and password. The user id is still in the Dockerfile only because I was focusing on hiding the password and haven't yet removed it:

```
FROM microsoft/dotnet:2.2-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 80

ENV ConnectionStrings:MagsConnectionMssql="Server=[Azure SQL endpoint];
    Initial Catalog=DPO419Mags;Persist Security Info=False;
    User ID=ENVID;Password=ENVPW;MultipleActiveResultSets=False;
    Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
ENV DB_UserId="lerman"

FROM microsoft/dotnet:2.2-sdk AS build
```

This connection string can easily be moved out to docker-compose.yml and controlled by external means, as well. The API's Startup Configuration method overwrites the user and password placeholders with the values of provided environment variables, including the password DB_PW that I stored in the.env file whose contents look like this:

```
DB_PW=eiluj
```

The .env file and its DB_PW value will be read by docker-compose and passed into the container for the Dockerfile to read and make available to the app. In the end, I'm removing both the connection string and DB_UserId variables from Dockerfile.

But that connection string was for Azure SQL. I now need to change its value to point to the SQL Server container:

```
ENV ConnectionStrings:MagsConnectionMssql
  "Server=db;Database=DPO419Mags;User=sa;Password=ENVPW;"
```

I'm using the same variable name because that's what Startup.ConfigureServices is expecting. What's interesting to me is that the container runtime will understand the name of the db service defined in docker-compose as the server name. So, in the connection string, I can specify the server as db. The default user that the mssql/server image uses to set up the server is sa, so I've coded that directly into the connection string, which is why I no longer need an ENV variable for the user id. However, I'm still using a placeholder, ENVPW, for the password value.

As with the sample in the previous column, the placeholder will ultimately be replaced by the value of the DB_PW environment variable. Here's what the code now looks like that reads the connection string and the DB_PW variable, then updates the connection string before passing it to the SQL Server provider:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
  var config = new StringBuilder(
    Configuration["ConnectionStrings:MagsConnectionMssql"]);
  string conn = config.Replace("ENVPW", Configuration["DB_PW"])
                      .ToString();
  services.AddDbContext<MagContext>(options => options.UseSqlServer(conn));
}
```

Those are all the needed changes! Now let's debug and see what happens.

Debugging with the Database Container

Be sure that the docker-compose project is the startup project in the solution and that the debug button is set to Docker Compose, then hit F5 to debug the API.

> The SQL Server image isn't small—it's 1.45GB. So, the first time you pull it, be patient.

If this is the first time you're running the app this way and you've never pulled the mssql/server image, that will be the first thing to happen. The SQL Server image isn't small—it's 1.45GB. So, the first time you pull it, be patient. Note that it requires 2GB of RAM, so be sure to assign enough RAM to your Docker engine (in the Docker settings) to run it. Even so, this is still faster than installing SQL Server on your machine, and it only uses resources already assigned to Docker, not additional resources on your host machine. And once the image is there, it's breezy to spin up lots of instances. Another benefit is that any updates to the base image get pulled as small layers, not the entire thing again. Note that the Microsoft/dotnet sdk and aspnetcore-runtime images are already on my system, thanks to the work I did in the previous columns.

If you watch the build output, you'll see that the app creates the data volume prior to handling the mssql/server container—in my case "handling" means first pulling it from Docker Hub because it's new to my system. This order of operations occurs because the server is dependent on the volume, so the volume gets taken care of first. Once that's done, it starts building up the API's image if needed and then spins up the containers. The steps are clearly relayed in the build output, thanks to the tooling.

As with the previous versions of this app, the results of the GET method of the Magazines controller are displayed in a browser window.

```
C:\Users\Julie>docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
dataapidocker                       dev                 3d015ad168af        2 minutes ago       271MB
microsoft/dotnet                    2.2-sdk             f9f28101464e        3 weeks ago         1.74GB
microsoft/dotnet                    2.2-aspnetcore-runtime 78bd7dc97547     3 weeks ago         271MB
docker4w/nsenter-dockerd            latest              2f1c802f322f        5 months ago        187kB
mcr.microsoft.com/mssql/server      latest              885d07287041        6 months ago        1.45GB
```

Figure 3 **Inspecting the Images**

```
C:\Users\Julie>docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED
9820ea50fd98        dataapidocker:dev              "tail -f /dev/null"      4 minutes ago
82656007fb6a        mcr.microsoft.com/mssql/server "/opt/mssql/bin/sqls…"   4 minutes ago

  STATUS              PORTS                    NAMES
  Up 4 minutes        0.0.0.0:32794->80/tcp    dockercompose4124429769409226297_dataapidocker_1
  Up 4 minutes        0.0.0.0:1433->1433/tcp   dockercompose4124429769409226297_db_1
```

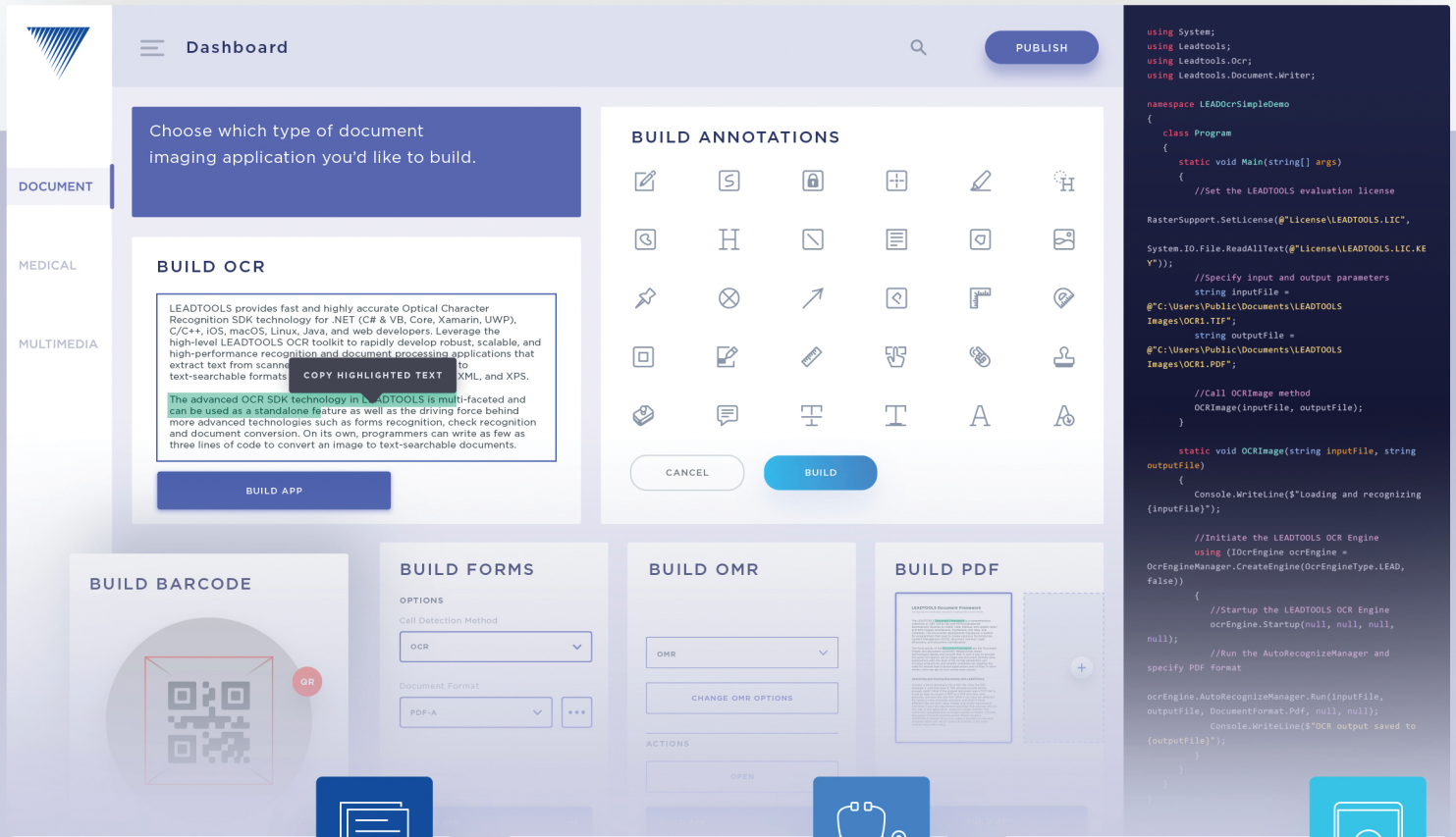
Figure 4 **The Running Containers**

```
C:\Users\Julie>docker volume ls
DRIVER              VOLUME NAME
local               dockercompose4124429769409226297_mssql-server-julie-data
```

Figure 5 **The Docker Volume Command**

LEADTOOLS®

BUILD BETTER APPS WITH LEADTOOLS



DOCUMENT

- Document Viewer & Converter
- OCR, MICR, OMR & ICR
- 1D & 2D Barcode
- Forms Recognition
- PDF, DOCX, HTML, SVG, RTF, TXT
- Annotations & TWAIN

MEDICAL

- DICOM, CCOW & HL7
- PACS Client & Server Framework
- DICOMWeb (WADO)
- Web & Desktop Viewers
- Image Processing & Annotations
- Medical 3D (MPR, MIP, VRT)

MULTIMEDIA

- Play, Capture, Convert & DVR
- Media Streaming
- MPEG-2 TS, RTSP, HTML5 & more
- OGG, FLV, ISO, AVI, WebM & more
- Audio & Video Processing
- DirectShow & Media Foundation

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

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

Exploring the Containers Created by docker-compose

Now let's see what's going on with Docker. I'll do that at the command line, although the Docker extension for Visual Studio Code is another cool option for achieving this.

First, I'll list the images with the command *docker images*. If you're new to using the Docker CLI, you may prefer to use the more explicit command *docker image ls*. As you can see in **Figure 3**, this shows that the mssql image is now on my machine and that a new version of my dataapidocker image was created, as well.

Next, I'll check out the containers with *docker ps*, the shortened equivalent of *docker container ls*. This output is wide so I'm showing the left half of the screen above the right half in **Figure 4**.

And, finally, to see the volume that was created, which is now tied to the db container, I can use the *docker volume ls* command, as shown in **Figure 5**.

Notice that the names of both containers and the data volume are prefaced with the same text generated by the Visual Studio debug process in combination with the tooling.

Remember that I've exposed the database from the container on the host's port 1433 so I can use tools on my development machine to look at the server that's inside the db container, along with the databases in the attached volume. I'll use Azure Data Studio to do this; **Figure 6** shows how I defined the connection.

> Having your apps encapsulated in Docker images makes it so easy to deploy the apps and all of their dependencies as containers.

With the connection made, I can then interact with the server and my new database. **Figure 7** shows the database explorer on the left, along with a query and results on the right.

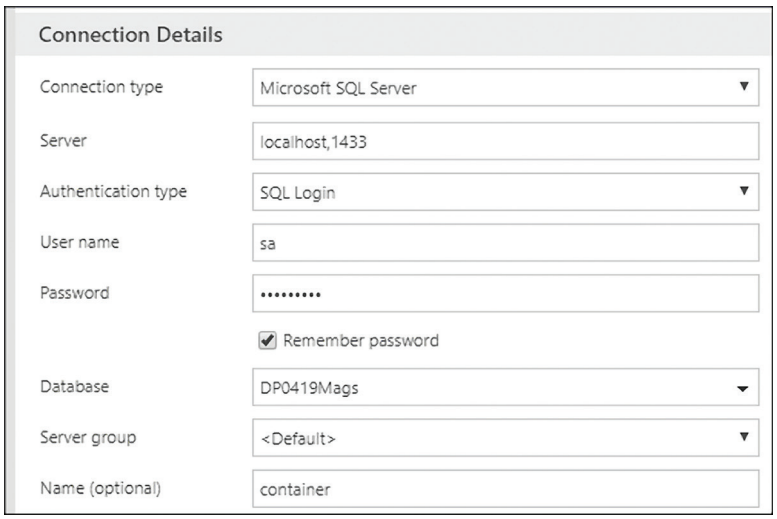


Figure 6 **Setting Up a Connection to My Containerized Database**

Your App's Containers Are Ready for Action

Having your apps encapsulated in Docker images makes it so easy to deploy the apps and all of their dependencies as containers. But for container newbies and veterans alike, I think that having the tools to develop apps so that they're already associated with images and orchestration makes leveraging containers so much more straightforward. And in this series I've focused on development and local debugging only. Following Part 2, I published a blog post about deploying the single-image solution to Azure for testing at bit.ly/2CR40x3. That solution stores its data in Azure SQL Database and I run the test container using Azure Container Instances. I also wrote a blog post about publishing the dataapidocker image from this article to Docker Hub and then hosting the full solution in a Linux virtual machine on Azure, which is another great way to test your orchestrated containers. You can find that blog post at bit.ly/2VBaN8M. ■

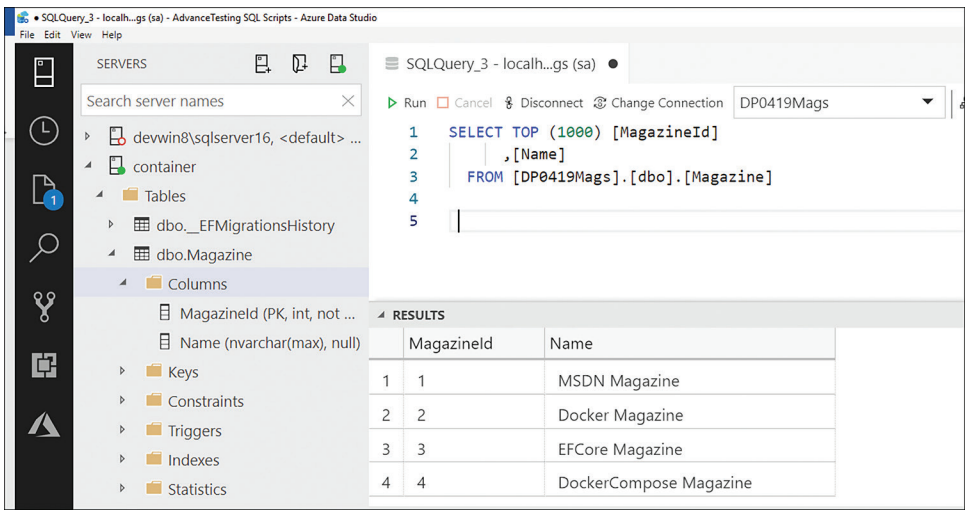


Figure 7 **Interacting with the Containerized Database in Azure Data Studio**

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 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 bit.ly/PS-Julie.*

Thanks to the following technical experts for reviewing this article: *Elton Stoneman (Docker), Travis Wright (Microsoft)*

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

TED NEWARD

Coding Naked: Naked Acting

Welcome back, NOFers. Last time, I examined how NakedObjects handles collections, which provide the ability to associate a number of objects with one another (msdn.com/magazine/mt833439). As I went through that piece, however, I introduced the concept of an "action" into the code without really explaining it in any detail. Because actions are the "code" pairing to properties' "data," it makes sense to go over actions in a bit more detail—and in particular, note the places where actions can live and how they appear in the UI.

Ready to act naked?

Actions

The NOF manual describes an action as "a method that is intended to be invoked by a user." It's important to note, though, that these methods aren't just user-invokable; the manual goes on, in the same sentence, to point out that "it may also be invoked programmatically from within another method or another object." Actions, then, are where behavior on an object is defined, and NOF will generate a menu item for the user to click/invoke for each action it discovers.

Actions, then, are where behavior on an object is defined, and NOF will generate a menu item for the user to click/invoke for each action it discovers.

One such action that easily comes to mind is the idea of performing the basic CRUD behaviors on a given Talk (or Speaker, for that matter), but recall that NOF already handles those as part of the core behavior of the UI as a whole. A menu item off the main menu allows you to create a new Speaker, which, remember, is defined on the SpeakerRepository, as you saw in the second article in this series (msdn.com/magazine/mt833268). Editing that speaker happens after a Speaker is selected and the user selects the "Edit" menu item that appears. In fact, the only thing missing from the list is the ability to delete a Speaker, so that's an easy place to start—I'll now add an Action that allows me to delete the currently visible Speaker.

(I know, I know—who would ever want to delete a Speaker? We're all so lovable, it's hard to imagine that anyone would ever want to remove a Speaker from the system. But go with me on this one, for pedagogical purposes if nothing else. Imagine they're retiring. Or something.)

To add a new action, I only need to add a new method to the Speaker type. By default, any public method on a domain object will be exposed as an action in the "Actions" menu of the UI, so adding this is pretty straightforward:

```
public void Delete()
{
  Container.DisposeInstance(this);
}
```

Recall that the Container property is an instance of the IDomainObjectContainer that's dependency injected onto each Speaker after it's created or returned from the SpeakerRepository. To remove a persisted object (such as a Speaker that was added with the CreateNewSpeaker action on the SpeakerRepository), the Container has to "dispose" of that instance.

However, when I add this to the Speaker class and run it, something bad happens—specifically, after deleting the Speaker, I get an error about a missing object. The NOF client is complaining that the domain object it was just displaying—the Speaker that was just deleted—no longer exists. That makes a ton of real sense, when you think about it, and the answer is to give the UI something else to display, such as the list of all the remaining Speakers in the system, which I can get easily enough from the Container again, like so:

```
public IQueryable<Speaker> Delete()
{
  Container.DisposeInstance(this);
  return Container.Instances<Speaker>();
}
```

Now, when the user selects to delete a Speaker, the UI removes them, shows the remainder of the list of Speakers, and life moves on. (Farewell, sweet Speaker. We remember you fondly.)

One problem that emerges, however, is that certain domain objects have a clear dependency on others within the system, and arbitrarily destroying those objects without thinking about those associations will sow chaos. (In fact, NOF will prevent deletion of an object that has any associated objects until those associated objects are deleted first.) In practical terms, this means if the Speaker has any Talks created, deleting the Speaker will leave those Talks orphaned. Ideally, you wouldn't allow Speakers who have one or more Talks created to be deleted. You could put a check inside the Delete method, and provide an error message if the user selects it and you can't follow through, but it's generally a better idea to disallow a user the option of even selecting something they're not allowed to do—in NOF parlance, you want to disable the action entirely.

Once again, NOF uses naming conventions to provide this kind of functionality. By providing a method that begins with the name "Disable" and continues with the name of the action you want to disable (in this case "Delete"), you can do that check long before the user ever selects it:

```
public string DisableDelete()
{
  if (Talks.Count > 0)
  {
    return "Speakers cannot be deleted until their Talks are removed.";
  }
  return null;
}
```

Notice that DisableDelete returns a string, which will either be null if the action is fine to execute, or will be displayed to the user. (By default, the menu item for the action will be disabled, so it won't be selectable by the user, but remember that actions can be invoked in other ways.)

What if you want to hide the action from the user entirely? That's a different question, but NOF uses the same approach—a HideDelete method—to indicate whether the action should be displayed. The choice between hidden and disabled is obviously a subtle one, and probably a UX debate that others are more qualified to have than I; suffice it to say, choose as your UX designer or your heart takes you.

> NOF is committed to the principle that the UI should be built from the structure of the code itself.

Action Parameters

Keep in mind that, in many cases, the action will require additional data before it can be carried out. An action may want to find all Speakers who have Talks that contain a particular search term, such as "Blockchain" or "IoT" (hopefully to remove them entirely from the system), which means the user has to have an opportunity to type in that search term. Again, NOF is committed to the principle that the UI should be built from the structure of the code itself, which means in this case that the UI will examine the parameters of the method and present a UI element that provides the user the chance to put the necessary input in place. You've seen this mechanism at work already, in that the FindSpeakerByLastName method of the SpeakerRepository does the same thing: accepts a string parameter for the last name for which to search, which NOF interprets to mean it needs to display a textbox after that action is invoked.

Naked Contributions

Recall from the earlier article I mentioned that the Naked Objects world consists of both domain objects and services, where services are collections of behavior that don't necessarily belong on the domain object. Essentially, if a method on a service takes a domain object as one of its parameters, and that parameter has a "ContributedAction" attribute on it, the NOF UI takes that to mean that this action should appear on the domain object's menu.

Imagine for a moment that you want the code to notify Speakers that their talk was accepted at the conference to be outside the Speaker object. (This makes sense, actually, from a domain modeling perspective, because notifying a Speaker has really nothing to do with being a Speaker.) You create a service, then, that looks something like this (taking care to make sure it's registered with NOF in the Services property of the NakedObjectsRunSettings class, as well):

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

  public void AcceptTalk([ContributedAction] Talk talk)
  {
    Container.InformUser("Great! The Speaker will be emailed right now");
  }
}
```

The implementation of the notification, of course, will take more than just showing a message to the user, but that's not the important part—the key here is the ContributedAction attribute on the Talk parameter, which will cause this action to appear in the Talk list of actions. This provides a great deal of flexibility regarding where code appears in the system, compared to where it appears in the UI.

Note that it's possible to contribute actions to groups of domain objects, as well, using the same kind of convention, only applying it to a parameter that's a collection of domain objects rather than a single one. For example, if I want to send notifications to a collection of Speakers, I can do so from a given method, like this:

```
public void NotifySpeakers([ContributedAction] IQueryable<Speaker>
speakers)
{
  foreach (Speaker s in speakers)
  {
    // Send SMS or email or whatever
  }
}
```

From a UI perspective, any collection of Speakers will now have a checkbox appear in front of each one (allowing the user to select which of the Speakers will appear in the collection), and the "Actions" menu presented in the table will have the "Notify Speakers" action displayed. As with all actions, if additional input from the user is required (such as the message to send each Speaker), this will appear as part of the UI before the method is executed.

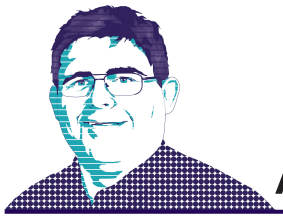
Wrapping Up

Actions represent an important aspect of the Naked Objects Framework experience, in that they're where the vast majority of the "business rules" of any application will sit. Services can provide opportunities to host business logic, to be sure, but it's inside actions on a domain object that domain-driven developers will find the best opportunities to provide the code to "do the work."

Happy coding! ■

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

Thanks to the following technical expert for reviewing this article: *Richard Pawson*



FRANK LA VIGNE

Exploring Data with R

Since the very first Artificially Intelligent column, all the code samples I've provided have been in Python. That's because Python currently reigns as the language of data science and AI. But it's not alone—languages like Scala and R hold a place of prominence in this field. For developers wondering why they must learn yet another programming language, R has unique aspects that I've not encountered elsewhere in a career that's spanned Java, C#, Visual Basic, Python and Perl. With R being a language that readers are likely to encounter in the data science field, I think it's worth exploring here.

R itself is an implementation of the S programming language, which was created in the 1970s for statistical processing at Bell Labs. S was designed to provide an interactive experience for developers who at the time worked with Fortran for statistical processing. While we take interactive programming environments for granted today, it was revolutionary at the time.

R was conceived in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and derives its name from the first initial of its creators, while also playing on the name of S. Version 1.0.0 of R was released in 2000 and has since enjoyed wide adoption in research departments thanks in part to its wide array of built-in statistical algorithms. It's also easily extensible via functions and extension packages.

A robust developer community has emerged around R, with the most popular repository for R packages being the Comprehensive R Archive Network (CRAN). CRAN has various packages that cover anything from Bayesian Accrual Prediction to Spectral Processing for High Resolution Flow Infusion Mass Spectrometry. A complete list of R packages available in CRAN is online at bit.ly/2DGjuEJ. Suffice it to say that R and CRAN provide robust tools for any data science or scientific research project.

Getting Started with R

Perhaps the fastest way to run R code is through a Jupyter Notebook on the Azure Notebook service. For details on Jupyter Notebooks, refer to my February 2018 article on the topic at msdn.com/magazine/mt829269. However, this time make sure to choose R as the language when creating a new notebook. The R logo should appear on the top right of the browser window. In a blank cell, enter the following code and execute it:

```
# My first R code
print("hello world")
x <- 3.14
y = 1.21
x
y
```

Code download available at bit.ly/2vwC0L0.

The output should read the traditional "hello world" greeting, as well as the values 3.14 and 1.21. None of this should come as novel or unique to any software developer. Note that the assignment operator can also be "<-" and not just the more commonly used equals sign. Both are syntactically equal. Also take note that the # character introduces a comment and applies to the rest of the line.

Vectors are one-dimension arrays that can hold numeric data, character data or logical data. They're created with the c function. The c stands for "combine." Enter the following into a new cell and execute it:

```
num_vec <- c(1,2,3.14) # numeric vector
char_vec <- c("blog","podcast","livestream") # character vector
bool_vec <- c(TRUE,TRUE,FALSE) #logical vector
#print out values
num_vec
char_vec
bool_vec
```

The values displayed should match the values set in the code. You may now be wondering if vectors can contain mixed types. Enter the following code into a new cell:

```
mix_vec <- c(1,"lorem ispum",FALSE)
mix_vec
```

While the code does run, sharp-eyed readers will notice single quotes around each element in the vector. This indicates that the values were converted to character values. R has the typeof function to check the type of any given variable. Enter the following code to inspect the vectors already created:

```
typeof(num_vec)
typeof(char_vec)
typeof(bool_vec)
typeof(mix_vec)
```

One other useful function to know is ls, which displays all the objects in the current working environment. Enter "ls()" into a new cell, execute it, and observe that the output contains the four vectors just defined, along with the x and y variables defined in the first cell.

Working with Data

The best way to experience the true power and elegance of the R language is by using it to explore and manipulate data. R makes it easy to load datasets and quickly get an understanding of their dimensions, structure and statistical properties. For the next few examples, I'll use a dataset that's near and dear to me: basic statistics on my blogging activity. I've run and maintained a technology blog

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

since 2004 and have kept basic statistics on how frequently I posted each month. Additionally, I have added the number of days in each month and the average post per day value (PPD). PPD is the number of posts in a given month divided by the number of days in that month. I have placed the CSV file in the project library on the Azure Notebook Service at bit.ly/2V76d2G.

Enter the following code into a new cell to load the data into an R data frame, a tabular data structure with columns for variables and rows for observations, and display the first six and the last three records using the head and tail functions, respectively, like so:

```
postData <- read.csv(file="franksworldposts.csv", header=TRUE, sep=",")
head(postData)
tail(postData, 3)
```

Using the str function, I can view the basic structure and data types of the DataFrame. Enter the following code into a new cell:

```
str(postData)
```

The output should reveal that the DataFrame has 183 observations, or rows, and consists of four variables, or columns. The Posts and Days.in.Month variables are integers, while the PPD is a numeric type. The Month variable is a factor with 183 levels, where factor is a data type that corresponds to categorical variables in statistics. Factors are the functional equivalent to categorical in Python Pandas and can be either strings or integers. They're ideal for variables with a limited number of unique values, or, in R terms, levels. In this DataFrame, the Month field represents a month between February 2004 and April 2019. As dates do not repeat, there are no duplicate categorical values.

Now that my data is loaded, I can sort and query it to explore it further. Perhaps I can glean some insights. For instance, if I wanted to view the top-10 months where I was most productive on my blog, I could perform a descending sort on the Posts column. To do so, enter the following code into a new cell and execute it:

```
sortedPostData <- postData[order(-postData$Posts),]
head(sortedPostData, 10)
```

The top-10 most active months have all been within the last three years. To explore the data set further, I can perform a filtering operation to determine which months have had 100 or more posts. In R, the subset function does just that. Enter the following code to apply this filter and assign the output to a new DataFrame called over100, like so:

```
over100 <- subset(postData, subset = Posts >= 100)
over100
```

The results look similar to the previous output of the top 10. To check the count of rows, use the nrow function to count the number of rows in the DataFrame, like this:

```
nrow(over100)
```

The output indicates that there are 11 rows where there were 100 or more blog posts in a given month. With 100 posts, May 2005 just missed the top-10 most active months, falling into 11th place. Clearing the 100-posts-per-month threshold wasn't a milestone I would reach again for 11 years. Is there a pattern of starting the blog with intensity only to have it fade out and then pick it up again? Let's examine the data further.

Now would be a good time to explore how to view individual rows and columns in a DataFrame. For example, to view the first row in the DataFrame, enter the following code to view the contents of the entire row:

```
postData[1,]
```

Note that the index for the DataFrame starts at 1 and not 0, as in most other programming languages. To view just the Posts field for the first row, enter the following code:

```
postData[1,2]
```

To view all the values in the Posts field, use the following line of code:

```
postData[,2]
```

Alternatively, you may also use the following syntax to display the columns based on their name. Enter the following line of code and confirm that its output matches the output from the line prior:

```
postData$Posts
```

As R has its roots in statistical processing, there are many built in functions to view the basic shape and properties of the data. Use the following code to get a better understanding of the data in the Post column:

```
mean(postData$Posts)
max(postData$Posts)
min(postData$Posts)
summary(postData$Posts)
```

Now, compare this to the PPD column, like so:

```
mean(postData$PPD)
max(postData$PPD)
min(postData$PPD)
summary(postData$PPD)
```

From the data we see that the number of posts vary from one per month all the way to 225 over the course of 15 years. What if I wanted to explore only the first year? Enter the following code to display only the records for the first year of blogging, along with statistical summaries for the Post and PPD fields:

```
postData[1:12,]
summary(postData[1:12,2]) # Posts
summary(postData[1:12,4]) # PPD
```

While the numbers here tell a story, very often a graph will reveal more about trends and patterns. Fortunately, R has rich graph plotting capabilities built in. Let's explore those.

Visualizing Data

Creating plots in R is very simple and can be done with a single line of code. Let's start by using the post counts and PPD values for the first year. Here's the code to do that:

```
plot(postData[1:12,2], xlab="Month Index", ylab="Posts",
  main="Posts in the 1st Year")
plot(postData[1:12,4], xlab="Month Index", ylab="PPD",
  main="PPD in the 1st Year")
```

The output should resemble **Figure 1**.

For the first year of blogging, the graph shows that post activity steadily grew the first year with a steep growth curve between the third and sixth months. After a late summer dip, 2004 finished up strong. Additionally, the graphs reveal that there's high correlation between the number of posts in a month and the number of posts per day. While this may be intuitive, it's interesting to see it displayed in graph form.

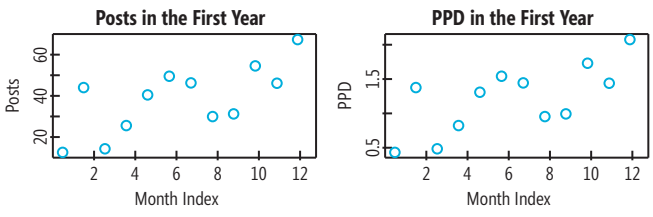


Figure 1 **Plotting the Posts and PPD Columns**

Now, I would like to see a graph of blog posts over the entire 15-year span and see if a pattern emerges over a longer period of time. Enter the following code to graph the entire timespan:

```
plot(postData[,2], xlab="Month Index", ylab="Posts",
  main="All Posts")
```

The results, shown in **Figure 2**, do show a clear trend, if not a well-defined pattern. Blogging activity started out fairly strong but declined steadily, picking up again around 30 months ago. The trend of late is decidedly upward. There's also the one significant outlier.

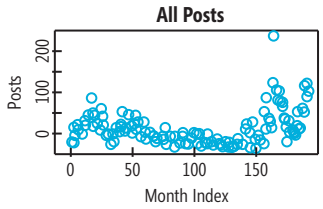


Figure 2 **Posts Over 15 Years**

Correlation Matrix

Earlier, I noted a correlation between the Posts and PPD columns. R has a built-in function to display a correlation matrix, which is a table displaying correlation coefficients between variables. Each cell in the table shows the correlation between two variables.

A correlation matrix quickly summarizes data and reveals relationships between variables. Values closer to 1 have a high correlation, while those closer to 0 have low correlation. Negative values indicate a negative correlation. To view the correlation matrix for the postData DataFrame, it's first necessary to isolate the numeric fields into their own DataFrame and then call the cor function. Enter the following code into a new cell and execute it:

```
postsCor <- postData[, c(2, 3 ,4)]
cor(postsCor)
```

The output reveals a near-perfect correlation between Posts and PPD, while Days.In.Month has a slightly negative correlation to PPD.

Wrapping Up

While R's syntax and approach may differ from traditional programming languages, I find it an elegant solution for data wrangling and mathematical processing. For software engineers serious about building a career in data science, R is an important skill to develop.

In this article, I explored some of the fundamentals of the R programming language. I showed how to use built-in functions to load and explore data within DataFrames, to gain insights through statistics, and to plot graphs. In fact, everything in this article was written in what would be referred to as "base" R, as it doesn't rely on any third-party packages. However, some R users prefer the "tidyverse" suite of packages, which uses a different style. I'll explore that in an upcoming column. ■

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 technical expert for reviewing this article: *Andy Leonard, David Smith*

AI-Powered Biometric Security in ASP.NET Core

Stefano Tempesta

This article, in two parts, introduces the policy-based authorization model in ASP.NET Core 3, which aims to decouple authorization logic from the underlying user roles. It presents a specific example of this authorization process based on biometric information, such as face or voice recognition. In this case, access to a building is restricted when an unauthorized intrusion is detected. The severity of the intrusion is assessed by an anomaly detection service built into Azure Machine Learning.

Site Access

The context is an extremely secured site—think of a military area, or a hospital, or a datacenter. Access is restricted to authorized people, with some limitations. The following steps describe the security flow enforced at the door of each building to check people in:

1. A person requesting access to a building swipes their access pass on the door's card reader.
2. Cameras detect motion and capture the face and body of the person; this should prevent the use of a printed photo, for example, to trick the camera with face-only recognition.
3. The card reader and cameras are registered as Internet of Things (IoT) devices and stream recorded data to Azure IoT Hub.
4. Microsoft Cognitive Services compares the person against a database of people authorized to access the building.
5. An authorization flow matches the biometric information collected by the IoT devices with the identity of the person on the access pass.
6. An Azure Machine Learning service is invoked to assess the risk level of the access request, and whether it's an unauthorized intrusion.
7. Authorization is granted by an ASP.NET Core Web API by checking for specific policy requirements owned by the profile defined in the previous steps.

If there's a mismatch between the detected identity of the person and the access pass, access to the site is blocked immediately. Otherwise, the flow continues by checking whether any of the following anomalies have been encountered:

- Atypical frequency of access to the building.
- Whether the person has exited the building earlier (check out).
- Number of accesses permitted per day.

Some of the technology discussed in this article is still in preview. All information is subject to change.

This article discusses:

- Site access security flow
- Authorization in ASP.NET Core
- Face and voice recognition using Microsoft Cognitive Services

Technologies discussed:

ASP.NET Core 3, Microsoft Azure Cognitive Services Vision and Speech APIs

Code download available at:

bit.ly/2IXPZCo

- Whether the person is on duty.
- Criticality of the building (you may not want to restrict access to a canteen, but enforce a stricter policy for access to a server datacenter).
- Whether the person is bringing someone or something else along.
- Past occurrences of similar access typologies to the same building.
- Risk-level changes measured in the past.
- Number of intrusions detected in the past.

The anomaly detection service runs in Azure Machine Learning and returns a score, expressed as a likelihood that the access is a deviation from the standard, or not. The score is expressed in a range between zero and one, where zero is "no risk detected," all good, full trust granted; and one is "red alert," block access immediately! The risk level of each building determines the threshold that's considered acceptable for allowing access to the building for any value greater than zero.

Training is an asynchronous process.

Authorization in ASP.NET Core

ASP.NET Core provides a simple authorization declarative role and a rich policy-based model. Authorization is expressed in requirements, and handlers evaluate a user's claims against those requirements. For the purpose of authorizing users to access a site, I'll describe how to generate custom policy requirements and their authorization handler. For more information about the authorization model in ASP.NET Core, please refer to the documentation at bit.ly/2UYZaJh.

As noted, a custom policy-based authorization mechanism consists of requirements and (typically) an authorization handler. Granting access to a building consists of invoking an API that unlocks the entry door. IoT devices stream biometric information to an Azure IoT Hub, which in turn triggers the verification workflow by posting the site ID, a unique identifier of the site. The Web API POST method simply returns an HTTP code 200 and a JSON message with user name and site ID if the authorization is successful. Otherwise, it throws the expected HTTP 401 Unauthorized Access error code. But let's go in order: I begin with the Startup class of the Web API, specifically the ConfigureServices method, which contains the instructions for configuring the necessary services to run the ASP.NET Core application. The authorization policies are added by calling the AddAuthorization method on the services object. The AddAuthorization method accepts a collection of policies that must be possessed by the API function when invoked for authorizing its execution. I need only one policy in this case, which I call "AuthorizedUser." This policy, however, has several requirements to meet, which reflect the biometric characteristics of a person I want to verify: face, body and voice. The three requirements are each represented by a specific class that implements the IAuthorizationRequirement interface, as shown in **Figure 1**. When listing the requirements for the AuthorizedUser policy, I also specify the confidence level required for meeting the requirement. As I noted earlier, this value, between zero and one, expresses the accuracy of the identification of the respective biometric attribute. I'll get back to this later when discussing biometric recognition with Cognitive Services.

Figure 1 **Configuration of the Authorization Requirements in the Web API**

```
public void ConfigureServices(IServiceCollection services)
{
  var authorizationRequirements = new List<IAuthorizationRequirement>
  {
    new FaceRecognitionRequirement(confidence: 0.9),
    new BodyRecognitionRequirement(confidence: 0.9),
    new VoiceRecognitionRequirement(confidence: 0.9)
  };

  services
    .AddAuthorization(options =>
    {
      options.AddPolicy("AuthorizedUser", policy => policy.Requirements =
        authorizationRequirements);
    })
```

The AuthorizedUser authorization policy contains multiple authorization requirements, and all requirements must pass in order for the policy evaluation to succeed. In other words, multiple authorization requirements added to a single authorization policy are treated on an AND basis.

The three policy requirements that I implemented in the solution are all classes that implement the IAuthorizationRequirement interface. This interface is actually empty; that is, it doesn't dictate the implementation of any method. I've implemented the three requirements consistently by specifying a public ConfidenceScore property for capturing the expected level of confidence that the recognition API should meet for considering this requirement successful. The FaceRecognitionRequirement class looks like this:

```
public class FaceRecognitionRequirement : IAuthorizationRequirement
{
  public double ConfidenceScore { get; }

  public FaceRecognitionRequirement(double confidence) =>
    ConfidenceScore = confidence;
}
```

Similarly, the other requirements for body and voice recognition are implemented, respectively, in the BodyRecognitionRequirement and VoiceRecognitionRequirement classes.

Authorization to execute a Web API action is controlled through the Authorize attribute. At its simplest, applying AuthorizeAttribute to a controller or action limits access to that controller or action to any authenticated user. The Web API that controls access to a site exposes a single access controller, which contains only the Post action. This action is authorized if all requirements in the specified "AuthorizedUser" policy are met:

```
[ApiController]
public class AccessController : ControllerBase
{
  [HttpPost]
  [Authorize(Policy = "AuthorizedUser")]
  public IActionResult Post([FromBody] string siteId)
  {
    var response = new
    {
      User = HttpContext.User.Identity.Name,
      SiteId = siteId
    };
    return new JsonResult(response);
  }
}
```

Figure 2 **The Custom Authorization Handler**

```
public class FaceRequirementHandler :
  AuthorizationHandler<FaceRecognitionRequirement>
{
  protected override Task HandleRequirementAsync(
    AuthorizationHandlerContext context,
      FaceRecognitionRequirement requirement)
  {
    string siteId =
      (context.Resource as HttpContext).Request.Query["siteId"];
    IRecognition recognizer = new FaceRecognition();
    if (recognizer.Recognize(siteId, out string name) >=
      requirement.ConfidenceScore)
    {
      context.User.AddIdentity(new ClaimsIdentity(
        new GenericIdentity(name)));
      context.Succeed(requirement);
    }
    return Task.CompletedTask;
  }
}
```

Each requirement is managed by an authorization handler, like the one in **Figure 2**, which is responsible for the evaluation of a policy requirement. You can choose to have a single handler for all requirements, or separate handlers for each requirement. This latter approach is more flexible as it allows you to configure a gradient of authorization requirements that you can easily configure in the Startup class. The face, body and voice requirement handlers extend the AuthorizationHandler<TRequirement> abstract class, where TRequirement is the requirement to be handled. Because I want to evaluate three requirements, I need to write a custom handler that extends AuthorizationHandler for FaceRecognitionRequirement, BodyRecognitionRequirement and VoiceRecognitionRequirement each. Specifically, the HandleRequirementAsync method, which determines whether an authorization requirement is met. This method, as it's asynchronous, doesn't return a real value, except to indicate that the task has completed. Handling authorization consists of marking a requirement as "successful" by invoking the Succeed method on the authorization handler context. This is actually verified by a "recognizer" object, which uses the Cognitive Services API internally (more in the next section). The recognition action, performed by the Recognize method, obtains the name of the identified person and returns a value (score) that expresses the level of confidence that the identification is more (value closer to one) or less (value closer to zero) accurate. An expected level was specified in the API setup. You can tune this value to whatever threshold is appropriate for your solution.

Besides evaluating the specific requirement, the authorization handler also adds an identity claim to the current user. When an identity is created, it may be assigned one or more claims issued by a trusted party. A claim is a name-value pair that represents what the subject is. In this case, I'm assigning the identity claim to the user in context. This claim is then retrieved in the Post action of the Access controller and returned as part of the API's response.

The last step to perform to enable this custom authorization process is the registration of the handler within the Web API. Handlers are registered in the services collection during configuration:

```
services.AddSingleton<IAuthorizationHandler, FaceRequirementHandler>();
services.AddSingleton<IAuthorizationHandler, BodyRequirementHandler>();
services.AddSingleton<IAuthorizationHandler, VoiceRequirementHandler>();
```

This code registers each requirement handler as a singleton using the built-in dependency injection (DI) framework in ASP.NET Core. An instance of the handler will be created when the application starts, and DI will inject the registered class into the relevant object.

Face Identification

The solution uses the Azure Cognitive Services for Vision API to identify a person's face and body. For more information about Cognitive Services and details on the API, please visit bit.ly/2sxsqry.

The Vision API provides face attribute detection and face verification. Face detection refers to the ability to detect human faces in an image. The API returns the rectangle coordinates of the location of the face within the processed image, and, optionally, can extract a series of face-related attributes such as head pose, gender, age, emotion, facial hair, and glasses. Face verification, in contrast, performs an authentication of a detected face against a person's pre-saved face. Practically, it evaluates whether two faces belong to the same person. This is the specific API I use in this security project. To get started, please add the following NuGet package to your Visual Studio solution: Microsoft.Azure.CognitiveServices.Vision.Face 2.2.0-preview

The .NET managed package is in preview, so make sure that you check the "Include prerelease" option when browsing NuGet, as shown in **Figure 3**.

Using the .NET package, face detection and recognition are straightforward. Broadly speaking, face recognition describes the work of comparing two different faces to determine if they're similar or belong to the same person. The recognition operations mostly use the data structures listed in **Figure 4**.

The verification operation takes a face ID from a list of detected faces in an image (the DetectedFace collection) and determines whether the faces belong to the same person by comparing the ID against a collection of persisted faces (PersistedFace). Persisted face images that have a unique ID and a name identify a Person. A group of persons can, optionally, be gathered in a PersonGroup in order to improve recognition performance. Basically, a person is a basic unit of identity and the person object can have one or more known faces registered. Each person is defined within a particular PersonGroup—a collection of people—and the identification is done against a PersonGroup. The security system would create one or more PersonGroup objects and then associate people

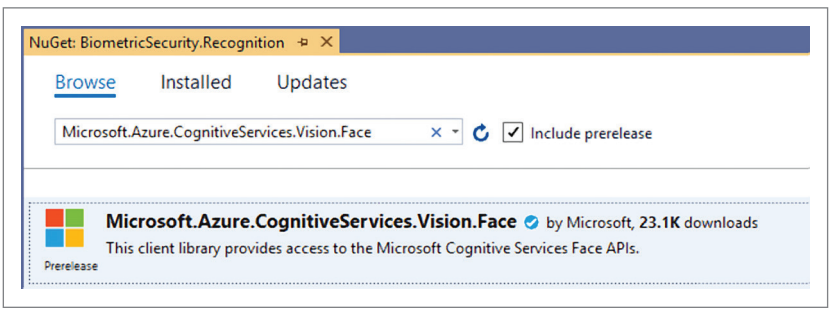


Figure 3 **NuGet Package for the Face API**

with them. Once a group is created, the PersonGroup collection must be trained before an identification can be performed using it. Moreover, it has to be retrained after adding or removing any person, or if any person has their registered face edited. The training is done by the PersonGroup Train API. When using the client library, this is simply a call to the TrainPersonGroupAsync method:

```
await faceServiceClient.TrainPersonGroupAsync(personGroupId);
```

Training is an asynchronous process. It may not be finished even after the TrainPersonGroupAsync method returns. You may need to query the training status with the GetPersonGroupTrainingStatusAsync method until it's ready before progressing with face detection or verification.

When performing face verification, the Face API computes the similarity of a detected face among all the faces within a group, and returns the most comparable person(s) for that test face. This is done through the IdentifyAsync method of the client library. The test face needs to be detected using the aforementioned steps, and the face ID is then passed to the Identify API as a second argument. Multiple face IDs can be identified at once, and the result will contain all the Identify results. By default, Identify returns only one person that matches the test face best. If you prefer, you can specify the optional parameter maxNumOfCandidatesReturned to let Identify return more candidates. The code in **Figure 5** demonstrates the process of identifying and verifying a face:

First, you need to obtain a client object for the Face API by passing your subscription key and the API endpoint. You can obtain both values from your Azure Portal where you provisioned the Face API service. You then detect any face visible in an image, passed as a stream to the DetectWithStreamAsync method of the client's Face object. The Face object implements the detection and verification operations for the Face API. From the detected faces, I ensure that only one is actually detected, and obtain its ID—its unique identifier in the registered face collection of all authorized people to access that site. The IdentifyAsync method then performs the identification of the detected face within a PersonGroup, and returns a list of best matches, or candidates, sorted by confidence level. With the person ID of the first candidate, I retrieve the person name, which is eventually returned to the Access Web API. The face authorization requirement is met.

Figure 4 **Data Structures for the Face API**

Name	Description
DetectedFace	This is a single face representation retrieved by the face detection operation. Its ID expires 24 hours after it's created.
PersistedFace	When **DetectedFace** objects are added to a group (such as FaceList or Person), they become PersistedFace objects, which can be retrieved at any time and do not expire.
FaceList/ LargeFaceList	This is an assorted list of **PersistedFace** objects. A **FaceList** has a unique ID, a name string and, optionally, a user data string.
Person	This is a list of **PersistedFace** objects that belong to the same person. It has a unique ID, a name string and, optionally, a user data string.
PersonGroup/ LargePersonGroup	This is an assorted list of **Person** objects. It has a unique ID, a name string and, optionally, a user data string. A **PersonGroup** must be trained before it can be used in recognition operations.

Voice Recognition

The Azure Cognitive Services Speaker Recognition API provides algorithms for speaker verification and speaker identification. Voices have unique characteristics that can be used to identify a person, just like a fingerprint. The security solution in this article uses voice as a signal for access control, where the subject says a pass phrase into a microphone registered as an IoT device. Just as with face recognition, voice recognition also requires a pre-enrollment of authorized people. The Speaker API calls an enrolled person a "Profile." When enrolling a profile, the speaker's voice is recorded saying a specific phrase, then a number of features are extracted and the chosen phrase is recognized. Together, both extracted features and the chosen phrase form a unique voice signature. During verification, an input voice and phrase are compared against the enrollment's voice signature and phrase, in order to verify whether they're from the same person and the phrase is correct.

Looking at the code implementation, the Speaker API doesn't benefit from a managed package in NuGet like the Face API, so

Figure 5 **The Face Recognition Process**

```
public class FaceRecognition : IRecognition
{
  public double Recognize(string siteId, out string name)
  {
    FaceClient faceClient = new FaceClient(
      new ApiKeyServiceClientCredentials("<Subscription Key>"))
    {
      Endpoint = "<API Endpoint>"
    };

    ReadImageStream(siteId, out Stream imageStream);

    // Detect faces in the image
    IList<DetectedFace> detectedFaces =
      faceClient.Face.DetectWithStreamAsync(imageStream).Result;

    // Too many faces detected
    if (detectedFaces.Count > 1)
    {
      name = string.Empty;
      return 0;
    }

    IList<Guid> faceIds = detectedFaces.Select(f => f.FaceId.Value).ToList();

    // Identify faces
    IList<IdentifyResult> identifiedFaces =
      faceClient.Face.IdentifyAsync(faceIds, "<Person Group ID>").Result;

    // No faces identified
    if (identifiedFaces.Count == 0)
    {
      name = string.Empty;
      return 0;
    }

    // Get the first candidate (candidates are ranked by confidence)
    IdentifyCandidate candidate =
      identifiedFaces.Single().Candidates.FirstOrDefault();

    // Find the person
    Person person =
      faceClient.PersonGroupPerson.GetAsync("", candidate.PersonId).Result;
    name = person.Name;

    return candidate.Confidence;
  }
```

the approach I'll take is to invoke the REST API directly with an HTTP client request and response mechanism. The first step is to instantiate an HttpClient with the necessary parameters for authentication and data type:

```
public VoiceRecognition()
{
  _httpClient = new HttpClient();
  _httpClient.BaseAddress = new Uri("<API Endpoint>");
  _httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key",
    "<Subscription Key>");
  _httpClient.DefaultRequestHeaders.Accept.Add(
     new MediaTypeWithQualityHeaderValue("application/json"));
}
```

The Recognize method in **Figure 6** develops in several steps, as follows: After obtaining the audio stream from the IoT device at the site, it attempts to identify that audio against a collection of enrolled profiles. Identification is coded in the IdentifyAsync method. This asynchronous method prepares a multipart request message that contains the audio stream and the identification profile IDs, and submits a POST request to a specific endpoint. If the response from the API is HTTP Code 202 (Accepted), then the value returned is a URI of the operation that runs in the background. This operation, at the identified URI, is checked by the Recognize method every 100 ms for completion. When it succeeds, you've obtained the profile ID of the identified person. With that ID, you can proceed with the verification of the audio stream, which is the final confirmation that the recorded voice belongs to the identified person. This is implemented in the VerifyAsync method, which works similarly to the IdentifyAsync method except that it returns a VoiceVerificationResponse object, which contains the profile of the person, and thus their name. The verification response includes a confidence level, which is also returned to the Access Web API, as with the Face API.

I want to add a few additional comments about this API, indicating how it differs from the Face API. The voice verification API returns a JSON object that contains the overall result of the verification operation (Accept or Reject), confidence level (Low, Normal or High) and the recognized phrase:

```
{
  "result" : "Accept", // [Accept | Reject]
  "confidence" : "Normal", // [Low | Normal | High]
  "phrase": "recognized phrase"
}
```

This object is mapped to the VoiceVerificationResponse C# class for convenience of operation within the VerifyAsync method, but its level of confidence is expressed as a text:

```
public class VoiceVerificationResponse
{
  [JsonConverter(typeof(StringEnumConverter))]
  public Result Result { get; set; }

  [JsonConverter(typeof(StringEnumConverter))]
  public Confidence Confidence { get; set; }

  public string Phrase { get; set; }
}
```

The Access Web API, instead, expects a decimal value (double data type) between zero and one, so I specified some numeric values for the Confidence enumeration:

```
public enum Confidence
{
  Low = 1,
  Normal = 50,
  High = 99
}
```

I then converted these values to double before returning to the Access Web API:

```
private double ToConfidenceScore(Confidence confidence)
{
  return (double)confidence / 100.0d;
}
```

Figure 6 **Voice Recognition**

```
public double Recognize(string siteId, out string name)
{
  ReadAudioStream(siteId, out Stream audioStream);

  Guid[] enrolledProfileIds = GetEnrolledProfilesAsync();
  string operationUri =
    IdentifyAsync(audioStream, enrolledProfileIds).Result;

  IdentificationOperation status = null;
  do
  {
    status = CheckIdentificationStatusAsync(operationUri).Result;
    Thread.Sleep(100);
  } while (status == null);

  Guid profileId = status.ProcessingResult.IdentifiedProfileId;

  VoiceVerificationResponse verification =
    VerifyAsync(profileId, audioStream).Result;
  if (verification == null)
  {
    name = string.Empty;
    return 0;
  }

  Profile profile = GetProfileAsync(profileId).Result;
  name = profile.Name;

  return ToConfidenceScore(verification.Confidence);
}
```

Wrapping Up

That's it for this first part, in which I discussed the overall site access security flow, and covered the implementation of the authorization mechanism in ASP.NET Core Web API using custom policies and requirements. I then illustrated face and voice recognition, using the relevant Cognitive Services API, as a mechanism to restrict access based on biometric information of pre-authorized, or enrolled, person profiles. In the second part of this article, I'll go through the data streaming from the IoT devices as a triggering point for requesting access, and the final confirmation from the Access API to unlock (or lock!) the access door. I'll also cover a machine learning-based anomaly detection service that will run on any access attempt to identify its risk.

The source code for this initial part of the solution is available on GitHub at bit.ly/2IXPZCo. ■

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 who reviewed this article: *Barry Dorrans*

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

Super-DRY Development for ASP.NET Core

Thomas Hansen

DRY is one of those really important software architecture acronyms. It means "Don't Repeat Yourself" and articulates a critical principle to anyone who's maintained a legacy source code project. That is, if you repeat yourself in code, you'll find that every bug fix and feature update will have you repeating your modifications.

Code repetition reduces your project's maintainability, and makes it more difficult to apply changes. And the more repetition you have, the more spaghetti code you'll end up with. If, on the other hand, you avoid repetition, you can end up with a project that's significantly easier to maintain and bug fix, and you'll be a happier and more productive software developer, to boot. In short, committing to DRY code can help you create great code.

Once you start thinking in a DRY manner, you can bring this important architectural principle to a new level, where it feels as if your project is magically rising up from the ground—literally without having to apply any effort to create functionality. To the uninitiated, it may seem as if code is appearing out of thin air through the mechanisms of "super-DRY." Great code is almost always tiny, but brilliant code is even tinier.

In this article, I'll introduce you to the magic of super-DRY development and some of the tricks I've used over the years that can help you create your ASP.NET Core Web APIs with much less effort. Everything in this article is based on generalized solutions and the concept of DRY code, and uses only best practices from our industry. But first, some background theory.

This article discusses:

- Using dependency injection to improve encapsulation
- Employing ORM to abstract away from a database
- Achieving full modularization of code by avoiding direct references between modules

Technologies discussed:

ASP.NET Web APIs, HTTP REST, Generics, Polymorphism

CRUD, HTTP REST and SQL

Create, Read, Update and Delete (CRUD) is the foundational behavior of most data models. In most cases, your data entity types need these four operations, and in fact both HTTP and SQL are arguably built around them. HTTP POST is for creating items, HTTP GET is for reading items, HTTP PUT is for updating your items, and HTTP DELETE is for deleting items. SQL likewise evolves around CRUD with insert, select, update and delete. Once you give it some thought, it's pretty obvious that it's basically all about CRUD, assuming you don't want to go "all in" and implement a CQRS architecture.

So you have the language mechanisms necessary to talk about HTTP verbs in a manner that they propagate all the way from the client's HTTP layer, through your C# code, and into your relational database. Now, all you need is a generic way to implement these ideas through your layers. And you want to do it without repeating yourself, with a brilliant architectural foundation. So let's get started.

First, download the code at github.com/polterguy/magic/releases. Unzip the file and open magic.sln in Visual Studio. Start your debugger, and notice how you already have five HTTP REST endpoints in Swagger UI. Where did these HTTP endpoints originate? Well, let's look at the code, because the answer to that question may surprise you.

Look Mom, No Code!

The first thing you'll notice as you start browsing the code is that the ASP.NET Core Web project itself is literally empty. This is possible due to an ASP.NET Core feature that allows you to dynamically include controllers. If you want to see the internals behind this, you can check out the Startup.cs file. Basically, it's dynamically adding each controller from all the assemblies in your folder into your AppDomain. This simple idea allows you to reuse your controllers and to think in a modularized way as you compose your solutions. The ability to reuse controllers across multiple projects is step 1 on your way to becoming a super-DRY practitioner.

Open up the web/controller/magic.todo.web.controller project and look at the TodoController.cs file. You'll notice that it's empty. So where did these five HTTP REST endpoints come from? The answer is through the mechanism of object-oriented programming (OOP) and C# generics. The TodoController class inherits from CrudController, passing in its view model and its database model. In addition, it's using dependency injection to create an instance of the ITodoService, which it hands over to the CrudController base class.

Because the ITodoService interface inherits from ICrudService with the correct generic class, the CrudController base class happily accepts your service instance. In addition, at this point it can already use the service polymorphistically, as if it were a simple ICrudService, which of course is a generic interface with parameterized types. This provides access to five generically defined service methods in the CrudController. To understand the implications of this, realize that with the following simple code you've literally created all the CRUD operations you'll ever need, and you've propagated them from the HTTP REST layer, through the service layer, into the domain class hierarchy, ending up in the relational database layer. Here's the entire code for your controller endpoint:

```
[Route("api/todo")]
public class TodoController : CrudController<www.Todo, db.Todo>
{
  public TodoController(ITodoService service)
    : base(service)
  { }
}
```

This code gives you five HTTP REST endpoints, allowing you to create, read, update, delete and count database items, almost magically. And your entire code was "declared" and didn't contain a single line of functionality. Now of course it's true that code doesn't generate itself, and much of the work is done behind the scenes, but the code here has become "Super-DRY." There's a real advantage to working with a higher level of abstraction. A good analogy would be the relationship between a C# if-then statement and the underlying assembly language code. The approach I've outlined is simply a higher level of abstraction than hardcoding your controller code.

In this case, the www.Code type is your view model, the db.Todo type is your database model and ITodoService is your service implementation. By simply hinting to the base class what type you want to persist, you have arguably finished your job. The service layer again is equally empty. Its entire code can be seen here:

```
public class TodoService : CrudService<Todo>, ITodoService
{
  public TodoService([Named("default")] ISession session)
    : base(session, LogManager.GetLogger(typeof(TodoService)))
  { }
}
```

Zero methods, zero properties, zero fields, and yet there's still a complete service layer for your TODO items. In fact, even the service interface is empty. The following shows the entire code for the service interface:

```
public interface ITodoService : ICrudService<Todo>
{ }
```

Again, empty! Yet still, sim salabim, abra kadabra, and you have a complete TODO HTTP REST Web API application. If you open up the database model, you'll see the following:

```
public class Todo : Model
{
  public virtual string Header { get; set; }
  public virtual string Description { get; set; }
  public virtual bool Done { get; set; }
}
```

Once again, there's nothing here—just a couple of virtual properties and a base class. And yet, you're able to persist the type into your database. The actual mapping between your database and your domain type occurs in the TodoMap.cs class inside the magic.todo.model project. Here, you can see the entire class:

```
public class TodoMap : ClassMap<Todo>
{
  public TodoMap()
  {
    Table("todos");
    Id(x => x.Id);
    Map(x => x.Header).Not.Nullable().Length(256);
    Map(x => x.Description).Not.Nullable().Length(4096);
    Map(x => x.Done).Not.Nullable();
  }
}
```

This code instructs the ORM library to use the todos table, with the Id property as the primary key, and sets a couple of additional properties for the rest of the columns/properties. Notice that when you started this project, you didn't even have a database. This is because NHibernate automatically creates your database tables if they don't already exist. And because Magic by default is using SQLite, it doesn't even need a connection string. It'll automatically create a file-based SQLite database at a relative file path, unless you override its connection settings in appsettings.config to use MySQL or MSSQL.

Believe it or not, your solution already transparently supports almost any relational databases you can imagine. In fact, the only line of actual code you would need to add to make this thing function can be found in the magic.todo.services project, inside the ConfigureNinject class, which simply binds between the service interface and the service implementation. So, arguably, you added one line of code, and got an entire application as the result. Following is the only line of actual "code" used to create the TODO application:

```
public class ConfigureNinject : IConfigureNinject
{
  public void Configure(IKernel kernel, Configuration configuration)
  {
    // Warning, this is a line of C# code!
    kernel.Bind<ITodoService>().To<TodoService>();
  }
}
```

We've become super-DRY magicians, through intelligent use of OOP, generics and the principles of DRY. So the question begs: How can you use this approach to make your code better?

The answer: Start out with your database model and create your own model class, which can be done either by adding a new project inside your model folder, or by adding a new class to the existing magic.todo.model project. Then create your service interface in the contracts folder. Now implement your service in your services folder, and create your view model and controller. Make sure you bind between your service interface and your service implementation. Then if you choose to create new projects, you'll have to make sure ASP.NET Core loads your assembly by adding a reference to it in your magic.backend project. Notice that only the service project and the controller need to be referenced by your back end.

If you chose to use the existing projects, the last part isn't even necessary. One simple line of actual code to bind between your service implementation and your service interface, and you've created an entire ASP.NET Core Web API solution. That's one mighty line of code if you ask me. You can see me go through the entire process for an earlier version of the code in my "Super DRY Magic for ASP.NET Core" video at youtu.be/M3uKdPAvS1I.

Then imagine what occurs when you realize that you can scaffold this code, and automatically generate it according to your database schema. At this point, your computer scaffolding software system is arguably doing your coding, producing a perfectly valid Domain-Driven Design (DDD) architecture in the process.

No Code, No Bugs, No Problem

Most scaffolding frameworks apply shortcuts, or prevent you from extending and modifying their resulting code, such that using them for real-world applications becomes impossible. With Magic, this shortcoming is simply not true. It creates a service layer for you, and it uses dependency injection to inject a service interface to your controller. It also produces perfectly valid DDD patterns for you. And as you've created your initial code, every single part of your solution can be extended and modified as you see fit. Your project perfectly validates to every single letter in SOLID.

For instance, in one of my own solutions, I have a POP3 server fetcher thread in my service, declared for an EmailAccount domain model type. This POP3 service stores emails into my database from my POP3 server, running on a background thread. When an email is deleted, I want to make sure I also physically delete its attachments in storage, and if the user deletes an EmailAccount, I obviously want to delete its associated emails.

> We've become super-DRY Magicians, through intelligent use of OOP, generics and the principles of DRY. So the question begs: How can you use this approach to make your code better?

The code in **Figure 1** shows how I've overridden the deletion of an EmailAccount, which also should delete all emails and attachments. For the record, it uses Hibernate Query Language (HQL) to communicate with the database. This ensures that NHibernate will automatically create the correct SQL syntax, depending on to which database it's physically connected.

Doing the Math

Once you start philosophizing around these ideas, inspiration strikes. For instance, imagine a scaffolding framework built around Magic. From a mathematical point of view, if you have a database with 100 tables, each with an average of 10 columns, you'll find that the cost in terms of total lines of code can add up fast. For instance, to wrap all of these tables into an HTTP REST API requires seven lines of code per service interface, while 14 lines of code are needed per table for each service and 19 lines are needed per table for each controller. **Figure 2** runs down the elements involved and the lines of code required.

Figure 1 **Overriding the Deletion of an EmailAccount**

```
public sealed class EmailAccountService : CrudService<EmailAccount>,
  IEmailAccountService
{
  public EmailAccountService(ISession session)
    : base(session, LogManager.GetLogger(typeof(EmailAccountService)))
  { }

  public override void Delete(Guid id)
  {
    var attachments = Session.CreateQuery(
      "select Path from EmailAttachment where Email.EmailAccount.Id = :id");
    attachments.SetParameter("id", id);
    foreach (var idx in attachments.Enumerable<string>())
    {
      if (File.Exists(idx))
        File.Delete(idx);
    }

    var deleteEmails = Session.CreateQuery(
      "delete from Email where EmailAccount.Id = :id");
    deleteEmails.SetParameter("id", id);
    deleteEmails.ExecuteUpdate();

    base.Delete(id);
  }
}
```

Figure 2 **Adding Up the Cost in Code**

Component	Contracts	Average Lines of Code	Total Lines of Code
Service interfaces	100	7	700
Services	100	14	1,400
Controllers	100	19	1,900
Service interface and implementation	100	1	100
View models	100	17	1,700
Database models	100	17	1,700
Database mappings	100	20	2,000
		Total Lines of Code:	**9,500**

After all is said and done, you're looking at 9,500 lines of code. If you build a meta service capable of extracting an existing database schema, it becomes pretty obvious that you can generate this code using scaffolding—arguably avoiding any coding at all, yet still producing 9,500 lines of perfectly architected code, easily extended, using all the relevant design patterns and best practices. With only two seconds of scaffolding, your computer has done 80 percent of your work.

All you have to do now is go through the results of the scaffolding process, and override methods for your services and controllers for the domain types that require special attention for whatever reason. You're done with your Web API. Because the controller endpoints all have the exact same structure, duplicating this scaffolding process in the client layer is as easy as reading the API JSON declaration files generated by Swagger. This enables you to create your service layer for something such as Angular or React. And all this because your code and your Web API have predictable structures, based on generalization principles and the avoidance of repetition.

To put this in perspective, you've managed to create an HTTP REST Web API project that's probably twice as large as the open source Sugar CRM project in complexity, and you did 80 percent of the work in seconds. You've facilitated a software factory assembly line that's based on standardization of components and reuse of structure, while making the code for all your projects much easier to read and maintain. Even the parts that require modification and special behavior can be reused in your next project, thanks to the way controller endpoints and services are dynamically loaded into your Web API, without any dependencies.

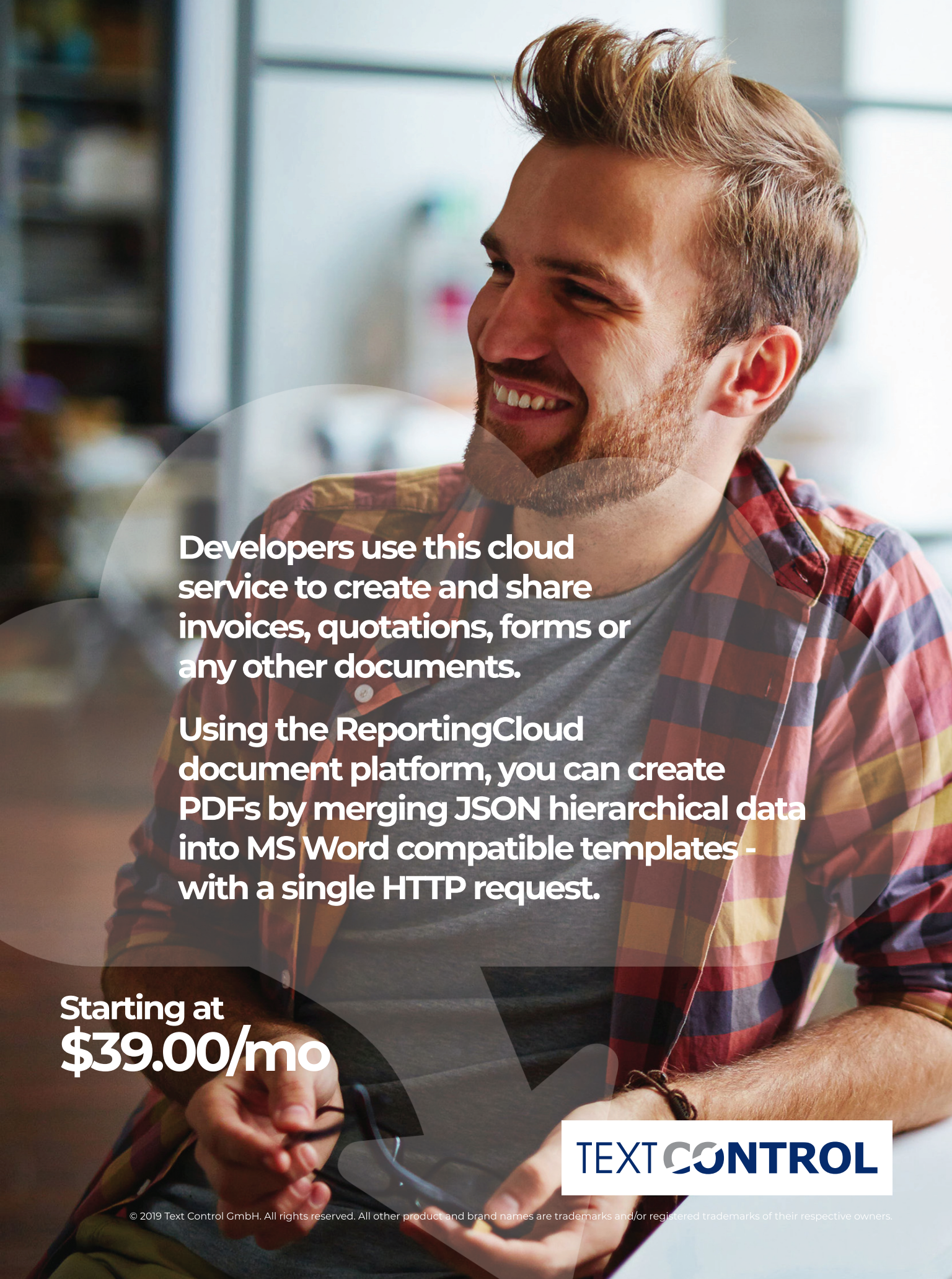
If you work for a consulting company, you probably start several new projects each year with similar types of requirements, where you need to solve the commonalities for each new project. With an understanding of a client's requirements and some initial implementation, a super-DRY approach enables you to literally finish an entire project in seconds. And of course, the composition of elements in your projects can be further reused, by identifying common modules, such as Authentication and Authorization. By implementing these modules in a common Web API project, you can apply them to any new project that poses similar problems to those you've seen before.

For the record, I'm making this sound easy, but the fact is that avoiding repetition is hard. It requires a willingness to refactor, refactor, refactor. And when you're done refactoring, you need to refactor a bit more. But the upside is too good to ignore. DRY principles can let you almost magically create code, by simply waving your scaffolding wand and composing modules out of pre-existing parts.

At the end of the day, the principles articulated here can help you leverage existing best practices to create your own Web APIs while avoiding repetition. There's a lot of good that can come from this approach, and hopefully it helps you appreciate the awesomeness of DRY. ■

Thomas Hansen *is a Zen software wizard currently living in Cyprus, where he juggles software code working its way through FinTech and trading systems.*

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



Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

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

INTENSE DEVELOPER TRAINING CONFERENCE

In-depth Technical Content On:

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

Save $400
When You Register By June 14!
Use Promo Code MSDN

vslive.com/microsofthq

GOLD SPONSOR
Progress

SUPPORTED BY
Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY
CONVERGE 360

AGENDA AT-A-GLANCE

#VSLIVE

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

START TIME	END TIME				
		Pre-Conference Full Day Hands-On Labs: Monday, August 12, 2019 *(Separate entry fee required)*			
7:00 AM	8:30 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries			
8:30 AM	12:30 PM	**HOL01** Hands-On Lab: Hands-On With Cloud-Native .NET Development - *Rockford Lhotka and Richard Seroter*	**HOL02** Hands-On Lab: Leveling Up—Dependency Injection in .NET - *Jeremy Clark*	**HOL03** Full Day Hands-On Lab: Xamarin and Azure: Build the Mobile Apps of Tomorrow - *Laurent Bugnion*	
12:30 PM	2:00 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
2:00 PM	5:30 PM	**HOL01** Hands-On Lab Continued - *Rockford Lhotka and Richard Seroter*	**HOL02** Hands-On Lab Continued - *Jeremy Clark*	**HOL03** Hands-On Lab Continued - *Laurent Bugnion*	
7:00 PM	9:00 PM	Dine-A-Round Dinner			
START TIME	**END TIME**	**Visual Studio Live! Day 1: Tuesday, August 13, 2019**			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**T01** Angular 101 - *Deborah Kurata*	**T02** Stunning Mobile Apps with the Xamarin Visual Studio System - *James Montemagno*	**T03** SQL Server 2019 Deep Dive - *Scott Klein*	**T04** C# and/or .NET Core 3.0 - *Dustin Campbell*
9:30 AM	10:45 AM	**T05** Moving to ASP.NET Core 2.X - *Philip Japikse*	**T06** Sharing C# Code Across Platforms - *Rockford Lhotka*	**T07** Application Modernization with Microsoft Azure - *Michael Crump*	**T08** DevOps Practices Can Make You A Better Developer - *Robert Green*
10:45 AM	11:15 AM	Sponsored Break - Visit Exhibitors - *Foyer*			
11:15 AM	12:15 PM	**KEYNOTE: Moving .NET Beyond Windows** - *James Montemagno, Principal Program Manager - Mobile Developer Tools, Microsoft*			
12:15 PM	1:30 PM	Lunch - McKinley / Visit Exhibitors - *Foyer*			
1:30 PM	2:45 PM	**T09** Angular Best Practices - *Deborah Kurata*	**T10** To Be Announced	**T11** Data Pipelines with End-2-End Azure Analytics - *Scott Klein*	**T12** Get Func-y: Understanding Delegates in .NET - *Jeremy Clark*
3:00 PM	4:15 PM	**T13** Versioning ASP.NET Core APIs -*Philip Japikse*	**T14** Visual Studio for Mac: From Check-out to Publish - *Dominic Nahous and Cody Beyer*	**T15** Why, When, and How to Enhance Your App with Azure Functions - *Daria Grigoriu & Eamon O'Reilly*	**T16** CI / CD with Azure DevOps - *Tiago Pascoal*
4:15 PM	5:45 PM	Microsoft Ask the Experts & Welcome Reception			
START TIME	**END TIME**	**Visual Studio Live! Day 2: Wednesday, August 14, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**W01** Getting Pushy with SignalR and Reactive Extensions - *Jim Wooley*	**W02** AI and Analytics with Apache Spark on Azure Databricks - *Andrew Brust*	**W03** Building Business Application Bots - *Michael Washington*	**W04** DevOps with ASP.NET Core, EF Core, & Azure DevOps - *Benjamin Day*
9:30 AM	10:45 AM	**W05** Building Reactive Client Experiences with RxJs and Angular Forms *Jim Wooley*	**W06** Power BI: What Have You Done for Me Lately - *Andrew Brust*	**W07** Windows Subsystem for Linux - *Rich Turner and Craig Loewen*	**W08** Real World Scrum with Azure DevOps - *Benjamin Day*
11:00 AM	12:00 PM	**GENERAL SESSION:** .NET Core 3 - *Scott Hunter, Director of Program Management, Microsoft*			
12:00 PM	1:30 PM	Birds-of-a-Feather Lunch - McKinley / Visit Exhibitors - Foyer			
1:30 PM	2:45 PM	**W09** C# in the Browser with Client-side Blazor and WebAssembly - *Rockford Lhotka*	**W10** UX Design Fundamentals: What do Your Users Really See? - *Billy Hollis*	**W11** Works On My Machine... Docker for Developers - *Chris Klug*	**W12** Bolting Security Into Your Development Process - *Tiago Pascoal*
2:45 PM	3:15 PM	Sponsored Break - Exhibitor Raffle @ 2:55 pm (Must be present to win) - Foyer			
3:15 PM	4:30 PM	**W13** Creating Business Applications Using Blazor (Razor Components) - *Michael Washington*	**W14** What's New with Azure App Service? - *Christina Compy*	**W15** Microservices with Azure Kubernetes Service (AKS) - *Vishwas Lele*	**W16** Database DevOps with SQL Server - *Brian Randell*
6:15 PM	9:00 PM	VSLive!'s Downtown Seattle Adventure			
START TIME	**END TIME**	**Visual Studio Live! Day 3: Thursday, August 15, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**TH01** Angular Application Testing Outside the Church of TDD - *Chris Klug*	**TH02** Add Native to Cross Platform with Xamarin.Essentials - *Veronika Kolesnikova*	**TH03** Best Practices for Serverless DevOps: Inner Loop, Outer Loop, Observability - *Colby Tresness*	**TH04** Visual Studio Productivity Tips and Tricks - *Allison Buchholtz-Au*
9:30 AM	10:45 AM	**TH05** Any App, Any Language with Visual Studio Code and Azure DevOps - *Brian Randell*	**TH06** MVVM Made Easy for WPF Applications - *Paul Sheriff*	**TH07** Cross Platform Automation with PowerShell Core 6.0	**TH08** Exceptional Development: Dealing With Exceptions in .NET - *Jason Bock*
11:00 AM	12:15 PM	**TH09** To Be Announced	**TH10** Now, The Two Worlds Collided - *Veronika Kolesnikov*	**TH11** What Every Developer Needs to Know About Deep Learning - *Vishwas Lele*	**TH12** Use Async Internals in .NET - *Adam Furmanek*
12:15 PM	2:00 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center			
2:00 PM	3:15 PM	**TH13** What's New in Bootstrap 4 - *Paul Sheriff*	**TH14** How to Interview a Developer - *Billy Hollis*	**TH15** Building Azure Cosmos DB Applications – Part I - *Leonard Lobel*	**TH16** Testability in .NET - *Jason Bock*
3:30 PM	4:45 PM	**TH17** To Be Announced	**TH18** To Be Announced	**TH19** Building Azure Cosmos DB Applications – Part II - *Leonard Lobel*	**TH20** Manual Memory Management in .NET Framework - *Adam Furmanek*
START TIME	**END TIME**	**Post-Conference Workshops: Friday, August 16, 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: ASP.NET Core with Azure DevOps - *Brian Randell*	**F02** Workshop: Web Development in 2019 - *Chris Klug*	**F03** Workshop: SQL Server for Developers: The Grand Expedition - *Andrew Brust and Leonard Lobel*	

Speakers and sessions subject to change

Denotes Microsoft Speaker/Session

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

MSIX: The Modern Way to Deploy Desktop Apps on Windows

Magnus Montin

MSIX is the new packaging format that was introduced with the October 2018 update of Windows 10. It aims to bring together the best of all previous installation technologies, such as MSI and ClickOnce, and will be the recommended way of installing applications on Windows going forward. This article shows you how to package a .NET desktop application and how to set up continuous integration (CI), continuous deployment (CD) and automatic updates of sideloaded MSIX packages using Azure Pipelines.

First, a bit of background. In Windows 8, Microsoft introduced an API and runtime called the Windows Runtime that mainly sought to provide a set of platform services to a new kind of application, which was originally referred to as "modern," "Metro," "immersive," or just a "Windows Store" app. This kind of app was born of the mobile device revolution and typically targeted multiple device form factors, such as phones, tablets, and laptops, and was usually installed and updated from the central Microsoft Store.

This article discusses:

- MSIX packaging of "classic" desktop applications
- DevOps
- Continuous Integration
- Continuous Deployment

Technologies discussed:

MSIX, Universal Windows Platform, Desktop Bridge, .NET, Continuous Integration, Continuous Deployment, DevOps, App Installer, Azure Pipelines, YAML

Code download available at:

msdn.com/magazine/0619magcode

This class of app has evolved quite a bit since then and is now known as a Universal Windows Platform (UWP) app. UWP apps run in a sandbox called an AppContainer that's isolated from other processes. They explicitly declare capabilities to require the permissions needed to function properly, and it's up to the user to decide whether these capabilities should be accepted. This is in contrast to a traditional desktop application that typically runs as a full-trust process with the current user's full read and write permissions.

In the Anniversary Update of Windows 10, Microsoft introduced the Desktop Bridge (also known as the Centennial project). It let you package your traditional desktop application as a UWP app, but still run it as a full-trust process. A packaged application can be uploaded to the Microsoft Store or the Store for Business and benefit from the streamlined deployment and built-in licensing and automatic update facilities the store provides. Once you've packaged your application, you can also start using the new Windows 10 APIs and migrate your code to the UWP in order to reach customers across all devices.

Even if you're not interested in the Store or the UWP, you may still want to package your line-of-business desktop applications to take advantage of the new app model that Windows 10 brings. It provides clean installs and uninstalls of apps by automatically redirecting all operations against the registry and some well-known system folders to a local folder of the installed application, where a virtual file system and registry are set up. You don't have to do anything in your source code for this to happen—it's taken care of for you automatically by Windows. The idea is that when a package is uninstalled, the entire local folder is removed, leaving no traces of the app left on the system.

MSIX is basically a successor to the Desktop Bridge, and the contents of an MSIX package—and the limitations that apply to packaged apps—are roughly the same as with the APPX format that the Desktop Bridge uses. The requirements are listed in the

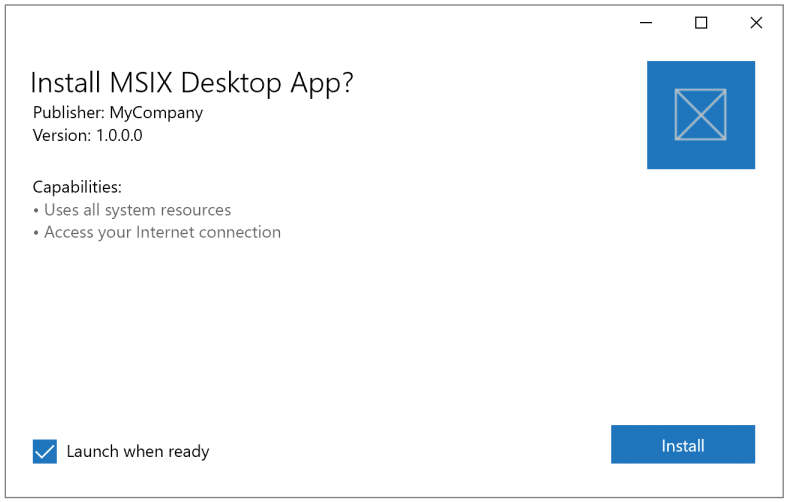


Figure 1 **The App Installer and MSIX Installation Experience in Windows 10**

official docs at bit.ly/2OvCcVW and should be addressed before you decide whether to package your applications. Some of them apply only to apps that are being published to the Store.

Figure 2 **The YAML File That Defines the MSIX Build Pipeline**

```
pool:
  vmImage: vs2017-win2016
variables:
  buildPlatform: 'x86'
  buildConfiguration: 'release'
  major: 1
  minor: 0
  build: 0
  revision: $[counter('rev', 0)]

steps:
- powershell: |
   [Reflection.Assembly]::LoadWithPartialName("System.Xml.Linq")
   $path = "Msix/Package.appxmanifest"
   $doc = [System.Xml.Linq.XDocument]::Load($path)
   $xName =
     [System.Xml.Linq.XName]
       "{http://schemas.microsoft.com/appx/manifest/foundation/windows10}Identity"
   $doc.Root.Element($xName).Attribute("Version").Value =
     "$(major).$(minor).$(build).$(revision)";
   $doc.Save($path)
  displayName: 'Version Package Manifest'

- task: MSBuild@1
  inputs:
    solution: Msix/Msix.wapproj
    platform: $(buildPlatform)
    configuration: $(buildConfiguration)
    msbuildArguments: '/p:OutputPath=NonPackagedApp
     /p:UapAppxPackageBuildMode=SideLoadOnly  /p:AppxBundle=Never
/p:AppxPackageOutput=$(Build.ArtifactStagingDirectory)\MsixDesktopApp.msix
/p:AppxPackageSigningEnabled=false'
  displayName: 'Package the App'

- task: DownloadSecureFile@1
  inputs:
    secureFile: 'certificate.pfx'
  displayName: 'Download Secure PFX File'

- script: '"C:\Program Files (x86)\Windows Kits\10\bin\10.0.17763.0\x86\signtool"
    sign /fd SHA256 /f $(Agent.TempDirectory)/certificate.pfx /p secret $(
    Build.ArtifactStagingDirectory)/MsixDesktopApp.msix'
  displayName: 'Sign MSIX Package'

- task: PublishBuildArtifacts@1
  displayName: 'Publish Artifact: drop'
```

MSIX adds a new feature called modification packages. It's a concept similar to MST transformations that enables IT administrators to customize an app, typically from a third-party vendor, without having to repackage it from scratch each time a new feature or bug fix is released. The modification package is merged with the main application at run time and may disable some features of the app, by changing some registry settings, for example. From a developer's point of view, this might not bring that much to the table, assuming you own both the source code and the build and release pipelines for your apps, but for large enterprises it may cut costs and prevent what's known as package paralysis.

The definition of the MSIX format is open sourced on GitHub and Microsoft plans to provide an SDK that can be used to pack and unpack MSIX packages on all major OSes, including both Linux and macOS. MSIX was officially introduced with version 1809 of Windows 10 in October 2018, and Microsoft then added support for it to earlier versions—the April 2018 Update (version 1803) and the October 2017 Fall Creators Update (version 1709).

Packaging

If you have an existing installer, there's an MSIX Packaging Tool available in the Store that lets you convert it to an MSIX. This enables administrators to package existing applications without even having access to the original source code. For developers, Visual Studio 2017 version 15.5 and higher provides a Windows Application Packaging Project that makes the process of packaging an existing application straightforward. You'll find it under File | Add | New Project | Installed | Visual C# | Windows Universal. It includes an Application folder that you can right-click in the Solution Explorer and choose to add a reference to your Windows Presentation Foundation (WPF), Windows Forms (WinForms) or whatever desktop project you want to package. If you then right-click on the referenced application and choose Set as Entry Point, you'll be able to build, run and debug your application just as you're used to.

The difference between starting the original desktop process versus the packaging project is that the latter will run your application inside a modern app container. Behind the scenes, Visual Studio uses the MakeAppx and SignTool command-line tools from the Windows SDK to first create an .msix file, and then sign it with a certificate. This step isn't optional. All MSIX packages must be signed with a certificate that chains to a trusted root authority on the machine where you intend to install and run the packaged app.

Digital Signing The packaging project includes a default password-protected personal information exchange (PFX) format file that you probably want to replace with your own. If your enterprise doesn't provide you with a code-signing certificate, you can either buy one from a trusted authority or create a self-signed certificate. There's a "Create test certificate" option and an import wizard in Visual Studio, which you'll find if you open the Package.appxmanifest file in the default app manifest designer and look under the Packaging tab. If you're not that into wizards and dialogs, you can use the New-SelfSignedCertificate PowerShell cmdlet to create a certificate:

```
> New-SelfSignedCertificate -Type CodeSigningCert -Subject "CN=MyCompany,
  O=MyCompany, L=Stockholm, S=N/A, C=Sweden" -KeyUsage DigitalSignature
    -FriendlyName MyCertificate -CertStoreLocation "Cert:\LocalMachine\My"
      -TextExtension @('2.5.29.37={text}1.3.6.1.5.5.7.3.3',
        '2.5.29.19={text}Subject Type:End Entity')
```

The cmdlet outputs a thumbprint (like the *A27…D9F* here) that you can pass to another cmdlet, Move-Item, to move the certificate into the trusted root certification store:

```
>Move-Item Cert:\LocalMachine\My\A27A5DBF5C874016E1A0DEBF38A97061F6625D9F
  -Destination Cert:\LocalMachine\Root
```

Again, you need to install the certificate into this store on all computers where you intend to install and run the packaged app. You also need to enable sideloading of apps on these devices. On an unmanaged computer, this can be done under Update & Security | For Developers in the Settings app. On a device that's managed by an organization, you can turn on sideloading by pushing a policy with a mobile device management (MDM) provider.

The thumbprint can also be used to export the certificate to a new PFX file using the Export-PfxCertificate cmdlet:

```
>$pwd = ConvertTo-SecureString -String secret -Force -AsPlainText
>Export-PfxCertificate -cert
  "Cert:\LocalMachine\Root\A27A5DBF5C874016E1A0DEBF38A97061F6625D9F"
    -FilePath "c:/<SolutionFolder>/Msix/certificate.pfx" -Password $pwd
```

Remember to tell Visual Studio to use the generated PFX file to sign the MSIX package by selecting it under the Packaging tab in the designer, or by manually editing the .wapproj project file and replacing the values of the <PackageCertificateKeyFile> and <PackageCertificateThumbprint> elements.

Package Manifest The Package.appxmanifest file is an XML-based template that the build process uses to generate a digitally signed AppxManifest.xml file that includes all information the OS needs to deploy, display and update the packaged app. This is where you specify the display name and logo of your app, as it will appear in the Windows shell after the app has been installed.

Make sure that the Subject property of the certificate you use to sign the MSIX package with exactly matches the value of the Publisher attribute of the Identity element. Because a packaged desktop application can run only on desktop devices, you should also remove the TargetDeviceFamily element with the name of Windows.Universal from the Dependencies element in the default template that Visual Studio generates.

The MinVersion and MaxVersionTested attributes, or Minimum Version and Target Version as they're called in the dialog that shows up when you create a packaging project, is a UWP concept in which the former specifies the oldest version of the OS that your app is compatible with, and the latter is used to identify the set of APIs that are available when you compile the app. When packaging desktop applications that don't call into any Windows 10 APIs, you should select the same version. Whenever you don't, your code should include runtime API checks to avoid getting exceptions when running your app on devices that target the minimum version.

To generate the actual MSIX package, there's a wizard available under Project | Store | Create App Packages in Visual Studio. An end user installs an MSIX package by simply double-clicking on the generated .msix file. This brings up a built-in, non-customizable dialog, shown in **Figure 1**, that guides you through the process of installing the app.

Continuous Integration

If you want to set up CI for your MSIX packages, Azure Pipelines has great support. It supports Configuration as Code (CAC) through the use of YAML files and provides a cloud-hosted build agent that comes with all the software required to create MSIX packages pre-installed.

Before building the packaging project the same way the wizard in Visual Studio does using the MSBuild command line, the build process can version the MSIX package that's being produced by editing the Version attribute of the Package element in the Package.appxmanifest file. In Azure Pipelines, this can be achieved by using an expression for setting a counter variable that gets incremented for every build, and a PowerShell script that uses the System.Xml.Linq.XDocument class in .NET to change the value of the attribute. **Figure 2** shows an example YAML file that versions and creates an MSIX package based on a packaging project before it copies it to a staging directory on the build agent.

The name of the hosted virtual machine that runs Visual Studio 2017 on Windows Server 2016 is vs2017-win2016. It has the required UWP and .NET development workloads installed, including SignTool, which is used to sign the MSIX package after it has been created by MSBuild. Note that the PFX file shouldn't be added to the source control. It's also ignored by Git by default. Instead, it should be uploaded to Azure Pipelines as a secret file

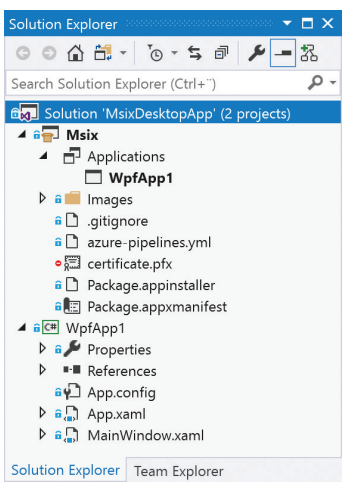


Figure 3 **A Packaged WPF Application Ready to Be Pushed to the Source Control**

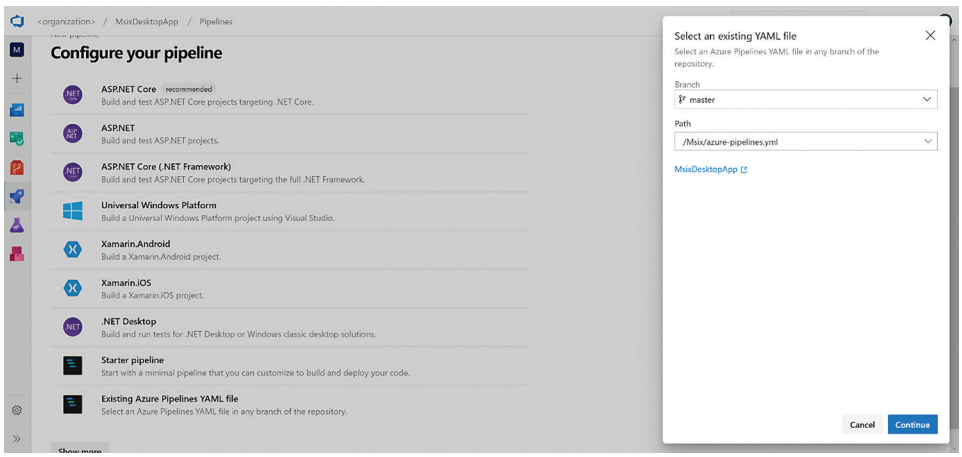


Figure 4 **The Pipeline Configuration Web Interface**

Figure 5 **An .appinstaller File That Will Look for Updated Files on \\server\foo**

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller xmlns="http://schemas.microsoft.com/appx/appinstaller/2018"
              Version="1.0.0.0"
              Uri="\\server\foo\MsixDesktopApp.appinstaller">
  <MainPackage Name="MyCompany.MySampleApp"
               Publisher="CN=MyCompany, O=MyCompany, L=Stockholm, S=N/A, C=Sweden"
               Version="1.0.0.0"
               Uri="\\server\foo\MsixDesktopApp.msix"
               ProcessorArchitecture="x86"/>
  <UpdateSettings>
    <OnLaunch HoursBetweenUpdateChecks="0" />
  </UpdateSettings>
</AppInstaller>
```

under the Library tab in the Web portal. Because it includes the private key of the certificate that represents the digital signature and identity of your company, you don't want to distribute it to more people than necessary.

In large enterprises where you release your software to multiple environments in different stages, it's considered a best practice to sign the packages as part of the release process and let the build pipeline produce unsigned packages. This not only lets you sign with different certificates for different environments, but it also gives you the ability to upload your packages to the Store where they'll be signed by a Microsoft certificate.

Also note that secrets, such as the password for the PFX file, shouldn't be included in the YAML file. Unlike variables that specify the targeted processor architecture and the package version, they're defined and set in the Web interface.

Figure 3 shows the solution explorer for a WPF application that was created using the default project template in Visual Studio and packaged using a Windows Application Packaging project. The YAML file has been added to the packaging project and is checked in to a source code repository together with the rest of the code.

To set up the actual build pipeline, you browse to the Azure DevOps portal at dev.azure.com/<*organization*> and create a new project. If you don't have an account, you can create one for free. Once you've signed in and created a project, you can either push the source code to the Git repository that's set up for you at https://<*organization*>@dev.azure.com/<*organization*>/<*project*>/_git/<*project*>, or use any other provider, such as GitHub. You'll get to choose the location of your repository when you create a new pipeline in the portal by clicking first on the "Pipelines" button and then on "New Pipeline."

On the Configure screen that comes next, you should select the "Existing Azure Pipelines YAML file" option and select the path to the checked-in YAML file in your repository, as **Figure 4** shows.

The MSIX package that's produced by the build can be downloaded and double-click installed on any Windows 10-compatible computer with the required certificate installed, or you could set up a CD pipeline that copies the package to a Web site or a file share where your end users can download it. I'll come back to this in just a bit.

Auto-updates While an MSIX package is able to extract itself and automatically replace any older version of the packaged app that may be present on the machine when you install it, the MSIX format doesn't provide any built-in support for automatically updating an app that has already been installed from an .msix file when you open it.

However, starting with the April 2018 Update of Windows 10, there's support for an app installer file that you can deploy along with your package to enable automatic updates. It contains a MainPackage element whose Uri attribute refers to the original or an updated MSIX package. **Figure 5** shows an example of a minimal .appinstaller file. Note that the Uri attribute of the root element specifies a URL or a UNC path to a file share where the OS will look for the updated files. When the URI differs between a currently installed version and a new app installer file, the deployment operation will redirect to the "old" URI.

The UpdateSettings element is used to tell the system when to check for updates and whether to force the user to update. The full schema reference, including the supported namespaces for each version of Windows 10, can be found in the docs at bit.ly/2TGWnCR.

If you add the .appinstaller file in **Figure 5** to the packaging project and set its Package Action property to Content and the Copy to Output Directory property to Copy if newer, you can then add another PowerShell task to the YAML file that updates the Version attributes of the root and MainPackage elements and saves the updated file to the staging directory:

```
- powershell: |
  [Reflection.Assembly]::LoadWithPartialName("System.Xml.Linq")
  $doc = [System.Xml.Linq.XDocument]::Load(
    "$(Build.SourcesDirectory)/Msix/Package.appinstaller")
  $version = "$(major).$(minor).$(build).$(revision)"
  $doc.Root.Attribute("Version").Value = $version;
  $xName =
    [System.Xml.Linq.XName]
      "{http://schemas.microsoft.com/appx/appinstaller/2018}MainPackage"
  $doc.Root.Element($xName).Attribute("Version").Value = $version;
  $doc.Save("$(Build.ArtifactStagingDirectory)/MsixDesktopApp.appinstaller")
displayName: 'Version App Installer File'
```

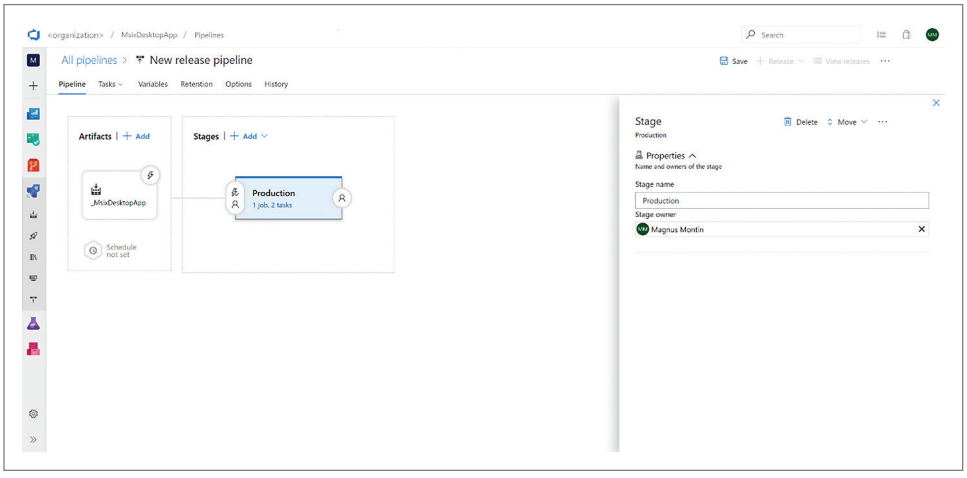


Figure 6 **The Pipeline Tab in the Azure DevOps Portal**

Figure 7 **A Release Pipeline Task That Modifies the Uris in the .appinstaller File**

```
- powershell: |
  [Reflection.Assembly]::LoadWithPartialName("System.Xml.Linq")
  $fileShare = "\\filesharestorageccount.file.core.windows.net\myfileshare\"
  $localFilePath =
    "$(System.DefaultWorkingDirectory)\_MsixDesktopApp\drop\MsixDesktopApp.
appinstaller"
  $doc = [System.Xml.Linq.XDocument]::Load("$localFilePath")
  $doc.Root.Attribute("Uri").Value = [string]::Format('{0}{1}', $fileShare,
    'MsixDesktopApp.appinstaller')
  $xName =
    [System.Xml.Linq.XName]"{http://schemas.microsoft.com/appx/
appinstaller/2018}MainPackage"
  $doc.Root.Element($xName).Attribute("Uri").Value = [string]::Format('{0}{1}',
    $fileShare, 'MsixDesktopApp.appx')
  $doc.Save("$localFilePath")
displayName: 'Modify URIs in App Installer File'
```

You'd then distribute the .appinstaller file to your end users and let them double-click on this one instead of the .msix file to install the packaged app.

Continuous Deployment

The app installer file itself is an uncompiled XML file that can be edited after the build, if required. This makes it easy to use when you deploy your software to multiple environments and when you want to separate the build pipeline from the release process.

If you create a release pipeline in the Azure Portal using the "Empty job" template and use the recently set up build pipeline as the source of the artifact to be deployed, as shown in **Figure 6**, you can then add the PowerShell task in **Figure 7** to the release stage in order to dynamically change the values of the two Uri attributes in the .appinstaller file to reflect the location to which the app is published.

In the task in **Figure 7**, the URI is set to the UNC path of an Azure file share. Because this is where the OS will look for the MSIX package when you install and update the app, I've also added another command-line script to the release pipeline that first maps the file share in the cloud to the local Z:\ drive on the build agent before it uses the xcopy command to copy the .appinstaller and .msix files there:

```
- script: |
  net use Z: \\filesharestorageccount.file.core.windows.net\myfileshare
    /u:AZURE\filesharestorageccount
    3PTYC+ociHIwNgCnyg7zsWoKBxRmkEc4Aew4FMzbpUl/
    dydo/3HVnl71XPeOuWxQcLddEUuqOfN8LtcpcOLYeg==
  xcopy $(System.DefaultWorkingDirectory)\_MsixDesktopApp\drop Z:\ /Y
  displayName: 'Publish App Installer File and MSIX package'
```

If you host your own on-premises Azure DevOps Server, you may of course publish the files to your own internal network share.

Web Installs If you choose to publish to a Web server, you can tell MSBuild to generate a versioned .appinstaller file and an HTML page that contains a download link and some information about the packaged app by supplying a few additional arguments in the YAML file:

```
- task: MSBuild@1
  inputs:
    solution: Msix/Msix.wapproj
    platform: $(buildPlatform)
    configuration: $(buildConfiguration)
    msbuildArguments: '/p:OutputPath=NonPackagedApp
/p:UapAppxPackageBuildMode=SideLoadOnly  /p:AppxBundle=Never
/p:GenerateAppInstallerFile=True
/p:AppInstallerUri=http://yourwebsite.com/packages/
/p:AppInstallerCheckForUpdateFrequency=OnApplicationRun
/p:AppInstallerUpdateFrequency=1 /p:AppxPackageDir=$(Build.ArtifactStagingDirectory)/'
  displayName: 'Package the App'
```

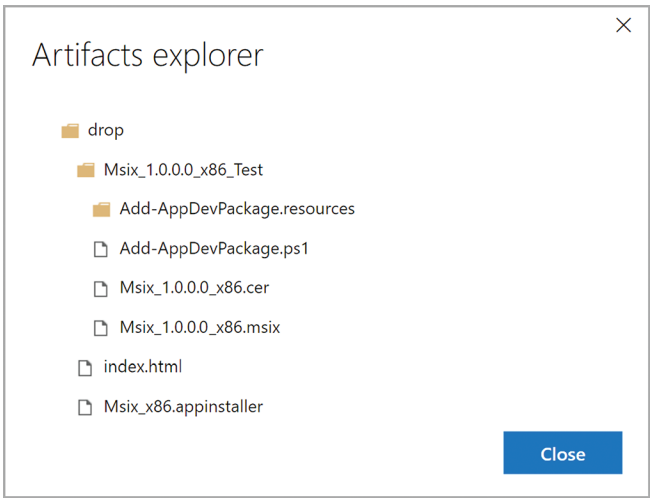


Figure 8 **The Build Artifacts Explorer in the Azure DevOps Portal**

Figure 8 shows the contents of the staging directory on the build agent after running the previous command. It's the same output you get from the wizard in Visual Studio if you choose to create packages for sideloading and check the "Enable automatic updates" checkbox. Using this approach, you can remove the manually created .appinstaller file at the possible expense of some flexibility regarding the configuration of the update behavior.

The generated HTML file includes a hyperlink prefixed with the browser-agnostic ms-appinstaller protocol activation scheme:

```
<a href="ms-appinstaller:?source=
  http://yourwebsite.com/packages/Msix_x86.appinstaller ">Install App</a>
```

If you set up a release pipeline that publishes the contents of the drop folder to your intranet or any other Web site, and the Web server supports byte-range requests and is configured properly, your end users can use this link to directly install the app without downloading the MSIX package first.

Wrapping Up

In this article you've seen how easy it is to package a .NET desktop application as an MSIX using Visual Studio. You also saw how to set up CI and CD pipelines in Azure Pipelines and how to configure automatic updates. MSIX is the modern way to deploy applications on Windows. It's built to be safe, secure and reliable, and lets you and your customers take advantage of the new app model and the modern APIs that have been introduced in Windows 10, regardless of whether you intend to upload your apps to the Microsoft Store or sideload them onto computers in your enterprise. As long as all of your users have moved to Windows 10, you should be able to leverage MSIX to package most Windows desktop apps that exist out there. ■

Magnus Montin *is a Microsoft MVP who works as a self-employed software developer and consultant in Stockholm, Sweden. He specializes in .NET and the Microsoft stack and has over a decade of hands-on experience. You can read his blog at blog.magnusmontin.net.*

Thanks to the following Microsoft technical expert for reviewing this article: *Matteo Pagani*

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

CHICAGO

Navigate Today's Tech in the Windy City

OCTOBER 6-10, 2019
SWISSOTEL, CHICAGO, IL

#VSLive

INTENSE DEVELOPER TRAINING CONFERENCE

In-depth Technical Content On:

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

New This Year!

On-Demand Session Recordings Now Available

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

SAVE $400! When You Register by August 2

Your Adventure Starts Here!

EVENT PARTNER: Microsoft

GOLD SPONSOR: CSC corptax.com

SUPPORTED BY: Visual Studio, msdn magazine, Visual Studio MAGAZINE

PRODUCED BY: CONVERGE 360 An 1105 MEDIA company

vslive.com/chicago

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
- Office 365 for the IT Pro
- PowerShell and DevOps
- Security
- Soft Skills for IT Pros

SAVE $400
WHEN YOU REGISTER BY JUNE 7
Use Promo Code MSDN

EVENT PARTNER Microsoft

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

PRODUCED BY Converge 360 — An 1105 Media company

AGENDA AT-A-GLANCE

Client and Endpoint Management	PowerShell and DevOps	Classic Infrastructure	Soft Skills for IT Pros	Security	Azure/ Public Hybrid	Office 365 for the IT Pro

START TIME	END TIME					
		TechMentor Pre-Conference Hands-On Labs: Monday, August 5, 2019 *(Separate entry fee required)*				
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Light Breakfast				
9:00 AM	12:00 PM	**HOL01** Hands-On Lab: Building a Bulletproof Privileged Access Workstation (PAW) - *Sami Laiho*		**HOL02** Hands-On Lab: Mastering Windows Troubleshooting—Advanced Hands-on Workshop - *Bruce Mackenzie-Low*		
12:00 PM	2:00 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center				
2:00 PM	5:00 PM	Hands-On Lab Continued - *Sami Laiho*		Hands-On Lab Continued - *Bruce Mackenzie-Low*		
6:30 PM	8:30 PM	Dine-A-Round Dinner - Suite in Hyatt Regency Lobby				
START TIME	**END TIME**	**TechMentor Day 1: Tuesday, August 6, 2019**				
7:00 AM	8:00 AM	Registration - Coffee and Light Breakfast				
8:00 AM	9:15 PM	**T01** Microsoft 365 Explained - *Ståle Hansen*	**T02** Top Free Tools for Monitoring Windows Performance - *Bruce Mackenzie-Low*	**T03** Follow the Breadcrumbs Azure Security Center In-Depth - *Mike Nelson*	**T04** From IT Pro to Cloud Pro - *Peter De Tender*	
9:30 AM	10:45 AM	**T05** Using PowerBI and Azure Log Analytics for End User Devices - *Kevin Kaminski*	**T06** How to Become a Community Rockstar—Learn How to Showcase Your Skills - *Cristal Kawula*	**T07** Backup, Distaster Recovery, and Business Continuity in Azure - *Orin Thomas*	**T08** Regex for Complete Noobs - *Thomas Rayner*	
11:00 AM	12:00 PM	**Keynote:** Windows Server 2019 Technical Foundation - *Jeff Woolsey, Principal Program Manager, Windows Server/Hybrid Cloud, Microsoft*				
12:00 PM	1:00 PM	Lunch - McKinley / Visit Exhibitors - Foyer				
1:00 PM	2:15 PM	**T09** Intune and Custom Policies - *Kevin Kaminski*	**T10** A World Beyond Passwords - *Lesha Bhansali & Kristina Cosgrave*	**T11** Configuring Windows Server & Client for Developing Hosted Linux Workloads - *Orin Thomas*	**T12** To Be Announced	
2:15 PM	2:45 PM	Sponsored Break - Visit Exhibitors - Foyer				
2:45 PM	4:00 PM	**T13** OneNote LifeHack: 5 Steps for Succeeding with Personal Productivity - *Ståle Hansen*	**T14** Hardening Windows Server - *Orin Thomas*	**T15** Familiar and Future-Proof: Lift and Shift Your Cluster into Azure IaaS - *Rob Hindman*	**T16** From Cmdlet to Function—Your PowerShell Beginnings - *Mike Nelson*	
4:00 PM	5:30 PM	Exhibitor Reception – Attend Exhibitor Demo - Foyer				
START TIME	**END TIME**	**TechMentor Day 2: Wednesday, August 7, 2019**				
7:30 AM	8:00 AM	Registration - Coffee and Light Breakfast				
8:00 AM	9:15 AM	**W01** Ten Practical Tips to Secure Your Corporate Data with Microsoft 365 - *Peter Daalmans*	**W02** Surviving a Ransomware Attack: Notes from the Field - *Émile Cabot*	**W03** Everything You Need to Know About Storage Spaces Direct—Part 1 - *Cosmos Darwin*	**W04** Azure is 100 % Highly Available or is it? - *Peter De Tender*	
9:30 AM	10:45 AM	**W05** Everything You Need to Know About Calling in Microsoft Teams - *Ståle Hansen*	**W06** Get Started with Azure MFA the Right Way - *Jan Ketil Skanke*	**W07** Everything You Need to Know About Storage Spaces Direct—Part 2 - *Cosmos Darwin*	**W08** Become the World's Greatest Azure-Bender - *Peter De Tender*	
11:00 AM	12:00 PM	**Panel Discussion:** The Future of IT - *Sami Laiho and Dave Kawula (Moderators); Peter De Tender, Thomas Maurer, John O'Neill Sr., and Orin Thomas*				
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch - *McKinley - Visit Exhibitors - Foyer*				
1:00 PM	1:30 PM	Networking Break - Exhibitor Raffle @ 1:10 pm (Must be present to win) - *Foyer in front of Business Center*				
1:30 PM	2:45 PM	**W09** Build Your Azure Infrastructure Like a Pro - *Aleksandar Nikolic*	**W10** Implementing Proactive Security in the Cloud—Part 1 - *Sami Laiho*	**W11** Get the Best of Azure and Windows Server with Windows Admin Center - *Haley Rowland*	**W12** Oww Help! SCCM is Getting SaaSified - *Peter Daalmans*	
3:00 PM	4:15 PM	**W13** Mastering Azure Using Cloud Shell, PowerShell, and Bash! - *Thomas Maurer*	**W14** Implementing Proactive Security in the Cloud—Part 2 - *Sami Laiho*	**W15** Master Software Definded Networking in Windows Server 2019 - *Greg Cusanza*	**W16** Discussion of Modern Device Management from a (Microsoft Surface) Hardware Engineer - *Carl Luberti*	
6:15 PM	9:00 PM	TechMentor's Downtown Seattle Adventure				
START TIME	**END TIME**	**TechMentor Day 3: Thursday, August 8, 2019**				
7:30 AM	8:00 AM	Registration - Coffee and Light Breakfast				
8:00 AM	9:15 AM	**TH01** Hackers Won't Pass—Microsoft 365 Identity & Threat Protection—Part 1 - *Sergey Chubarov*	**TH02** Azure Cloud Shell: The Easiest Way to Manage Azure with Command-line Tools - *Aleksandar Nikolic*	**TH03** Replicate Your Storage/ Data to Azure the Easy Way - *Arpita Duppala*	**TH04** Android Enterprise Management with Intune - *Jan ketil Skanke*	
9:30 AM	10:45 AM	**TH05** Hackers Won't Pass—Microsoft 365 Identity & Threat Protection—Part 2 - *Sergey Chubarov*	**TH06** PowerShell Core on Linux: Benefits and Challenges - *Aleksandar Nikolic*	**TH07** A Look Into the Hybrid Cloud Lifestyle of an Azure Stack Operator - *Thomas Maurer*	**TH08** "AaronLocker:" Robust and Practical Application Whitelisting for Windows - *Aaron Margosis*	
11:00 AM	12:15 PM	**TH09** Using PowerShell to Rock Your Labs - *Dave Kawula*	**TH10** The Force Awakens—Azure SQL Server for the On-prem DBA - *Alexander Arvidsson*	**TH11** 12 Ways to Hack Multi Factor Authentication (MFA) - *Roger Grimes*	**TH12** Windows Autopilot: Modern Device Provisioning - *Michael Niehaus*	
12:15 PM	2:00 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center				
2:00 PM	3:15 PM	**TH13** Migrating to Windows Server 2019 Like the Pro's - *Dave Kawula*	**TH14** Boring is Stable, Stable is Good—Best Practices in Practice for SQL Server - *Alexander Arvidsson*	**TH15** Start Using JEA Today to Stop Overprivileged User Accounts - *John O'Neill Sr.*	**TH16** Wireshark Essentials: Your First Day with Wireshark - *Richard Hicks*	
3:30 PM	4:45 PM	**TH17** The Case of the Shrinking Data: Data Deduplication in Windows Server 2019 - *Dave Kawula*	**TH18** Malware Protection in Windows 10 - *Émile Cabot*	**TH19** Supporting Surfaces? Learn the Surface Diagnostic Toolkit for Business Now - *John O'Neill Sr.*	**TH20** Always On VPN: The Good, The Bad, and the Ugly! - *Richard Hicks*	
START TIME	**END TIME**	**TechMentor Post-Conference Workshops: Friday, August 9, 2019** *(Separate entry fee required)*				
8:30 AM	9:00 AM	Post-Conference Workshop Registration - Coffee and Light Breakfast				
9:00 AM	12:00 PM	**F01** Workshop: Enhance Security While Increasing Your Admin Superpowers - *John O'Neill Sr.*		**F02** Workshop: Building Real World Labs in Azure - *Dave Kawula*		
12:00 PM	1:00 PM	Lunch - *McKinley*				
1:00 PM	4:00 PM	Workshop Continued - *John O'Neill Sr.*		Workshop Continued - *Dave Kawula*		

Logo denotes sessions with a Microsoft speaker.

Speakers and sessions subject to change

CONNECT WITH TECHMENTOR

Twitter
@TechMentorEvent

Facebook
Search "TechMentor"

LinkedIn
Search "TechMentor"

TechMentorEvents.com/MicrosoftHQ

Text-to-Speech Synthesis in .NET

Ilia Smirnov

I often fly to Finland to see my mom. Every time the plane lands in Vantaa airport, I'm surprised at how few passengers head for the airport exit. The vast majority set off for connecting flights to destinations spanning all of Central and Eastern Europe. It's no wonder, then, that when the plane begins its descent, there's a barrage of announcements about connecting flights. "If your destination is Tallinn, look for gate 123," "For flight XYZ to Saint Petersburg, proceed to gate 234," and so on. Of course, flight attendants don't typically speak a dozen languages, so they use English, which is not the native language of most passengers. Considering the quality of the public announcement (PA) systems on the airliners, plus engine noise, crying babies and other disturbances, how can any information be effectively conveyed?

Well, each seat is equipped with headphones. Many, if not all, long-distance planes have individual screens today (and local ones have at least different audio channels). What if a passenger could choose the language for announcements and an onboard computer system allowed flight attendants to create and send dynamic (that is, not pre-recorded) voice messages? The key challenge here is the dynamic nature of the messages. It's easy to pre-record safety instructions, catering options and so on, because they're rarely updated. But we need to create messages literally on the fly.

This article discusses:

- Computer-based speech synthesis
- Concatenative unit selection text-to-speech (TTS) systems
- Machine learning in TTS processing

Technologies discussed:

.NET Speech API, .NET Speech SDK, Microsoft Cognitive Services, Speech Synthesis Markup Language (SSML)

Fortunately, there's a mature technology that can help: text-to-speech synthesis (TTS). We rarely notice such systems, but they're ubiquitous: public announcements, prompts in call centers, navigation devices, games, smart devices and other applications are all examples where pre-recorded prompts aren't sufficient or using a digitized waveform is proscribed due to memory limitations (a text read by a TTS engine is much smaller to store than a digitized waveform).

Computer-based speech synthesis is hardly new. Telecom companies invested in TTS to overcome the limitations of pre-recorded messages, and military researchers have experimented with voice prompts and alerts to simplify complex control interfaces. Portable synthesizers have likewise been developed for people with disabilities. For an idea of what such devices were capable of 25 years ago, listen to the track "Keep Talking" on the 1994 Pink Floyd album "The Division Bell," where Stephen Hawking says his famous line: "All we need to do is to make sure we keep talking."

TTS APIs are often provided along with their "opposite"—speech recognition. While you need both for effective human-computer interaction, this exploration is focused specifically on speech synthesis. I'll use the Microsoft .NET TTS API to build a prototype of an airliner PA system. I'll also look under the hood to understand the basics of the "unit selection" approach to TTS. And while I'll be walking through the construction of a desktop application, the principles here apply directly to cloud-based solutions.

Roll Your Own Speech System

Before prototyping the in-flight announcement system, let's explore the API with a simple program. Start Visual Studio and create a console application. Add a reference to System.Speech and implement the method in **Figure 1**.

Now compile and run. Just a few lines of code and you've replicated the famous Hawking phrase.

Figure 1 **System.Speech.Synthesis Method**

```
using System.Speech.Synthesis;

namespace KeepTalking
{
  class Program
  {
    static void Main(string[] args)
    {
      var synthesizer = new SpeechSynthesizer();
      synthesizer.SetOutputToDefaultAudioDevice();
      synthesizer.Speak("All we need to do is to make sure we keep talking");
    }
  }
}
```

When you were typing this code, IntelliSense opened a window with all the public methods and properties of the SpeechSynthesizer class. If you missed it, use "Control-Space" or the "dot" keyboard shortcut (or look at bit.ly/2PCWpat). What's interesting here?

First, you can set different output targets. It can be an audio file or a stream or even null. Second, you have both synchronous (as in the previous example) and asynchronous output. You can also adjust the volume and the rate of speech, pause and resume it, and receive events. You can also select voices. This feature is important here, because you'll use it to generate output in different languages. But what voices are available? Let's find out, using the code in **Figure 2**.

On my machine with Windows 10 Home the resulting output from **Figure 2** is:

```
Id: TTS_MS_EN-US_DAVID_11.0 | Name: Microsoft David Desktop |
  Age: Adult | Gender: Male | Culture: en-US
Id: TTS_MS_EN-US_ZIRA_11.0 | Name: Microsoft Zira Desktop |
  Age: Adult | Gender: Female | Culture: en-US
```

There are only two English voices available, and what about other languages? Well, each voice takes some disk space, so they're not installed by default. To add them, navigate to Start | Settings | Time & Language | Region & Language and click Add a language, making sure to select Speech in optional features. While Windows supports more than 100 languages, only about 50 support TTS. You can review the list of supported languages at bit.ly/2UNNvba.

After restarting your computer, a new language pack should be available. In my case, after adding Russian, I got a new voice installed:

```
Id: TTS_MS_RU-RU_IRINA_11.0 | Name: Microsoft Irina Desktop |
  Age: Adult | Gender: Female | Culture: ru-RU
```

Now you can return to the first program and add these two lines instead of the synthesizer.Speak call:

```
synthesizer.SelectVoice("Microsoft Irina Desktop");
synthesizer.Speak("Всё, что нам нужно сделать, это продолжать говорить");
```

If you want to switch between languages, you can insert SelectVoice calls here and there. But a better way is to add some structure to speech. For that, let's use the PromptBuilder class, as shown in **Figure 3**.

Notice that you have to call EndVoice, otherwise you'll get a runtime error. Also, I used CultureInfo as another way to specify a language. PromptBuilder has lots of useful methods, but I want to draw your attention to AppendTextWithHint. Try this code:

```
var builder = new PromptBuilder();
builder.AppendTextWithHint("3rd", SayAs.NumberOrdinal);
builder.AppendBreak();
builder.AppendTextWithHint("3rd", SayAs.NumberCardinal);
synthesizer.Speak(builder);
```

Another way to structure input and specify how to read it is to use Speech Synthesis Markup Language (SSML), which is a cross-platform recommendation developed by the international Voice Browser Working Group (w3.org/TR/speech-synthesis). Microsoft TTS engines provide comprehensive support for SSML. This is how to use it:

```
string phrase = @"<speak version=""1.0""
  xmlns=""http://www.w3.org/2001/10/synthesis""
  xml:lang=""en-US"">";
phrase += @"<say-as interpret-as=""ordinal"">3rd</say-as>";
phrase += @"<break time=""1s""/>";
phrase += @"<say-as interpret-as=""cardinal"">3rd</say-as>";
phrase += @"</speak>";
synthesizer.SpeakSsml(phrase);
```

Notice it employs a different call on the SpeechSynthesizer class.

Now you're ready to work on the prototype. This time create a new Windows Presentation Foundation (WPF) project. Add a form and a couple of buttons for prompts in two different languages. Then add click handlers as shown in the XAML in **Figure 4**.

Obviously, this is just a tiny prototype. In real life, PopulateMessages will probably read from an external resource. For example, a flight attendant can generate a file with messages in multiple languages by using an application that calls a service like Bing Translator (bing.com/translator). The form will be much more sophisticated and dynamically generated based on available languages.

Figure 2 **Voice Info Code**

```
using System;
using System.Speech.Synthesis;

namespace KeepTalking
{
  class Program
  {
    static void Main(string[] args)
    {
      var synthesizer = new SpeechSynthesizer();
      foreach (var voice in synthesizer.GetInstalledVoices())
      {
        var info = voice.VoiceInfo;
        Console.WriteLine($"Id: {info.Id} | Name: {info.Name} |
          Age: {info.Age} | Gender: {info.Gender} | Culture: {info.Culture}");
      }
      Console.ReadKey();
    }
  }
}
```

Figure 3 **The PromptBuilder Class**

```
using System.Globalization;
using System.Speech.Synthesis;

namespace KeepTalking
{
  class Program
  {
    static void Main(string[] args)
    {
      var synthesizer = new SpeechSynthesizer();
      synthesizer.SetOutputToDefaultAudioDevice();
      var builder = new PromptBuilder();
      builder.StartVoice(new CultureInfo("en-US"));
      builder.AppendText("All we need to do is to keep talking.");
      builder.EndVoice();
      builder.StartVoice(new CultureInfo("ru-RU"));
      builder.AppendText("Всё, что нам нужно сделать, это продолжать говорить");
      builder.EndVoice();
      synthesizer.Speak(builder);
    }
  }
}
```

There will be error handling and so on. But the point here is to illustrate the core functionality.

Deconstructing Speech

So far we've achieved our objective with a surprisingly small codebase. Let's take an opportunity to look under the hood and better understand how TTS engines work.

There are many approaches to constructing a TTS system. Historically, researchers have tried to discover a set of pronunciation rules on which to build algorithms. If you've ever studied a foreign language, you're familiar with rules like "Letter 'c' before 'e,' 'i,' 'y' is pronounced as 's' as in 'city,' but before 'a,' 'o,' 'u' as 'k' as in 'cat.'" Alas, there are so many exceptions and special cases—like pronunciation changes in consecutive words—that constructing a comprehensive set of rules is difficult. Moreover, most such systems tend to produce a distinct "machine" voice—imagine a beginner in a foreign language pronouncing a word letter-by-letter.

For more naturally sounding speech, research has shifted toward systems based on large databases of recorded speech fragments, and these engines now dominate the market. Commonly known as *concatenation unit selection* TTS, these engines select speech samples (units) based on the input text and concatenate them into phrases. Usually, engines use two-stage processing closely resembling compilers: First, parse input into an internal list- or tree-like structure with phonetic transcription and additional metadata, and then synthesize sound based on this structure.

Because we're dealing with natural languages, parsers are more sophisticated than for programming languages. So beyond tokenization (finding boundaries of sentences and words), parsers must correct typos, identify parts of speech, analyze punctuation, and decode abbreviations, contractions and special symbols. Parser output is typically split by phrases or sentences, and formed into collections describing words that group and carry metadata such as part of speech, pronunciation, stress and so on.

Parsers are responsible for resolving ambiguities in the input. For example, what is "Dr."? Is it "doctor" as in "Dr. Smith," or "drive" as in "Privet Drive?" And is "Dr." a sentence because it starts with an uppercase letter and ends with a period? Is "project" a noun or a verb? This is important to know because the stress is on different syllables.

These questions are not always easy to answer and many TTS systems have separate parsers for specific domains: numerals, dates, abbreviations, acronyms, geographic names, special forms of text like URLs and so on. They're also language- and region-specific. Luckily, such problems have been studied for a long time and we have well-developed frameworks and libraries to lean on.

The next step is generating pronunciation forms, such as tagging the tree with sound symbols (like transforming "school" to "s

Figure 4 **The XAML Code**

```
using System.Collections.Generic;
using System.Globalization;
using System.Speech.Synthesis;
using System.Windows;

namespace GuiTTS
{
  public partial class MainWindow : Window
  {
    private const string en = "en-US";
    private const string ru = "ru-RU";
    private readonly IDictionary<string, string> _messagesByCulture =
      new Dictionary<string, string>();

    public MainWindow()
    {
      InitializeComponent();
      PopulateMessages();
    }

    private void PromptInEnglish(object sender, RoutedEventArgs e)
    {
      DoPrompt(en);
    }

    private void PromptInRussian(object sender, RoutedEventArgs e)
    {
      DoPrompt(ru);
    }

    private void DoPrompt(string culture)
    {
      var synthesizer = new SpeechSynthesizer();
      synthesizer.SetOutputToDefaultAudioDevice();
      var builder = new PromptBuilder();
      builder.StartVoice(new CultureInfo(culture));
      builder.AppendText(_messagesByCulture[culture]);
      builder.EndVoice();
      synthesizer.Speak(builder);
    }

    private void PopulateMessages()
    {
      _messagesByCulture[en] = "For the connection flight 123 to
        Saint Petersburg, please, proceed to gate A1";
      _messagesByCulture[ru] =
        "Для пересадки на рейс 123 в Санкт-Петербург, пожалуйста, пройдите к выходу A1";
    }
  }
}
```

Neural Networks in TTS

Statistical or machine learning methods have for years been applied in all stages of TTS processing. For example, Hidden Markov Models are used to create parsers producing the most likely parse, or to perform labeling for speech sample databases. Decision trees are used in unit selection or in grapheme-to-phoneme algorithms, while neural networks and deep learning have emerged at the bleeding edge of TTS research.

We can consider an audio sample as a time-series of waveform sampling. By creating an auto-regressive model, it's possible to predict the next sample. As a result, the model generates speech-kind bubbling, like a baby learning to talk by imitating sounds. If we further condition this model on the audio transcript or the pre-processing output from an existing TTS system, we get a parameterized model of speech. The output of the model describes a spectrogram for a vocoder producing actual waveforms. Because this process doesn't rely on a database with recorded samples, but is generative, the model has a small memory footprint and allows for adjustment of parameters.

Because the model is trained on natural speech, the output retains all of its characteristics, including breathing, stresses and intonation (so neural networks can potentially solve the prosody problem). It's possible also to adjust the pitch, create a completely different voice and even imitate singing.

At the time of this writing, Microsoft is offering its preview version of a neural network TTS (**bit.ly/2PAYXWN**). It provides four voices with enhanced quality and near instantaneous performance.

k uh l"). This is done by special grapheme-to-phoneme algorithms. For languages like Spanish, some relatively straightforward rules can be applied. But for others, like English, pronunciation differs significantly from the written form. Statistical methods are then employed along with databases for known words. After that, additional post-lexical processing is needed, because the pronunciation of words can change when combined in a sentence.

While parsers try to extract all possible information from the text, there's something that's so elusive that it's not extractable: prosody or intonation. While speaking, we use prosody to emphasize certain words, to convey emotion, and to indicate affirmative sentences, commands and questions. But written text doesn't have symbols to indicate prosody. Sure, punctuation offers some context: A comma means a slight pause, while a period means a longer one, and a question mark means you raise your intonation toward the end of a sentence. But if you've ever read your children a bedtime story, you know how far these rules are from real reading.

Moreover, two different people often read the same text differently (ask your children who is better at reading bedtime stories—you or your spouse). Because of this you cannot reliably use statistical methods since different experts will produce different labels for supervised learning. This problem is complex and, despite intensive research, far from being solved. The best programmers can do is use SSML, which has some tags for prosody.

Speech Generation

Now that we have the tree with metadata, we turn to speech generation. Original TTS systems tried to synthesize signals by combining sinusoids. Another interesting approach was constructing a system of differential equations describing the human vocal tract as several connected tubes of different diameters and lengths. Such solutions are very compact, but unfortunately sound quite mechanical. So, as with musical synthesizers, the focus gradually shifted to solutions based on samples, which require significant space, but essentially sound natural.

To build such a system, you have to have many hours of high-quality recordings of a professional actor reading specially constructed text. This text is split into units, labeled and stored into a database. Speech generation becomes a task of selecting proper units and gluing them together.

Because you're not synthesizing speech, you can't significantly adjust parameters in the runtime. If you need both male and female voices or must provide regional accents (say, Scottish or Irish), they have to be recorded separately. The text must be constructed to cover all possible sound units you'll need. And the actors must read in a neutral tone to make concatenation easier.

Splitting and labeling are also non-trivial tasks. It used to be done manually, taking weeks of tedious work. Thankfully, machine learning is now being applied to this.

Unit size is probably the most important parameter for a TTS system. Obviously, by using whole sentences, we could make the most natural sounds even with correct prosody, but recording and storing that much data is impossible. Can we split it into words? Probably, but how long will it take for an actor to read an entire dictionary? And what database size limitations are we facing? On the other side, we cannot just record the alphabet—that's sufficient only for a spelling bee contest. So usually units are selected as two three-letter groups. They're not necessarily syllables, as groups spanning syllable borders can be glued together much better.

Now the last step. Having a database of speech units, we need to deal with concatenation. Alas, no matter how neutral the intonation was in the original recording, connecting units still requires adjustments to avoid jumps in volume, frequency and phase. This is done with digital signal processing (DSP). It can also be used to add some intonation to phrases, like raising or lowering the generated voice for assertions or questions.

Wrapping Up

In this article I covered only the .NET API. Other platforms provide similar functionality. MacOS has NSSpeechSynthesizer in Cocoa with comparable features, and most Linux distributions include the eSpeak engine. All of these APIs are accessible through native code, so you have to use C# or C++ or Swift. For cross-platform ecosystems like Python, there are some bridges like Pyttsx, but they usually have certain limitations.

Cloud vendors, on the other hand, target wide audiences, and offer services for most popular languages and platforms. While functionality is comparable across vendors, support for SSML tags can differ, so check documentation before choosing a solution.

Microsoft offers a Text-to-Speech service as part of Cognitive Services (bit.ly/2XWorku). It not only gives you 75 voices in 45 languages, but also allows you to create your own voices. For that, the service needs audio files with a corresponding transcript. You can write your text first then have someone read it, or take an existing recording and write its transcript. After uploading these datasets to Azure, a machine learning algorithm trains a model for your own unique "voice font." A good step-by-step guide can be found at bit.ly/2VE8th4.

A very convenient way to access Cognitive Speech Services is by using the Speech Software Development Kit (bit.ly/2DDTh9I). It supports both speech recognition and speech synthesis, and is available for all major desktop and mobile platforms and most popular languages. It's well documented and there are numerous code samples on GitHub.

TTS continues to be a tremendous help to people with special needs. For example, check out linka.su, a Web site created by a talented programmer with cerebral paralysis to help people with speech and musculoskeletal disorders, autism, or those recovering from a stroke. Knowing from personal experience what limitations they're facing, the author created a range of applications for people who can't type on a regular keyboard, can only select one letter at a time, or just touch a picture on a tablet. Thanks to TTS, he literally gives a voice to those who do not have one. I wish that we all, as programmers, could be that useful to others. ■

Ilia Smirnov *has more than 20 years of experience developing enterprise applications on major platforms, primarily in Java and .NET. For the last decade, he has specialized in simulation of financial risks. He holds three master's degrees, FRM and other professional certifications.*

Thanks to the following Microsoft technical expert for reviewing this article: *Sheng Zhao*

Visual Studio LIVE! EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS | SQL Server LIVE! TRAINING FOR DBAs AND IT PROS | TECHMENTOR IN-DEPTH TRAINING FOR IT PROS | Office & SharePoint LIVE! ON-PREMISE, CLOUD, & CROSS-PLATFORM TRAINING | Artificial Intelligence LIVE! AI FOR DEVELOPERS AND DATA SCIENTISTS | Cloud & Containers LIVE! CLOUD-NATIVE, PAAS & SERVERLESS COMPUTING

The Ultimate Education Destination

LIVE! 360

TECH EVENTS WITH PERSPECTIVE

6 Great Events, 1 Low Price!

November 17-22, 2019 | ORLANDO
Royal Pacific Resort at Universal

Live! 360 brings the IT and developer community together for a unique conference, featuring 6 co-located conferences for (almost) every IT title. Customize your learning by choosing from hundreds of sessions, dozens of track topics, workshops and hands-on labs from hundreds of expert speakers.

EVENT PARTNER Microsoft

SILVER SPONSOR aws

SUPPORTED BY msdn magazine | Redmond Channel Partner | Redmond MAGAZINE | VIRTUALIZATION & Cloud Review | Visual Studio MAGAZINE

PRODUCED BY CONVERGE 360 An 1105 MEDIA company

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

Visual Studio Live! features unbiased and practical developer training on the Microsoft Platform. Explore hot topics such as ASP.NET Core, JavaScript, Xamarin, Database Analytics and more!

SQL Server LIVE!

TRAINING FOR DBAs AND IT PROS

Whether you are a DBA, developer, IT Pro or Analyst, SQL Server Live! will provide you with the skills you need to drive your data to succeed.

Office & SharePoint LIVE!

ON-PREMISE, CLOUD, & CROSS-PLATFORM TRAINING

Office & SharePoint Live! provides leading-edge training to administrators and developers who must customize, deploy and maintain SharePoint Server on-premises and in Office 365.

TECHMENTOR

IN-DEPTH TRAINING FOR IT PROS

TechMentor gives a strong emphasis on doing more with the tech you already own plus solid coverage of what is next - striking the perfect balance of training essentials for the everyday IT Pro.

Artificial Intelligence LIVE!

AI FOR DEVELOPERS AND DATA SCIENTISTS

Artificial Intelligence Live! offers real-world training on the languages, libraries, APIs, tools and cloud services you need to implement real AI and Machine Learning solutions today and into the future.

NEW! IN 2019 Cloud & Containers LIVE!

CLOUD-NATIVE, PAAS & SERVERLESS COMPUTING

Cloud & Containers Live! covers both IT infrastructure and software development aspects of cloud-native software design, development, release, deployment, operations, instrumentation/monitoring and maintenance.

Who Should Attend:

- Developers
- IT Professionals
- Database Administrators
- Business Intelligence Professionals
- SharePoint Professionals
- and more!

Save $500 when you register by August 16!

Promo Code MSDN

live360events.com/orlando

CONNECT WITH LIVE! 360

twitter.com/live360
@live360

facebook.com
Search "Live 360"

instagram.com
@live360_events

linkedin.com
Join the "Live! 360" group!

DINO ESPOSITO

Revisiting the ASP.NET Core Pipeline

Nearly any server-side processing environment has its own pipeline of pass-through components to inspect, re-route or modify incoming requests and the outgoing responses. Classic ASP.NET had it arranged around the idea of HTTP modules, whereas ASP.NET Core employs the more modern architecture based on middleware components. At the end of the day, the purpose is the same—letting configurable external modules intervene in the way the request (and later the response) goes through in the server environment. The primary purpose of middleware components is altering, and filtering in some way, the flow of data (and in some specific cases just short-circuiting the request, stopping any further processing).

The ASP.NET Core pipeline is nearly unchanged since version 1.0 of the framework, but the upcoming release of ASP.NET Core 3.0 invites a few remarks on the current architecture that have gone for the most part unnoticed. So, in this article, I'll revisit the overall functioning of the ASP.NET Core runtime pipeline and focus on the role and possible implementation of HTTP endpoints.

ASP.NET Core for the Web Back End

Especially in the past couple of years, building Web applications with the front end and back end completely decoupled has become quite common. Therefore, most ASP.NET Core projects are today simple Web API, UI-less projects that just provide an HTTP façade to a single-page and/or mobile application built, for the most part, with Angular, React, Vue and their mobile counterparts.

When you realize this, a question pops up: In an application that's not using any Razor facilities, does it still make sense to bind to the MVC application model? The MVC model doesn't come for free, and in fact, to some extent, it may not even be the most lightweight option once you stop using controllers to serve action results. To press the question even further: Is the action result concept itself strictly necessary if a significant share of the ASP.NET Core code is written just to return JSON payloads?

With these thoughts in mind, let's review the ASP.NET Core pipeline and the internal structure of middleware components and the list of built-in runtime services you can bind to during startup.

The Startup Class

In any ASP.NET Core application, one class is designated as the application bootstrapper. Most of the time, this class takes the name of Startup. The class is declared as a startup class in the configuration of the Web host and the Web host instantiates and invokes it via reflection. The class can have two methods—ConfigureServices (optional) and Configure. In the first method, you receive the current (default) list of runtime services and are expected to add more services to prepare the ground for the actual application logic. In the Configure method, you configure for both the default services and for those you explicitly requested to support your application.

The Configure method receives at least an instance of the application builder class. You can see this instance as a working instance of the ASP.NET runtime pipeline passed to your code to be configured as appropriate. Once the Configure method returns, the pipeline workflow is fully configured and will be used to carry on any further request sent from connected clients. **Figure 1** provides a sample implementation of the Configure method of a Startup class.

The Use extension method is the principal method you employ to add middleware code to the otherwise empty pipeline workflow. Note that the more middleware you add, the more work the server needs to do to serve any incoming requests. The most minimal is the pipeline, the fastest will be the time-to-first-byte (TTFB) for the client.

You can add middleware code to the pipeline using either lambdas or ad hoc middleware classes. The choice is up to you: The lambda is more direct, but the class (and preferably some extension methods) will make the whole thing easier to read and maintain. The middleware code gets the HTTP context of the request and a reference to the next middleware in the pipeline, if any. **Figure 2** presents an overall view of how the various middleware components link together.

Each middleware component is given a double chance to intervene in the life of the ongoing request. It can pre-process the request as received from the chain of components registered to run earlier,

Figure 1 **Basic Example of the Configure Method in the Startup Class**

```
public void Configure(IApplicationBuilder app)
{
  app.Use(async (context, nextMiddleware) =>
  {
    await context.Response.WriteAsync("BEFORE");
    await nextMiddleware();
    await context.Response.WriteAsync("AFTER");
  });

  app.Run(async (context) =>
  {
    var obj = new SomeWork();
    await context
      .Response
      .WriteAsync("<h1 style='color:red;'>" +
                  obj.SomeMethod() +
                 "</h1>");
  });
}
```

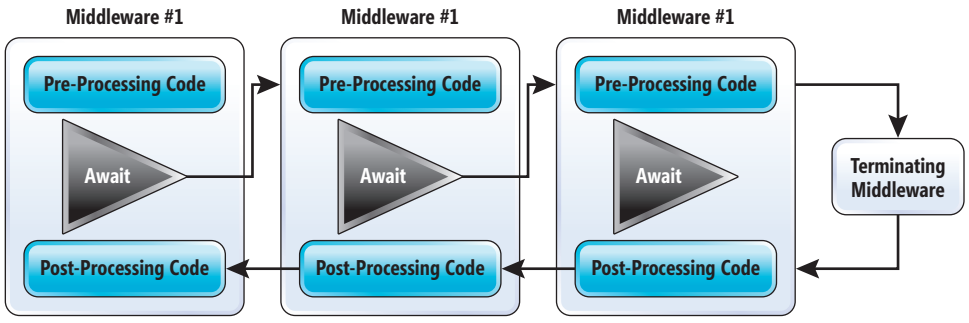


Figure 2 **The ASP.NET Core Runtime Pipeline**

and then it's expected to yield to the next component in the chain. When the last component in the chain gets its chance to pre-process the request, the request is passed to the terminating middleware for the actual processing aimed at producing a concrete output. After that, the chain of components is walked back in the reverse order as shown in **Figure 2**, and each middleware has its second chance to process—this time, though, it will be a post-processing action. In the code of **Figure 1**, the separation between pre-processing and post-processing code is the line:

```
await nextMiddleware();
```

> A middleware component that deliberately doesn't yield to the next actually terminates the chain, causing the response to be sent to the requesting client.

The Terminating Middleware

Key in the architecture shown in **Figure 2** is the role of the terminating middleware, which is the code at the bottom of the Configure method that terminates the chain and processes the request. All demo ASP.NET Core applications have a terminating lambda, like so:

```
app.Run(async (context) => { ... };
```

Figure 3 **A Very Minimal ASP.NET Core Web API**

```
public void Configure(IApplicationBuilder app,
                      ICountryRepository country)
{
  app.Map("/country", countryApp =>
  {
    countryApp.Run(async (context) =>
    {
      var query = context.Request.Query["q"];
      var list = country.AllBy(query).ToList();
      var json = JsonConvert.SerializeObject(list);
      await context.Response.WriteAsync(json);
    });
  });

  // Work as a catch-all
  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Invalid call");
  }
});
```

The lambda receives an HttpContext object and does whatever it's supposed to do in the context of the application.

A middleware component that deliberately doesn't yield to the next actually terminates the chain, causing the response to be sent to the requesting client. A good example of this is the UseStaticFiles middleware, which serves a static resource under the specified Web root folder and terminates the request. Another example is UseRewriter, which may be able to order a client redirect to a new URL. Without a terminating middleware, a request can hardly result in some visible output on the client, though a response is still sent as modified by running middleware, for example through the addition of HTTP headers or response cookies.

There are two dedicated middleware tools that can also be used to short circuit the request: app.Map and app.MapWhen. The former checks if the request path matches the argument and runs its own terminating middleware, as shown here:

```
app.Map("/now", now =>
{
  now.Run(async context =>
  {
    var time = DateTime.UtcNow.ToString("HH:mm:ss");
    await context
      .Response
      .WriteAsync(time);
  });
});
```

The latter, instead, runs its own terminating middleware only if a specified Boolean condition is verified. The Boolean condition results from the evaluation of a function that accepts an HttpContext. The code in **Figure 3** presents a very thin and minimal Web API that only serves a single endpoint and does that without anything like a controller class.

If the URL matches /country, the terminating middleware reads a parameter from the query string and arranges a call to the repository to get the list of matching countries. The list object is then manually serialized in a JSON format directly to the output stream. By just adding a few other map routes you could even extend your Web API. It can hardly be simpler than that.

What About MVC?

In ASP.NET Core, the whole MVC machinery is offered as a black-box runtime service. All you do is bind to the service in the ConfigureServices method and configure its routes in the Configure method, as shown in the code here:

```
public void ConfigureServices(IServiceCollection services)
{
  // Bind to the MVC machinery
  services.AddMvc();
}
public void Configure(IApplicationBuilder app)
{
  // Use the MVC machinery with default route
  app.UseMvcWithDefaultRoute();

  // (As an alternative) Use the MVC machinery with attribute routing
  app.UseMvc();
}
```

At that point, you're welcome to populate the familiar Controllers folder and even the Views folder if you intend to serve HTML. Note that in ASP.NET Core you can also use POCO controllers, which are plain C# classes decorated to be recognized as controllers and disconnected from the HTTP context.

The MVC machinery is another great example of terminating middleware. Once the request is captured by the MVC middleware, everything goes under its control and the pipeline is abruptly terminated.

It's interesting to notice that internally the MVC machinery runs its own custom pipeline. It's not middleware-centric, but it's nonetheless a full-fledged runtime pipeline that controls how requests are routed to the controller action method, with the generated action result finally rendered to the output stream. The MVC pipeline is made of various types of action filters (action name selectors, authorization filters, exception handlers, custom action result managers) that run before and after each controller method. In ASP.NET Core content negotiation is also buried in the runtime pipeline.

At a more insightful look, the whole ASP.NET MVC machinery looks like it's bolted on top of the newest and redesigned middleware-centric pipeline of ASP.NET Core. It's like the ASP.NET Core pipeline and the MVC machinery are entities of different types just connected together in some way. The overall picture is not much different from the way MVC was bolted on top of the now-dismissed Web Forms runtime. In that context, in fact, MVC kicked in through a dedicated HTTP handler if the processing request couldn't be matched to a physical file (most likely an ASPX file).

Is this a problem? Probably not. Or probably not yet!

Putting SignalR in the Loop

When you add SignalR to an ASP.NET Core application, all you need to do is create a hub class to expose your endpoints. The interesting thing is that the hub class can be completely unrelated to controllers. You don't need MVC to run SignalR, yet the hub class behaves like a front-end controller for external requests. A method exposed from a hub class can perform any work—even work unrelated to the cross-app notification nature of the framework, as shown in **Figure 4**.

Can you see the picture?

The SignalR hub class can be seen as a controller class, without the whole MVC machinery, ideal for UI-less (or, rather, Razor-less) responses.

Figure 4 **Exposing a Method from a Hub Class**

```
public class SomeHub : Hub
{
  public void Method1()
  {
    // Some logic
    ...
    Clients.All.SendAsync("...");
  }

  public void Method2()
  {
    // Some other logic
    ...
    Clients.All.SendAsync("...");
  }
}
```

Putting gRPC in the Loop

In version 3.0, ASP.NET Core also provides native support for the gRPC framework. Designed along with the RPC guidelines, the framework is a shell of code around an interface definition language that fully defines the endpoint and is able to trigger communication between the connected parties using Protobuf binary serialization over HTTP/2. From the ASP.NET Core 3.0 perspective, gRPC is yet another invokable façade that can make server-side calculations and return values. Here's how to enable an ASP.NET Core server application to support gRPC:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddGrpc();
}
public void Configure(IApplicationBuilder app)
{
  app.UseRouting(routes =>
    {
      routes.MapGrpcService<GreeterService>();
    });
}
```

Note also the use of global routing to enable the application to support routes without the MVC machinery. You can think of UseRouting as a more structured way of defining app.Map middleware blocks.

The net effect of the previous code is to enable RPC-style calls from a client application to the mapped service—the GreeterService class. Interestingly, the GreeterService class is conceptually equivalent to a POCO controller, except that it has no need to be recognized as a controller class, as shown here:

```
public class GreeterService : Greeter.GreeterBase
{
  public GreeterService(ILogger<GreeterService> logger)
  {
  }
}
```

The base class—GreeterBase is an abstract class wrapped in a static class—contains the necessary plumbing to carry out the request/response traffic. The gRPC service class is fully integrated with the ASP.NET Core infrastructure and can be injected with external references.

The Bottom Line

Especially with the release of ASP.NET Core 3.0, there will be two more scenarios in which having an MVC-free controller-style façade would be helpful. SignalR has hub classes and gRPC has a service class, but the point is that they're conceptually the same thing that has to be implemented in different ways for different scenarios. The MVC machinery was ported to ASP.NET Core more or less as it was originally devised for classic ASP.NET, and it maintains its own internal pipeline around controllers and action results. At the same time, as ASP.NET Core is more and more used as a plain provider of back-end services, with no support for views, the need for a possibly unified, RPC-style façade for HTTP endpoints grows. ■

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: *Marco Cecconi*

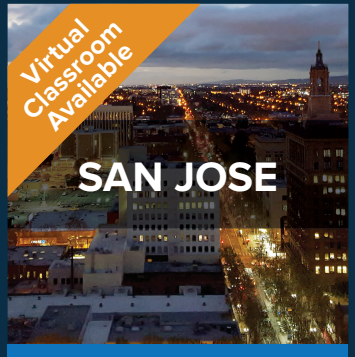
Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

Developer Training Conferences and Events

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



June 18-19

Develop an ASP.NET Core 2.x Service and Website with EF Core 2.x

REGISTER NOW!
vslive.com/sanjose

July 23

Developer's Guide to Getting Started with UX

REGISTER NOW!
vslive.com/virtual

August 12-16

Microsoft HQ

REGISTER NOW!
vslive.com/microsofthq

September 17-18

Developing Modern Web Apps with Azure

REGISTER NOW!
vslive.com/atlanta

Sept. 29-Oct. 3

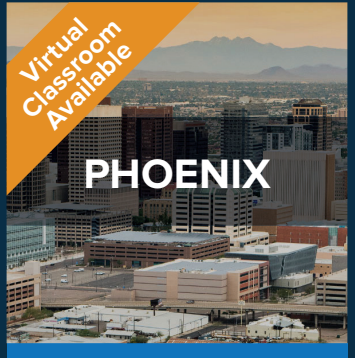
Westin Gaslamp Quarter

REGISTER NOW!
vslive.com/sandiego

October 6-10

Swissotel

REGISTER NOW!
vslive.com/chicago



October 29-30

Practical Real-World SQL Server Performance for Everyone

REGISTER NOW!
sqllive360.com/phoenix

November 17-22

Royal Pacific Resort at Universal

REGISTER NOW!
vslive.com/orlando

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE 360 An 1105 MEDIA company

vslive.com #VSLIVE

JAMES MCCAFFREY

Simplified Naive Bayes Classification Using C#

The goal of a naive Bayes classification problem is to predict a discrete value. For example, you might want to predict the authenticity of a gemstone based on its color, size and shape (0 = fake, 1 = authentic). In this article I show how to implement a simplified naive Bayes classification algorithm using the C# language.

The best way to understand where this article is headed is to take a look at the demo run in **Figure 1**. The demo program sets up 40 dummy data items. Each item has three predictor values: color (Aqua, Blue, Cyan, Dune), size (Small, Large), and shape (Pointed, Rounded, Twisted), followed by a binary class to predict (0 or 1).

The demo sets up an item to predict: (Cyan, Small, Twisted). Naive Bayes classification is based on probabilities, which in turn are based on counts in the data. The demo scans the 40-item dataset and computes and displays six joint counts: items that have both Cyan and 0 (3 items), Cyan and 1 (5), Small and 0 (18), Small and 1 (15), Twisted and 0 (5), Twisted and 1 (3). The demo also counts the number of class 0 items (24) and class 1 items (16).

> In my opinion, naive Bayes classification is best explained using a concrete example.

Using the count information, the demo calculates intermediate values called evidence terms (0.0082, 0.0131) and then uses the evidence terms to calculate pseudo-probabilities (0.3855, 0.6145) that correspond to predictions of class 0 and class 1. Because the second pseudo-probability is larger, the conclusion is that (Cyan, Small, Twisted) is class 1.

This article assumes you have intermediate or better programming skill with C# or a C-family language such as Python or Java, but doesn't assume you know anything about naive Bayes classification. The complete demo code and the associated data are presented in this article. The source code and the data are also available in the accompanying download. All normal error checking has been removed to keep the main ideas as clear as possible.

Code download available at msdn.com/magazine/0619magcode.

Understanding Naive Bayes Classification

Naive Bayes classification is both simple and complicated. Implementation is relatively simple, but the underlying math ideas are very complex. There are many different math equations that define naive Bayes classification. Three examples are shown in **Figure 2**. Letter P means probability; Ck is one of k classes (0, 1, 2, . . .); the pipe symbol is read "given that"; and X is a vector of input values such as (Cyan, Small, Twisted). Unfortunately, these equations don't provide much help when implementing naive Bayes classification until after you understand the technique.

```
C:\NaiveBayes\CSharp\bin\Debug\CSharp.exe

Begin simple naive Bayes demo

Training data:
[0] Aqua Small Twisted 1
[1] Blue Small Pointed 0
[2] Dune Large Rounded 0
[3] Dune Small Rounded 1
[4] Cyan Large Rounded 0
. . .

Item to classify:
Cyan Small Twisted

Joint counts:
  3   5
 18  15
  5   3

Class counts:
24 16

Evidence terms:
0.0082 0.0131

Probabilities:
0.3855 0.6145

Predicted class:
1

End naive Bayes classification
```

Figure 1 **Simplified Naive Bayes Classification Demo Run**

In my opinion, naive Bayes classification is best explained using a concrete example. The first step is to scan through the source data and compute joint counts. If there are nx predictor variables (three in the demo) and nc classes (two in the demo), then there are nx * nc joint counts to compute. Notice that the counting process means that predictor data must be discrete rather than numeric.

After calculating joint counts, 1 is added to each count. This is called Laplacian smoothing and is done to prevent any joint count from being 0, which would zero out the final results. For the demo data the smoothed joint counts are:

```
cyan and 0:     2 + 1 = 3
cyan and 1:     4 + 1 = 5
small and 0:   17 + 1 = 18
small and 1:   14 + 1 = 15
twisted and 0:  4 + 1 = 5
twisted and 1:  2 + 1 = 3
```

$$P(C_k|\boldsymbol{X}) = \frac{P(C_k) * P(\boldsymbol{X}|C_k)}{P(\boldsymbol{X})}$$

$$P(C_k|\boldsymbol{X}) = \frac{P(C_k) * P(x_1, \ldots, x_n|C_k)}{P(x_1, \ldots, x_n)}$$

$$P(C_k|\boldsymbol{X}) = \frac{1}{Z} * P(C_k) * \prod_{i=1}^{n} P(x_i|C_k)$$

$$where\ Z = \sum_{k} P(C_k)P(\boldsymbol{X} \mid C_k)$$

Figure 2 **Three Forms of Naive Bayes Classification Math**

Naive Bayes classification is both simple and complicated. Implementation is relatively simple, but the underlying math ideas are very complex.

Next, the raw counts of class 0 items and class 1 items are calculated. Because these counts will always be greater than zero, no smoothing factor is needed. For the demo data, the class counts are:

```
0:  24
1:  16
```

Next, an evidence term for each class is calculated. For class 0, the evidence term is:

```
Z(0) = (3 / 24+3) * (18 / 24+3) * (5 / 24+3) * (24 / 40)
       = 3/27 * 18/27 * 5/27 * 24/40
       = 0.1111 * 0.6667 * 0.1852 * 0.6
       = 0.0082
```

The first three terms of calculation for Z(0) use the smoothed joint counts for class 0, divided by the class count for 0 (24) plus the number of predictor variables (nx = 3) to compensate for the three additions of 1 due to the Laplacian smoothing. The fourth term is P(class 0). The calculation of the class 1 evidence term follows the same pattern:

```
Z(1) = (5 / 16+3) * (15 / 16+3) * (3 / 16+3) * (16 / 40)
       = 5/19 * 15/19 * 3/19 * 16/40
       = 0.2632 * 0.7895 * 0.1579 * 0.4
       = 0.0131
```

The last step is to compute pseudo-probabilities:

```
P(class 0) = Z(0) / (Z(0) + Z(1))
           = 0.0082 / (0.0082 + 0.0131)
           = 0.3855

P(class 1) = Z(1) / (Z(0) + Z(1))
           = 0.0131 / (0.0082 + 0.0131)
           = 0.6145
```

The denominator sum is called the evidence and is used to normalize the evidence terms so that they sum to 1.0 and can be loosely interpreted as probabilities. Note that if you're just interested in prediction, you can simply use the largest evidence term and skip the evidence normalization step.

The Demo Program

The complete demo program, with a few minor edits to save space, is presented in **Figure 3**. To create the program, I launched Visual Studio and created a new console application named NaiveBayes. I used Visual Studio 2017, but the demo has no significant .NET Framework dependencies so any version of Visual Studio will work fine.

After the template code loaded, in the editor window I removed all unneeded namespace references and added a reference to the System.IO namespace. In the Solution Explorer window, I right-clicked on file Program.cs, renamed it to the more descriptive NaiveBayesProgram.cs, and allowed Visual Studio to automatically rename class Program.

After building the project, I used Notepad to create the 40-item dummy data file with the contents shown in **Figure 4**, and saved it as BayesData.txt in the project root directory.

Loading Data into Memory

The demo uses a program-defined method named LoadData to read the data file into memory as an array-of-arrays style matrix of type string. Method LoadData assumes the class values are the last value on each line of data. An alternative design is to read the predictor values into a string matrix and read the 0-1 class values into an array of type integer.

Note that if you're just interested in prediction, you can simply use the largest evidence term and skip the evidence normalization step.

Method LoadData calls a helper method named MatrixString, which creates an array-of-arrays string matrix with the specified number of rows (40) and columns (4). An alternative design is to programmatically compute the number of rows and columns, but in my opinion the hardcoded approach is simpler and better.

Program Logic

All of the program control logic is contained in the Main method. The joint counts are stored in an array-of-arrays integer matrix

Figure 3 **The Naive Bayes Classification Demo Program**

```
using System;
using System.IO;
namespace CSharp
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("\nBegin simple naive Bayes demo\n");
      string fn = "C:\\NaiveBayes\\data.txt";

      int nx = 3;  // Number predictor variables
      int nc = 2;  // Number classes
      int N = 40;  // Number data items

      string[][] data = LoadData(fn, N, nx+1, ',');
      Console.WriteLine("Training data:");
      for (int i = 0; i < 5; ++i) {
        Console.Write("[" + i + "] ");
        for (int j = 0; j < nx+1; ++j) {
          Console.Write(data[i][j] + " ");
        }
        Console.WriteLine("");
      }
      Console.WriteLine(". . . \n");

      int[][] jointCts = MatrixInt(nx, nc);
      int[] yCts = new int[nc];

      string[] X = new string[] { "Cyan", "Small", "Twisted" };
      Console.WriteLine("Item to classify: ");
      for (int i = 0; i < nx; ++i)
        Console.Write(X[i] + " ");
      Console.WriteLine("\n");

      // Compute joint counts and y counts
      for (int i = 0; i < N; ++i) {
        int y = int.Parse(data[i][nx]);
        ++yCts[y];
        for (int j = 0; j < nx; ++j) {
          if (data[i][j] == X[j])
            ++jointCts[j][y];
        }
      }

      // Laplacian smoothing
      for (int i = 0; i < nx; ++i)
        for (int j = 0; j < nc; ++j)
          ++jointCts[i][j];

      Console.WriteLine("Joint counts: ");
      for (int i = 0; i < nx; ++i) {
        for (int j = 0; j < nc; ++j) {
          Console.Write(jointCts[i][j] + " ");
        }
        Console.WriteLine("");
      }

      Console.WriteLine("\nClass counts: ");
      for (int k = 0; k < nc; ++k)
        Console.Write(yCts[k] + " ");
      Console.WriteLine("\n");

      // Compute evidence terms
      double[] eTerms = new double[nc];
      for (int k = 0; k < nc; ++k) {
        double v = 1.0;
        for (int j = 0; j < nx; ++j) {
          v *= (double)(jointCts[j][k]) / (yCts[k] + nx);
        }
        v *= (double)(yCts[k]) / N;
        eTerms[k] = v;
      }

      Console.WriteLine("Evidence terms:");
      for (int k = 0; k < nc; ++k)
        Console.Write(eTerms[k].ToString("F4") + " ");
      Console.WriteLine("\n");

      double evidence = 0.0;
      for (int k = 0; k < nc; ++k)
        evidence += eTerms[k];

      double[] probs = new double[nc];
      for (int k = 0; k < nc; ++k)
         probs[k] = eTerms[k] / evidence;

      Console.WriteLine("Probabilities: ");
      for (int k = 0; k < nc; ++k)
        Console.Write(probs[k].ToString("F4") + " ");
      Console.WriteLine("\n");

      int pc = ArgMax(probs);
      Console.WriteLine("Predicted class: ");
      Console.WriteLine(pc);

      Console.WriteLine("\nEnd naive Bayes ");
      Console.ReadLine();
    } // Main

    static string[][] MatrixString(int rows, int cols)
    {
      string[][] result = new string[rows][];
      for (int i = 0; i < rows; ++i)
        result[i] = new string[cols];
      return result;
    }

    static int[][] MatrixInt(int rows, int cols)
    {
      int[][] result = new int[rows][];
      for (int i = 0; i < rows; ++i)
        result[i] = new int[cols];
      return result;
    }

    static string[][] LoadData(string fn, int rows,
      int cols, char delimit)
    {
      string[][] result = MatrixString(rows, cols);
      FileStream ifs = new FileStream(fn, FileMode.Open);
      StreamReader sr = new StreamReader(ifs);
      string[] tokens = null;
      string line = null;
      int i = 0;
      while ((line = sr.ReadLine()) != null)  {
        tokens = line.Split(delimit);
        for (int j = 0; j < cols; ++j)
          result[i][j] = tokens[j];
        ++i;
      }
      sr.Close(); ifs.Close();
      return result;
    }

    static int ArgMax(double[] vector)
    {
      int result = 0;
      double maxV = vector[0];
      for (int i = 0; i < vector.Length; ++i) {
        if (vector[i] > maxV) {
          maxV = vector[i];
          result = i;
        }
      }
      return result;
    }
  } // Program
} // ns
```

named jointCts, which is created using a helper method called MatrixInt. The row indices of jointCts point to the predictor values (Cyan, Small, Twisted) and the column indices refer to the class value. For example, jointCts[2][1] holds the count of data items that have both Twisted and class 1. An integer array with length 2 named yCts holds the number of data items with class 0 and class 1.

The code that uses the joint counts and class counts to compute the two evidence terms is:

```
double[] eTerms = new double[nc];
for (int k = 0; k < nc; ++k) {
  double v = 1.0;
  for (int j = 0; j < nx; ++j) {
    v *= (double)(jointCts[j][k]) / (yCts[k] + nx);
  }
  v *= (double)(yCts[k]) / N;
  eTerms[k] = v;
}
```

I call these values pseudo-probabilities because, although they sum to 1.0, they don't really represent the likelihood of a result from many repeated sampling experiments.

Notice that v is the product of nx+1 fractions that are all less than 1.0. This isn't a problem for the demo program because there are only nx = 3 predictor values and none of the fraction terms can ever be less than 1/27 = 0.0370. But in some situations, you could run into arithmetic trouble by multiplying many very small values.

A technique for avoiding arithmetic errors when computing the evidence terms is to use the log trick that takes advantage of the facts that log(A * B) = log(A) + log(B) and log(A / B) = log(A) - log(B). By computing the log of each evidence term, and then taking the exp function of that result, you can use addition and subtraction of many small values instead of multiplication and division. One possible refactoring using the log trick is:

```
double[] eTerms = new double[nc];
for (int k = 0; k < nc; ++k) {
  double v = 0.0;
  for (int j = 0; j < nx; ++j) {
    v += Math.Log(jointCts[j][k]) - Math.Log(yCts[k] + nx);
  }
  v += Math.Log(yCts[k]) - Math.Log(N);
  eTerms[k] = Math.Exp(v);
}
```

Generating the Predicted Class

After the evidence terms have been computed, the demo program sums them and uses them to compute pseudo-probabilities. I call these values pseudo-probabilities because, although they sum to 1.0, they don't really represent the likelihood of a result from many repeated sampling experiments. However, you can cautiously interpret pseudo-probabilities as mild forms of confidence. For example, pseudo-probabilities of (0.97, 0.03) suggest class 0 with a bit more strength than (0.56, 0.44).

The predicted class is generated by calling the program-defined ArgMax function, which returns the index of the largest value in a numeric array. For example, if an array holds values (0.20, 0.50, 0.90, 0.10) then ArgMax returns 2.

Wrapping Up

The demo program presented in this article performs binary classification because there are only two class values. The program logic can also be used without modification for multiclass classification. The predictor values in the demo program are all categorical. If your data has numeric data, such as a weight variable with values like 3.78 and 9.21, you can apply the technique presented in this article by binning the numeric data into categories such as light, medium and heavy.

Figure 4 **File BayesData.txt Contents**

```
Aqua,Small,Twisted,1
Blue,Small,Pointed,0
Dune,Large,Rounded,0
Dune,Small,Rounded,1
Cyan,Large,Rounded,0
Aqua,Small,Rounded,1
Aqua,Small,Rounded,0
Cyan,Small,Pointed,1
Cyan,Small,Pointed,1
Dune,Small,Rounded,1
Dune,Small,Rounded,0
Dune,Small,Rounded,1
Dune,Small,Rounded,1
Cyan,Small,Pointed,1
Dune,Small,Rounded,1
Dune,Large,Rounded,0
Cyan,Small,Twisted,1
Blue,Small,Rounded,0
Aqua,Small,Pointed,1
Aqua,Small,Pointed,1
Dune,Small,Twisted,0
Blue,Small,Rounded,0
Dune,Small,Rounded,0
Blue,Small,Twisted,0
Dune,Small,Rounded,0
Aqua,Large,Pointed,1
Dune,Large,Rounded,0
Dune,Small,Rounded,0
Dune,Small,Rounded,0
Cyan,Large,Rounded,0
Dune,Small,Twisted,0
Dune,Large,Twisted,0
Dune,Small,Rounded,0
Dune,Small,Rounded,0
Dune,Large,Rounded,0
Aqua,Large,Rounded,1
Aqua,Small,Rounded,0
Aqua,Small,Rounded,1
Dune,Small,Rounded,0
Blue,Small,Rounded,0
```

The program logic can also be used without modification for multiclass classification.

There are several other forms of naive Bayes classification in addition to the type presented in this article. One form uses the same underlying math principles as those used by the demo program, but can handle data where all the predictor values are numeric. However, you must make assumptions about the math properties of the data, such as that the data is a normal (Gaussian) distribution with a certain mean and standard deviation. ■

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

Thanks to the following Microsoft technical experts for reviewing this article: *Chris Lee, Ricky Loynd, Kirk Olynyk*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

Sept 29 - Oct 3, 2019 | Westin Gaslamp Quarter

EXPERIENCE TECH TRAINING IN THE WARM CALIFORNIA SUN

INTENSE DEVELOPER TRAINING CONFERENCE

In-depth Technical Content On:

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

New This Year!

On-Demand Session Recordings Now Available

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

Save $400! When You Register by July 26

Your Adventure Starts Here!

vslive.com/sandiego

SUPPORTED BY

Microsoft | Visual Studio | msdn magazine | Visual Studio MAGAZINE

PRODUCED BY

CONVERGE360 An 1105 MEDIA company

AGENDA AT-A-GLANCE

#VSLive

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

START TIME	END TIME				
		Pre-Conference Full Day Hands-On Labs: Sunday, September 29, 2019 *(Separate entry fee required)*			
7:30 AM	9:00 AM	Pre-Conference Hands-On Lab Registration - Coffee and Morning Pastries			
9:00 AM	6:00 PM	**HOL01** Full Day Hands-On Lab: Develop an ASP.NET Core 2 App with EF Core 2 in a Day - *Philip Japikse*		**HOL02** Day Hands-On Lab: Building a Modern DevOps Pipeline on Microsoft Azure with ASP.NET Core and Azure DevOps - *Brian Randell & Mickey Gousset*	
START TIME	**END TIME**	**Pre-Conference Workshops: Monday, September 30, 2019** *(Separate entry fee required)*			
7:30 AM	8:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries			
8:00 AM	5:00 PM	**M01** Workshop: Modern Security Architecture for ASP.NET Core - *Brock Allen*	**M02** Workshop: Kubernetes on Azure - *Vishwas Lele*	**M03** Workshop: Cross-Platform C# Using .NET Core, Kubernetes, and WebAssembly - *Rockford Lhotka & Jason Bock*	
6:45 PM	9:00 PM	Dine-A-Round			
START TIME	**END TIME**	**Day 1: Tuesday, October 1, 2019**			
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**T01** MVVM and ASP.NET Core Razor Pages - *Ben Hoelting*	**T02** Azure, Windows and Xamarin: Using the Cloud to Power Your Cross-platform Applications - *Laurent Bugnion*	**T03** .NET Core and Azure, Microservices to Messaging - *Brady Gaster*	**T04** What's New in C# 7.X And C# 8 - *Philip Japikse*
9:30 AM	10:45 AM	**T05** Angular 101 - *Deborah Kurata*	**T06** Securing Web APIs from Mobile and Native Applications - *Brock Allen*	**T07** SQL Server 2019 Deep Dive - *Scott Klein*	**T08** How Microsoft Does DevOps - *Mickey Gousset*
11:00 AM	12:00 PM	**Keynote:** AI for the Rest of Us - *Damian Brady, Cloud Developer Advocate, Microsoft*			
12:00 PM	1:00 PM	Lunch			
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors			
1:30 PM	2:45 PM	**T09** Securing Single Page Applications (SPAs) - *Ben Hoelting*	**T10** To Be Announced	**T11** Data Pipelines with End-2-End Azure Analytics - *Scott Klein*	**T12** .NET Standard, .NET Core, why and how? - *Laurent Bugnion*
3:00 PM	4:15 PM	**T13** Managing Your Angular Async Data Effectively - *Deborah Kurata*	**T14** Cross-Platform Development with Xamarin, C#, and CSLA .NET - *Rockford Lhotka*	**T15** What Every Developer Ought to Know About #deeplearning and #neuralnetwork - *Vishwas Lele*	**T16** Azure Pipelines - *Brian Randell*
4:15 PM	5:30 PM	Welcome Reception			
START TIME	**END TIME**	**Day 2: Wednesday, October 2, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**W01** Diving Deep Into ASP.NET Core 2.x - *Philip Japikse*	**W02** Building End-to-End ML Solution Using Azure Machine Learning Service - *Raj Krishnan*	**W03** Go Serverless with Azure Functions - *Eric Boyd*	**W04** Exceptional Development: Dealing With Exceptions in .NET - *Jason Bock*
9:30 AM	10:45 AM	**W05** WebAssembly: the Browser is your OS - *Jeremy Likness*	**W06** Azure Data Explorer—An in-depth Look at the New Microsoft PaaS Offering - *Raj Krishnan*	**W07** Keep Secrets with Azure Key Vault - *Eric D. Boyd*	**W08** Testability in .NET - *Jason Bock*
11:00 AM	12:00 PM	**General Session:** Moving .NET Beyond Windows = James Montemagno, Principal Program Manager - *Mobile Developer Tools, Microsoft*			
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch			
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:15pm (Must be present to win)			
1:30 PM	1:50 PM	**W09** Fast Focus: Getting Started with ASP.NET Core 2.0 Razor Pages - *Walt Ritscher*	**W10** Fast Focus: The Four Flavors of Power BI - *Thomas LeBlanc*	**W11** Fast Focus: Serverless of Azure 101 - *Eric D. Boyd*	
2:00 PM	2:20 PM	**W12** Fast Focus: Sharing C# Code Across Platforms - *Rockford Lhotka*	**W13** Fast Focus: Running .NET on Linux: What's Different? - *Steve Roberts*	**W14** Fast Focus: Lessons about DevOps from 3D Printing - *Colin Dembovsky*	
2:30 PM	3:45 PM	**W15** Angular vs React - a Comparison - *Gregor Dzierzon*	**W16** Effective Visualizations with Power BI - *Thomas LeBlanc*	**W17** Serverless .NET on AWS - *Steve Roberts*	**W18** Better Azure DevOps - Security 101 - *Brian Randell*
4:00 PM	5:15 PM	**W19** Object Oriented Programming Using TypeScript - *Gregor Dzierzon*	**W20** Transition from SSIS to Azure Data Factory - *Thomas LeBlanc*	**W21** Cloud Debugging – A Revolutionary Approach - *Alon Fliess*	**W22** DevOps for Machine Learning - *Damian Brady*
6:45 PM	9:00 PM	VSLive! Event			
START TIME	**END TIME**	**Day 3: Thursday, October 3, 2019**			
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries			
8:00 AM	9:15 AM	**TH01** Advanced Serverless Workflows with Durable Functions - *Jeremy Likness*	**TH02** Stunning Mobile Apps with the Xamarin Visual Design System - *James Montemagno*	**TH03** Building a Stronger Team, One Strength at a Time - *Angela Dugan*	**TH04** Creating Business Applications Using Blazor (Razor Components) - *Michael Washington*
9:30 AM	10:45 AM	**TH05** What's New in Bootstrap 4 - *Paul Sheriff*	**TH06** How to Have Better Business Intelligence through Visualizations - *Walt Ritscher*	**TH07** How Do You Measure Up? Collect the Right Metrics for the Right Reasons - *Angela Dugan*	**TH08** Building Business Applications Using Bots - *Michael Washington*
11:00 AM	12:15 PM	**TH09** Signal R: Real-time for All Things - *Brady Gaster*	**TH10** Microsoft Power Platform, RAD Done Right: Building Dynamic Mobile Apps with PowerApps - *Walt Ritscher*	**TH11** Past, Present & Future of C# Debugging - *Alon Fliess*	**TH12** Automate Your Life with PowerShell on Lambda - *Steve Roberts*
12:15 PM	1:30 PM	Lunch			
1:30 PM	2:45 PM	**TH13** Advanced Fiddler Techniques - *Robert Boedigheimer*	**TH14** Building UWP Apps for Multiple Devices - *Tony Champion*	**TH15** To Microservice or Not to Microservice? How? - *Alon Fliess*	**TH16** Testing in Production Using Azure and Visual Studio Team Services (VSTS) - *Colin Dembovsky*
3:00 PM	4:15 PM	**TH17** Improving Web Performance - *Robert Boedigheimer*	**TH18** Building Cross Device Experiences with Project Rome - *Tony Champion*	**TH19** Getting Started with Unit Testing in Visual Studio - *Paul Sheriff*	**TH20** Modernizing Your Source Control: Migrating to Git from Team Foundation Version Control (TFVC) - *Colin Dembovsky*

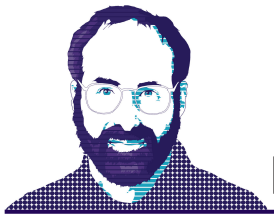
Speakers and sessions subject to change

CONNECT WITH US

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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



DAVID PLATT

Testing 1 ... 2 ... 3 ...

The single best way to improve your UX designs is to test them on live users before coding them. Strangely, I find almost nobody does this today. I think it's because of the name, "UX Testing." The "T" word implies something that you do at the end of the process, to ensure that you've satisfied the requirements. But UX testing needs to happen before coding, to generate the correct requirements for your coders.

I can't pull a design out of my head fully formed, like Athena from the brow of Zeus. Neither can you. What we can do is generate mock designs to show our test users, incorporate their feedback (I like this, I hate that, and oh, wait a minute, I forgot to mention such-and-such) into improved mockups; iterating until they're good enough.

Here's an example from my book, "The Joy of UX" (Addison-Wesley Professional, June 2016). It's a mobile app for commuter rail riders near Boston. Instead of overwhelming you with a full timetable, the app shows only the times for trains going to and from the station you specify.

On Monday, I generated mockups with an editor called Balsamiq. You can see three iterations of the app UX in the panels in **Figure 1**. The existing app had required lots of jumping around to find the train time you wanted, so I created a design with no jumping at all. I put everything the user needs on one page: inbound trains, outbound trains, next train arrival time, alerts, passes, the works. You can see this in the left-most pane of **Figure 1**.

I emailed my Harvard students, asking who rode commuter rail, and snagging three riders as testing volunteers. On Tuesday, I showed them the mockup via Skype, asking, "You want to find the train to take tomorrow. What do you do?"

They unanimously hated this layout. It was complex, cramped, hard to pick out what they wanted. I omitted the arrival times, figuring they wouldn't care. They did.

I couldn't have been happier. The mockup stimulated the users' thoughts, encouraging them to speak out. I wouldn't have gotten this feedback otherwise.

OK, back to Balsamiq. By Wednesday afternoon, I had the design shown in the middle panel of **Figure 1**. I used a tab control to separate the pass from the schedule, and expanded the schedule information on its own tab. I showed this new mockup to the same users.

They liked it somewhat better. The app now displayed the schedule grids as they appeared on the station monitors, so it felt familiar. They now told me that in the morning, they only care about inbound trains, and in the afternoon, only about outbound trains. Displaying both on the same tab was more distracting than helpful.

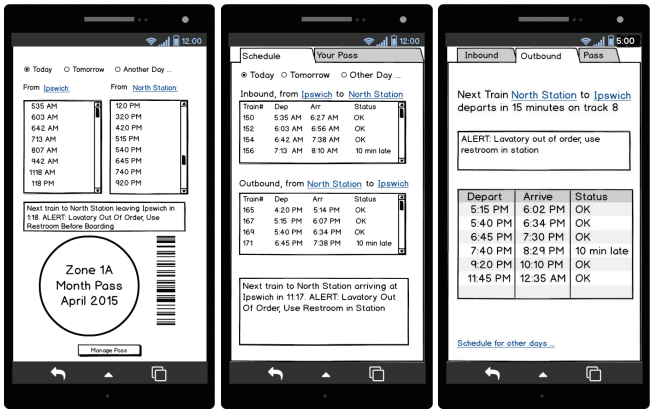


Figure 1 **Commuter Rail App Iterations**

Additionally, these discussions triggered a key breakthrough. One user looked at the next-train countdown timer, now somewhat easier to see but still not great. She said, "That next train timer—I need it up top, easy to see quickly, so I know if I need to run."

This is what I call a double-smack. You smack your forehead, saying, "I never imagined [whatever]." And then within 30 seconds, you smack your forehead again, saying, "What could be more obvious than [whatever]?"

Which led me to the right-most panel in **Figure 1**. Inbound and outbound trains each occupy their own tab, with plenty of space for easy reading. The countdown timer is at the top, the alerts visible in the middle. They liked this one a lot. It's the one I coded.

It pays to do this exploration at the initial mockup stage. Coding each mockup would cost much more time and money. When I got the users' feedback, I'd have to either a) throw my investment away because they didn't like it, or b) keep something users hated because I wouldn't throw away my investment. Neither choice leads to great apps.

So, why don't more companies do this? I can only guess that it's the "T" word. Consider the usual development process: You write the code, you ship the code, then you test the code. (Customers are a big help here.) What should I call this process to indicate its vital position in the earliest stage of the design, rather than at the end? Validation? No. Iteration? I don't know. Use the comment board to tell me: What word(s) would communicate this vital necessity? ■

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.*

Address the ELEPHANT IN THE ROOM

Bad address data costs you money, customers and insight.

Melissa's 30+ years of domain experience in address management, patented fuzzy matching and multi-sourced reference datasets power the global data quality tools you need to keep addresses clean, correct and current. The result? Trusted information that improves customer communication, fraud prevention, predictive analytics, and the bottom line.

- Global Address Verification
- Digital Identity Verification
- Email & Phone Verification
- Location Intelligence
- Single Customer View

+ .NET
+ Microsoft® SSIS
+ Microsoft® Dynamics CRM

See the Elephant in Your Business -

Name it and Tame it!

melissa

www.Melissa.com | 1-800-MELISSA

Free API Trials, Data Quality Audit & Professional Services.