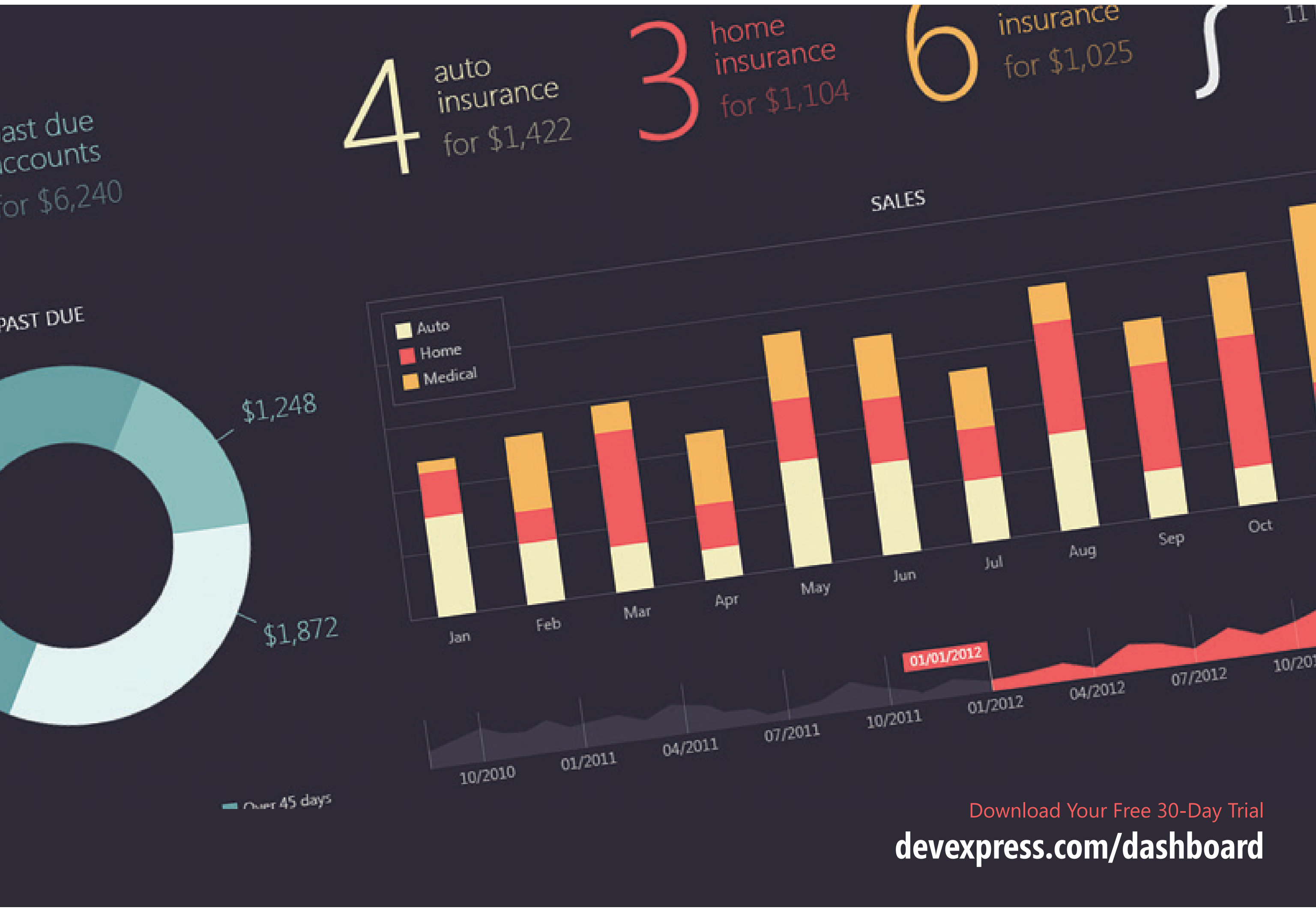
THE MICROSOFT JOURNAL FOR DEVELOPERS

FEBRUARY 2016 VOL 31 NO 2

msdn magazine

Your Next Great Dashboard Starts Here

Create high impact and information rich decision support systems for desktops and the web with the DevExpress Universal Subscription.



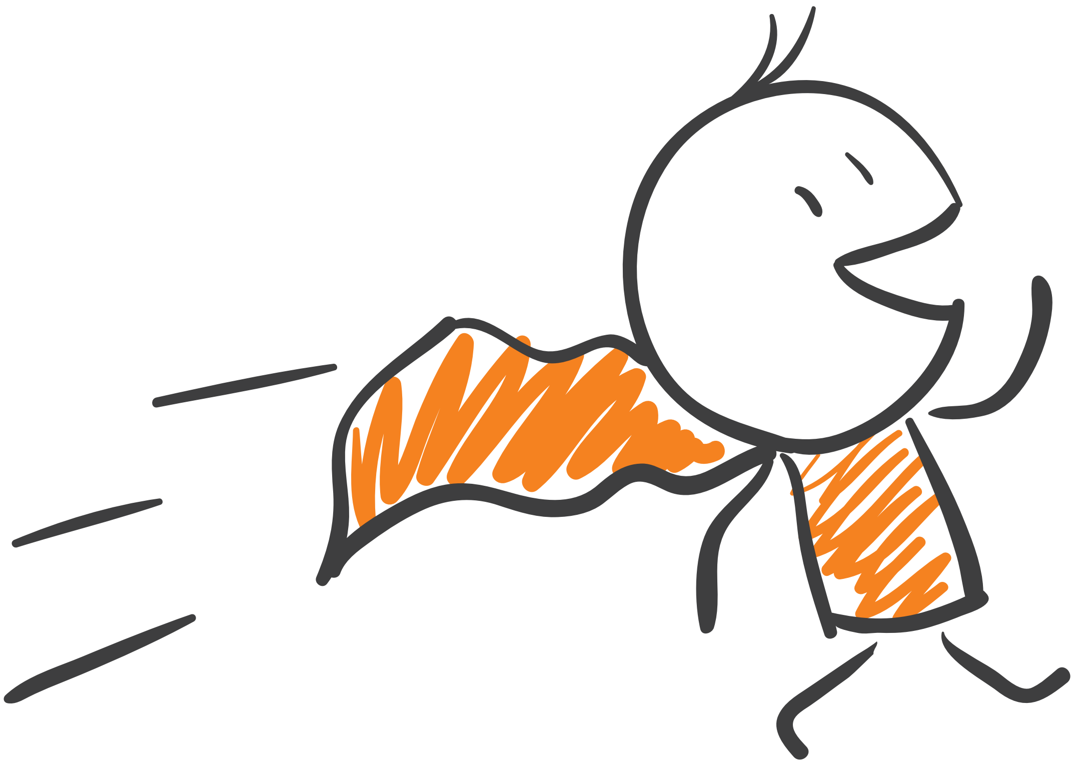
Download Your Free 30-Day Trial

devexpress.com/dashboard

DevExpress®

Unleash the UI Superhero in You

With DevExpress tools, you'll deliver amazing user-experiences for Windows®, the Web and Your Mobile World



Experience the DevExpress difference today.
Download your free 30-day trial.
devexpress.com/try

DevExpress® WIN ASP MVC WPF

THE MICROSOFT JOURNAL FOR DEVELOPERS

FEBRUARY 2016 VOL 31 NO 2

msdn magazine

UWP Apps for Web Devs...18

Microsoft

TEXT CONTROL

ASP.NET MVC REPORTING

The first cross-browser, true WYSIWYG HTML5-based document and template editor.

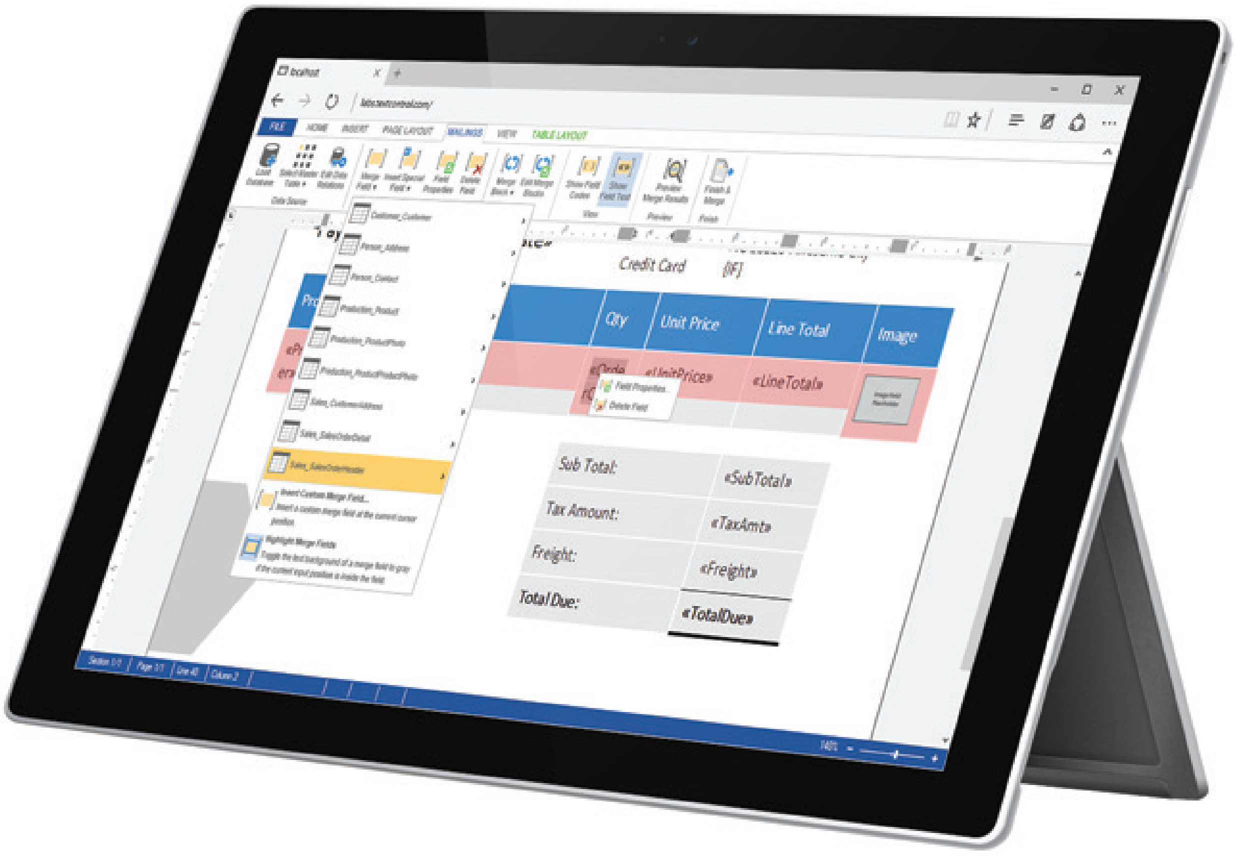
Now with full ASP.NET MVC support.

Give your users an MS Word compatible editor to create powerful documents and reporting templates anywhere - in any browser. Feature-complete including mail merge, sections, headers and footers, drawings and shapes, tables, barcodes and charts.

Available for ASP.NET Web Forms and MVC.

Visual Studio Partner

HTML



Edit and create MS Word documents

Create and modify Adobe® PDF documents

Create reports and mail merge templates

Integrate with Microsoft® Visual Studio

```
PM> Install-Package TXTextControl.Web
```

Live demo and 30-day trial version download at:

www.textcontrol.com/html5

X13 released

TEXT CONTROL

msdn magazine

FEBRUARY 2016 VOLUME 31 NUMBER 2

ID STATEMENT MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. $35.00, International $60.00. Annual digital subscription rates payable in U.S. funds are: U.S. $25.00, International $25.00. Single copies/back issues: U.S. $10, all others $12. Send orders with payment to: MSDN Magazine, P.O. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: P.O. Box 201, Richmond Hill, ON L4B 4R5, Canada.

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

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

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

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; E-mail: jlong@meritdirect.com; Web: www.meritdirect.com/1105

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

Microsoft

BPA WORLDWIDE

1906 100 2006 AMERICAN BUSINESS MEDIA A Century Moving Business Forward

WPA Western Publications Association

LEADTOOLS®
THE WORLD LEADER IN IMAGING SDKs

DESKTOP TABLET MOBILE

OCR FOR ANY CLIENT, DEVICE OR SERVER. DONE.

LEADTOOLS OCR SDK

Fast, accurate and reliable optical character recognition for use in any application or environment

Utilize multiple cores for unparalleled performance

Supports multiple text recognition engines including OCR, ICR, MICR, MRZ and MRP

Automatically detect, segment and recognize multiple languages on the same document

.NET Windows API WinRT Linux iOS OS X Android HTML5

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

MEAN Streak

If you've been reading *MSDN Magazine*, you know Ted Neward. He's been a regular in our pages going back to 2008, writing The Polyglot Programmer and then The Working Programmer columns. Over the years, he's covered everything from SQL and NoSQL databases and C# tips and tricks, to the dangers of rigid programming methodologies. Through it all, Neward has applied his trademark wit and irreverence.

For the past several months, Neward has been writing about MongoDB, Express, AngularJS and Node.js—collectively known as the MEAN stack. The Node.js platform and the MEAN stack built around it have quickly earned a strong following among JavaScript developers, and with Microsoft supporting both Node.js and MongoDB on Microsoft Azure, it has emerged as a solid option for Visual Studio developers, as well. You can read this month's column on using the MongoDB database at msdn.com/magazine/mt632276, and be sure to check out the opening column in the series at msdn.com/magazine/mt185576.

I caught up with Neward and asked him about his ongoing MEAN development series, and what motivated him to cover the topic in depth. Of course, the growing popularity of MEAN development was a factor, but he says he was drawn to the stack because Node developers tend to approach problems in ways that are "fundamentally different" from seasoned .NET programmers.

"MEAN has some interesting ideas to it, and I think it's a useful tool to have in your tool belt, but I think the more important thing to realize is that MEAN is just an approach and a pre-defined set of architectural 'pieces,'" Neward says. "It's just as reasonable to have a Mongo + Web API + AngularJS + C# stack, or a DocumentDB + Web API + AngularJS + C# stack."

In writing about the MEAN stack, Neward says he's learned plenty, including a bit about npm and package.json, about the workings of the require method, and about the behavior of callbacks. (He warns that callbacks do not execute serially, despite how things appear when working in the editor.) So what advice does Neward have for developers considering a move to MEAN? In a phrase, look out for gotchas.

"The biggest advice I have for developers is to make sure they're comfortable with JavaScript and the callback-based nature of Node.js. There's a lot of subtleties involved when the flow of execution doesn't quite behave the way the code on the page implies it will," he says, adding that even with JavaScript fixes being made, "certain code idioms will linger for years to come."

Readers can look forward to explorations into the Angular framework, replacing MongoDB in the stack with DocumentDB, and using Edge.js to access .NET-specific resources from Node.js.

Neward isn't done with MEAN yet. Readers can look forward to explorations into the Angular framework, replacing MongoDB in the stack with DocumentDB, and using Edge.js to access .NET-specific resources from Node.js. As Neward notes, "There's a lot left to explore here."

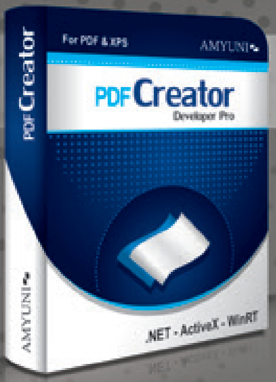
Are there specific aspects of the MEAN stack you would like to see Neward explore in his series? E-mail him at ted@tedneward.com.

[illegible]

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

Switch to Amyuni PDF

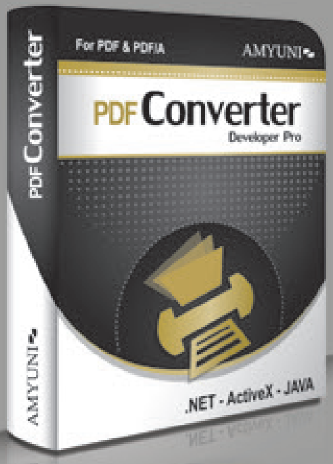
AMYUNI



NEW v5.5

Create and Edit PDFs in .NET, COM/ActiveX, WinRT & UWP

• Edit, process and print PDF 1.7 documents
• Create, fill-out and annotate PDF forms
• Fast and lightweight 32- and 64-bit components for .NET and ActiveX/COM
• New Universal Apps DLLs enable publishing C#, C++, CX or Javascript apps to windows Store
• Updated Postscript/EPS to PDF conversion module



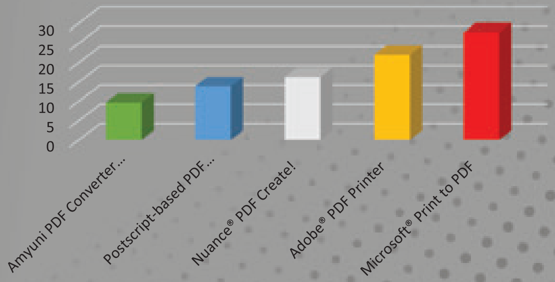
High Performance PDF Printer for Desktops and Servers

• Print to PDF in a fraction of the time needed with other tools. WHQL tested for all Windows platforms. Version 5.5 updated for Windows 10 support

Complete Suite of Accurate PDF Components

• All your PDF processing, conversion and editing in a single package
• Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
• Includes our Microsoft WHQL certified PDF Converter printer driver
• Export PDF documents into other formats such as Jpeg, PNG, XAML or HTML5
• Import and Export XPS files using any programming environment

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



Other Developer Components from Amyuni®

• WebkitPDF: Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver
• PDF2HTML5: Conversion of PDF to HTML5 including dynamic forms
• Postscript to PDF Library: For document workflow applications that require processing of Postscript documents
• OCR Module: Free add-on to PDF Creator uses the Tesseract engine for character recognition
• Javascript engine: Integrate a full Javascript interpreter into your applications to process PDF files or for any other need

CERTIFIED FOR Windows Server® 2008

Windows Server 2012 Certified

Windows® 7

Windows

All development tools available at
www.amyuni.com

AMYUNI Technologies

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

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

Loyalty Test

While at my first job, I underwent a life-changing surgery. It was paid for by my employer's excellent health benefits, so I paid nothing out of pocket.

I felt so grateful that I believed I was "meant to be" in that job. I repaid my firm's kindness by staying on for years. I considered leaving tantamount to betrayal. Over time, I became unproductive, hardly performed at my best, and began to resent my job. I felt handcuffed by my own loyalty.

Looking back, the best way I could have been loyal was to stay there as long as I performed at my peak—and then left with heartfelt gratitude, wishing my company and coworkers the best when I couldn't be at *my* best any longer.

Anytime I've stayed on at a job out of obligation, I've failed to live up to my own potential. Worse, I've actually betrayed the very managers to whom I've tried to be loyal, by growing to dislike my job.

Settling or Choosing?

Imagine I'm not *really* interested in someone I'm dating, but I "settle" for her and end up marrying her because I don't think I have a shot with anyone else. By doing this, I cheat her out of an amazing relationship with someone who in the future will love her fully. I'm cheating myself, too, because I don't think I'm worthy of such full love myself.

Settling for an employer is no different. To let go of a bad employer or relationship is as much an act of love and loyalty (both to myself and the other party) as it is to stay.

In my career now, I've cultivated my employability, partly by mastering the art of job hunting and the science of careerism. I can walk away and get another job, but I choose to stay because I love it. Now *that* is real loyalty.

The Antidote: Letting Go with Gratitude

Today, I give thanks for the rewards, friendships, health and growth my company has given me, and let it go for having been in my life.

Today, I understand that I'm not "meant to be" in any job. Even if I'm awarded a million dollars at work or have my life saved by my coworkers, it doesn't mean that I'm meant to be there. Where I am right now is, by definition, exactly where I am meant to be. If I'm not happy with it, I can change it. If I'm already happy, I can keep going. My destiny is the life I'm living right now, not the life I want to live in the future.

No matter what career mistakes I've made, that's *exactly* what I'm meant to do. The way to ensure that I don't repeat mistakes is to learn what I can and forgive that which is outside my control. When I thank these mistakes for being a part of my life and let them go, I unlock my own power and reach greater heights.

The Darker Side of Loyalty

At one point, I went through a string of bad jobs, characterized by long hours and harsh coworkers. Eventually, I found my way to an excellent company with perks, colleagues and a work-life balance to match.

And, yet, I found myself repeatedly frustrated at this new dream job. I'd come to believe that real work had to be very challenging, and that my life had to be difficult to have meaning. When I didn't feel external pressure to perform, I felt I'd sold myself short to a starry-eyed employer that didn't understand competitiveness.

In my career now, I've cultivated my employability, partly by mastering the art of job hunting and the science of careerism. I can walk away and get another job, but I *choose* to stay because I love it.

In retrospect, my loyalty to my own conception of an ideal employer had become self-destructive. I was thriving on the excitement and aggression of intense work, at the expense of self-respect and fulfillment. In lusting after grandeur and belligerence, control and power, my hankering would unfailingly turn to heartbreak. Finally wising up, I began to go for a slower burn, experimenting with companies who loved me and weren't afraid to show it calmly.

I did this by playing with my own assumptions. If I felt judgmental toward a company during my job search (for example, too trusting, too laid-back, not intelligent), I'd go against my instincts and give them a shot.

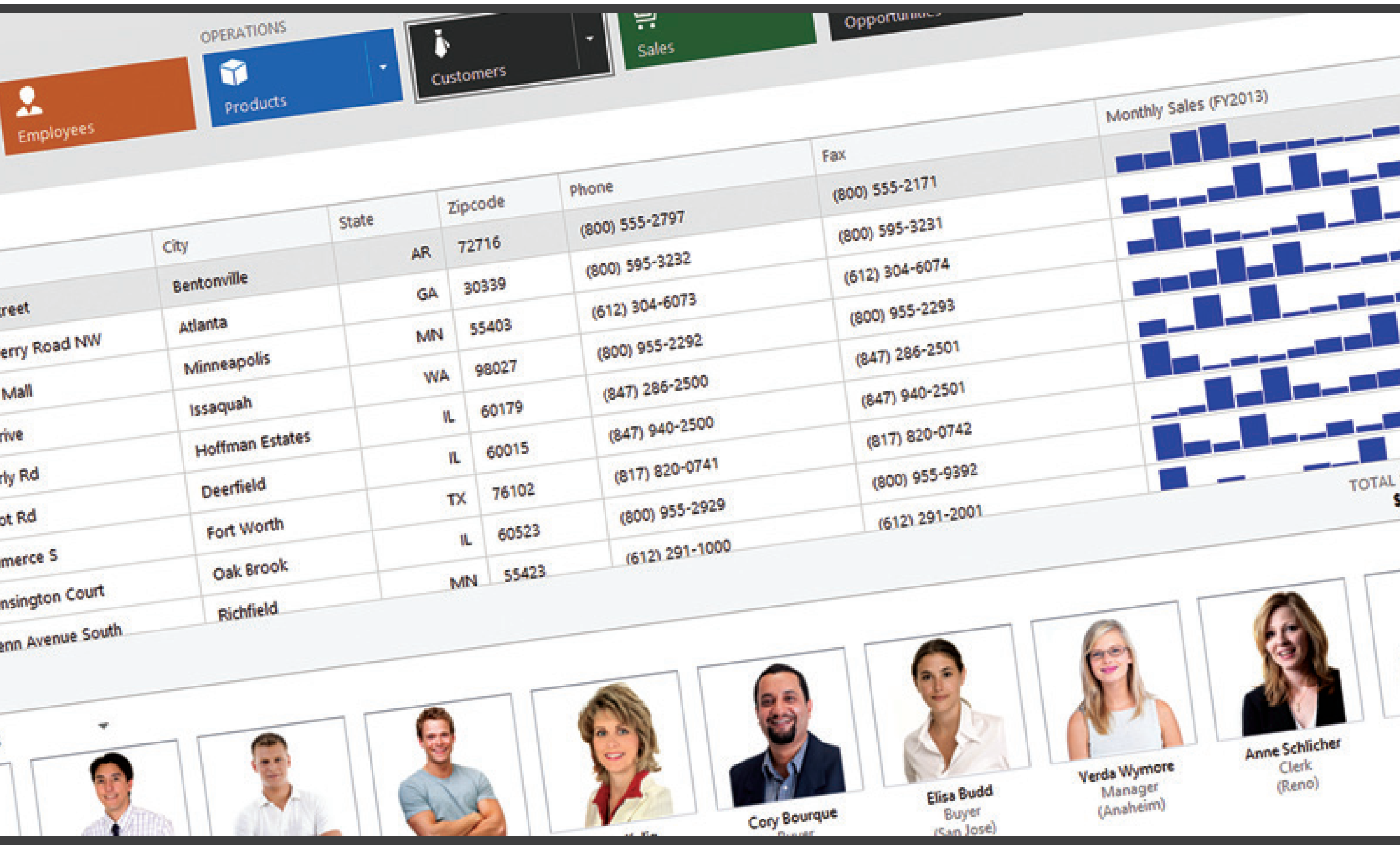
Ultimately, I started to err on the side of prizing balance, kindness, generosity, and respect in my interviewers and coworkers. Instead of dismissing these qualities as weak, I began to see them for what they really were—wisdom and love. Gradually, as I began to prize these qualities in others, I began to live them out in my own life. ■

KRISHNAN RANGACHARI *is the career coach for hackers. He's held senior engineering roles at Silicon Valley startups and large companies like Microsoft (where he started working full time at age 18). Visit radicalshifts.com to download his free career success kit.*

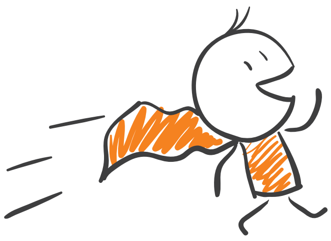
DevExpress®

Touch-Optimized Hybrid Apps

DevExpress Universal provides a comprehensive collection of UX tools so you can create touch-first apps that are built for today and ready for tomorrow.



	City	State	Zipcode	Phone	Fax
treet	Bentonville	AR	72716	(800) 555-2797	(800) 555-2171
erry Road NW	Atlanta	GA	30339	(800) 595-3232	(800) 595-3231
Mall	Minneapolis	MN	55403	(612) 304-6073	(612) 304-6074
rive	Issaquah	WA	98027	(800) 955-2292	(800) 955-2293
rly Rd	Hoffman Estates	IL	60179	(847) 286-2500	(847) 286-2501
ot Rd	Deerfield	IL	60015	(847) 940-2500	(847) 940-2501
merce S	Fort Worth	TX	76102	(817) 820-0741	(817) 820-0742
nsington Court	Oak Brook	IL	60523	(800) 955-2929	(800) 955-9392
enn Avenue South	Richfield	MN	55423	(612) 291-1000	(612) 291-2001



Your Next Great Hybrid App Starts Here

Build touch-first apps for any Windows device and leverage existing code investments.
devexpress.com/hybrid

Architecture Spinoffs of UXDD

Years go by and I'm just happy if I know and can recommend one or two effective ways of doing most software things. Don't let the emphasis of software marketing fool you. Software improves constantly, but software is not the core business of most companies out there. This means that it's more than acceptable for companies to keep using the same software for many years. According to the Tiobe.com index, in the summer of 2014, Visual Basic 6 was still in the Top 10 of most used programming languages. A year and a half later, it dropped quite a few positions, however. This seems to confirm the trend that the silent majority of companies tend to postpone full rewrites of software as much as possible. Realistically, the silent majority of companies have at least a five-year backlog in software technology that's still running in their businesses. But when it's time for companies to update those backlog systems, architects will look for state-of-the-art software.

Software Architecture Today

As far as software architecture is concerned, a couple of keywords are buzzing around these days. One is "CQRS," which is the acronym for Command and Query Responsibility Segregation, and the other is "polyglot persistence." I covered CQRS in a series of Cutting Edge articles recently. For a primer, you might want to read "CQRS for the Common Application" (msdn.com/magazine/mt147237) in the June 2015 issue of *MSDN Magazine*. CQRS is an architectural guideline that simply recommends you keep distinct the code that alters the state of the system from the code that reads and reports it. Broadly speaking, polyglot persistence refers to the emerging practice of using multiple persistence technologies in the context of the same data access layer. The expression, polyglot persistence, summarizes in two words the complex technical solutions adopted by social networks to deal with the humongous amount of data they have to manage every day. The essence of polyglot persistence is to just use the persistence technology that appears to be ideal for the type and quantity of data you have.

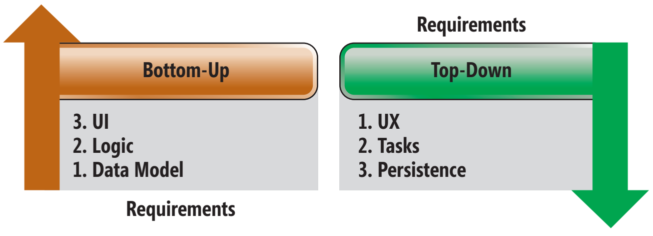


Figure 1 **Bottom-Up vs. Top-Down Design of a Software System**

While both CQRS and polyglot persistence have noticeable merits and potential benefits, it's sometimes hard to see their relevance in the context of a complex business application to revive from the ashes of a Visual Basic 6, Windows Forms or Web Forms large code base. Yet, relevance exists and missing it may be detrimental in the long run for the project and the company. At the same time, software architecture is not an act of faith. CQRS might seem like a great thing, but you must find the right perspective to see it fit into your scenario.

Well, if you look at software architecture through the lens of UXDD you can easily find a fit for both CQRS and polyglot persistence.

Well, if you look at software architecture through the lens of UXDD, you can easily find a fit for both CQRS and polyglot persistence.

Task-Based Design

Forms of UXDD workflows are being used already in companies of every size and I've personally seen flavors of it in small teams of 10 people or less, as well as in multi-billion dollar enterprises. When this happens there's a UX team that elaborates screens in collaboration with end customers, whether internal or external. The UX team produces some colorful artifacts in the form of PDF or HTML files and delivers that to the development team. The issue is that most of the time the development team ignores such artifacts until the entire back-end stack is designed and built. Too often the back end is designed taking into little or no account the actual tasks performed at the presentation level. As architects we diligently take care of the business-logic tasks and think of ways to optimize and generalize the implementation. We don't typically care enough about bottlenecks and suboptimal data presentation and navigation. When we get to discover that the actual experience users go through while working the application is less than ideal, it's usually too late to implement changes in a cost-effective manner, as shown in **Figure 1**.

/n software

We didn't invent the Internet...

...but our components help **you** power the apps that bring it to business.

TOOLS • COMPONENTS • ENTERPRISE ADAPTERS

- E-Business
  AS2, EDI/X12, NAESB, OFTP …
- Credit Card Processing
  Authorize.Net, TSYS, FDMS …
- Shipping & Tracking
  FedEx, UPS, USPS …
- Accounting & Banking
  QuickBooks, OFX …
- Internet Business
  Amazon, eBay, PayPal …
- Internet Protocols
  FTP, SMTP, IMAP, POP, WebDav …
- Secure Connectivity
  SSH, SFTP, SSL , Certificates …
- Secure Email
  S/MIME, OpenPGP …
- Network Management
  SNMP, MIB, LDAP, Monitoring …
- Compression & Encryption
  Zip, Gzip, Jar, AES …

.NET BizTalk Server SQL Server Mobile activex SharePoint 2010 DELPHI C++BUILDER Java php

The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

connectivity
powered by

To learn more please visit our website → www.nsoftware.com

Microsoft .NET, | Java | ActiveX | C++ | Delphi | C++ Builder | PHP | Objective-C | Windows | Windows Mobile | Mac OS X | Linux/Unix | iPhone

Let's face it. Well beyond its undisputed effectiveness for data storage, the relational model is misleading for software architecture. Or, at least, it started being seriously misleading after its first two decades of life. Already in the mid-1990s in the Java space the mismatch between relational and conceptual data models was a matter of fact that led to experimenting with O/RM tools. Generations of software architects (now managers) grew up with the idea that all that matters is a solid database foundation. No doubt that a solid database is the foundation of any solid software. The problem is the complexity and the amount of domain and application logic. It was no big deal until it was viable to add bits and pieces of such logic in stored procedures. Beyond that threshold software architects evolved toward the classic three-layer architecture (presentation, business, data) and later on in the domain-driven design (DDD) layered architecture: presentation, application, domain and infrastructure. No matter the number of layers, the design was essentially a bottom-up design.

The bottom-up design works beautifully if at least one of the following two conditions is verified:

1. The ideal UI is very close to the relational model.
2. Your users are dumb and passive.

Well beyond its undisputed effectiveness for data storage, the relational model is misleading for software architecture.

Software design seems an exact science when you look at tutorials ready-made to promote some cool new technology or framework. As you leave the aseptic lab to move into the real world, software design becomes the reign of fear, doubt and uncertainty—the no-man's land where only one rule exists: It depends.

UXDD simply suggests you don't start any serious development until you have some clear evidence (read: wireframes) of the ideal architecture of the information and the ideal interaction between users and the system. Top-down design starting from wireframes of the presentation layer guarantees you get really close to what users demanded. The risk of hearing the nefarious words "this is not what we asked for" is dramatically reduced. Keep in mind that if a wireframe is labeled as wrong, it's never a matter of preference; it's more likely a business matter. Ignoring the feedback of wireframes is likely creating deliberate bugs in the resulting system.

Figure 2 summarizes the architecture impedance mismatch nowadays. We have wireframes that describe the ideal presentation layer and we have the code that implements the business logic as it was understood. Unfortunately, more often than not, the two don't match. If software projects rarely live up to expectations—especially from a financial point of view—are you sure it's not because of the architecture impedance mismatch?

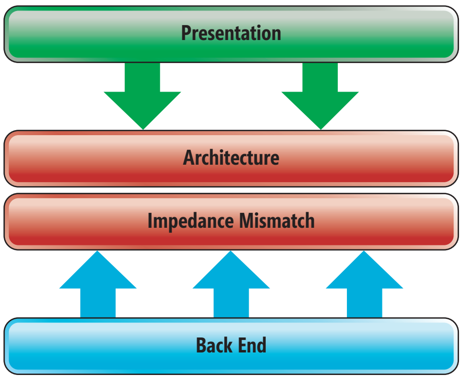


Figure 2 **Architecture Impedance Mismatch**

Addressing Architecture Impedance Mismatch

It turns out that a comprehensive domain model that covers the entire spectrum of the domain logic makes no sense at all, to put it mildly. The idea of domain models, and the entire DDD approach, came up to tackle complexity in the heart of software, but that was more than a decade ago. A lot has changed since and while some core elements of DDD—specifically the strategic design concepts and tools—are valuable more than ever today, an object-oriented domain model that comprehensively covers all read-and-write aspects of distinct bounded contexts is painful to build and unrealistic.

At the same time, the idea of a comprehensive domain model is aligned with a bottom-up vision of the software where you understand what users want, elaborate a model, code the model and then put some UI on top of it. As funny as you might see it, successful software is a nice and effective UX on top of some magical black box (MBB).

If you start any engineering effort from wireframes, then you know exactly what the MBB has to support effectively. And you also know that as long as the MBB does its job the system is really close to what users formally asked for. There's hardly any room for developers' assumptions and most of the iterations with customers are grouped in the initial phase of the project. Any change is cheap and once the UX tests have passed, most of what remains has the relevance of an implementation detail.

UXDD puts emphasis on tasks and events. In a top-down design, the starting point is a user action like a click or a selection. Each user action is bound to an entry point in the application layer. As an example, in ASP.NET MVC that means that every controller method (presentation layer) is bound to an entry point in an application layer class, as shown in **Figure 3**.

The risk of hearing the nefarious words "this is not what we asked for" is dramatically reduced.

The two SomeService classes are the section of the application layer that's visible from the part of the presentation layer represented by the SomeController class. The two SomeService classes may be part of the same physical project and assembly as the presentation or be distinct assemblies and projects. Consider that, architecturally speaking, the application layer goes hand-in-hand with the presentation layer. This means that when multiple presentation layers are required (that is, Web, mobile Web and mobile apps) it's acceptable to have multiple application layers with subsequent limited reusability of the logic.

You will have an entry point method in the application layer for each possible task the user might trigger from the approved UI. If you faithfully reproduce the wireframes you received, you know exactly what comes in and out of each screen. You just can't make it wrong. It can, perhaps, be inefficient or slow, but never wrong.

The application layer exchanges data-transfer objects with the presentation and orchestrates any required workflow for the task. In doing so, the application layer typically needs to read and write data from some persistent store, access external Web services and perform calculations. That's just domain logic and domain logic is invariant to use cases. Whether requested from a Web or mobile presentation, the business domain task runs the same and is fully reusable. In short, the idea of business logic is split into two more specific segments—application logic as the logic necessary to implement specific presentation use cases, and domain logic that's invariant to presentation and use cases.

If you faithfully reproduce the wireframes you received, you know exactly what comes in and out of each screen.

The domain logic can be designed according to a variety of patterns, including Transaction Script and Table Module. In these cases, the business rules are a foreign body injected in the classes you use. Typically, business rules form a calculator you invoke to get a response about the feasibility of an action. The Domain Model pattern—often misrepresented as the true essence of DDD—is simply a pattern that suggests you create classes that behave like the real entities and subsequently incorporate the business rules and hide their application in the methods they offer. There's no right or wrong way of designing domain logic—it's a pure matter of attitude and effectiveness.

In a task-based design, you see immediately if the action alters the state of the system or reports it. By using a CQRS design, you naturally split commands from queries, and having them in separate stacks brings the obvious benefits of writing and optimizing them separately with no risk of regression, even with the potential of simplified scalability. CQRS is a natural fit in a UXDD context.

Figure 3 **CQRS Controller**

```
public class SomeController
{
  public ActionResult SomeQueryTask(InputModel input)
  {
    var service = new QueryStack.SomeService();
    var model = service.GetActionViewModel(input);
    return View(model);
  }

  public ActionResult SomeCommandTask(InputModel input)
  {
    var service = new CommandStack.SomeService();
    service.GetActionViewModel(input);
    RedirectToAction(...);
  }
}
```

Persistence at Last!

Proceeding from the top to the bottom, inevitably persistence is the last of your concerns. Don't get it wrong: All applications need to have data persisted effectively and data that can be queried effectively. The last of your concerns doesn't mean you neglect persistence; it just means that in a user-oriented, interactive system, the design of persistence becomes critical only after other aspects—specifically UX and domain logic—have been cleared out. Just because of this, you have no constraints on which database technology should be used. Moreover, if you have a CQRS design, then you can easily have multiple persistence layers, one for commands and one for queries, managed independently and even—if that helps—based on different storage technologies and paradigms. For example, if you know that at some point analytics must be reported to the UI, you might want to adopt an event-sourcing framework and track each update and relevant actions as an event. This will give you the full log of whatever happened in the system and can be used at some point as the starting point of self-made business intelligence analysis. At the same time, if you need ready-made data to quickly serve to the user, you just keep the event store synced-up with a projection of the data that suits your presentation needs. A projection of data in this context is the state generated after a list of actions. It's the same as willing to know the bank account balance (the projection) resulting from a list of transactions (events). The query stack is often simply made of plain SQL Server views whereas the command stack might be based on relational tables, NoSQL or event store, including any combination thereof.

This is just what some call polyglot persistence.

Wrapping Up

Like it or not, users judge the system primarily from the gut feeling of their direct experience. Today, the key factor to save money on software projects is ensuring that you create, right away, exactly what users want. It's not just about doing it right, but doing it right, right away. With UXDD you know as early as possible for which output you're going to create the system and this cuts down the fixes to apply when you deploy the system and users don't actually like it. Furthermore, with a top-down design your design experience is naturally led toward modern patterns such as CQRS and polyglot persistence that, well beyond the level of buzzwords, are proven practices to build effective software. ■

Dino Esposito *is the coauthor of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2014) and "Modern Web Applications" (Microsoft Press, 2016). A technical evangelist for the Microsoft .NET Framework and Android platforms at JetBrains and frequent speaker at industry events worldwide, Esposito shares his vision of software at software2cents.wordpress.com and on Twitter: @despos.*

Thanks to the following technical expert for reviewing this article:
Jon Arne Saeteras

BEST FILE APIs

Open Create Convert Print Save

files from your *applications*!

Contact Us:
US: +1 888 277 6734
EU: +44 141 416 1112
AU: +61 2 8003 5926
sales@aspose.com

SCAN FOR
20% SAVINGS!

Refactoring an ASP.NET 5/EF6 Project and Dependency Injection

Dependency injection (DI) is all about loose coupling (bit.ly/1TZWVtW). Rather than hardcoding classes you depend on into other classes, you request them from somewhere else, ideally your class's constructor. This follows the Explicit Dependencies Principle, more clearly informing users of your class about the collaborators it requires. It also allows you to build more flexibility into your software for scenarios such as alternate configurations of a class's object instance, and it's really beneficial for writing automated tests for such classes. In my world, which is fraught with Entity Framework code, a typical example of coding without loose coupling is creating a repository or controller that instantiates a DbContext directly. I've done this thousands of times. In fact, my goal with this article is to apply what I've learned about DI to the code I wrote in my column, "The EF6, EF7 and ASP.NET 5 Soup" (msdn.com/magazine/dn973011). For example, here's a method where I instantiated a DbContext directly:

```
public List<Ninja> GetAllNinjas() {
  using (var context=new NinjaContext())
  {
    return context.Ninjas.ToList();
  }
}
```

Because I used this within an ASP.NET 5 solution and ASP.NET 5 has so much DI support built in, Rowan Miller from the EF team suggested I could improve the example by taking advantage of that DI support. I had been so focused on other aspects of the problem I hadn't even considered this. So, I went about refactoring that sample bit by bit, until I got the flow working as prescribed. Miller had actually pointed me toward a nice example written by Paweł Grudzień in his blog post, "Entity Framework 6 with ASP.NET 5" (bit.ly/1k4Tt4Y), but I explicitly chose to avert my eyes and not simply copy and paste from that blog. Instead, I worked the ideas out on my own so I could better comprehend the flow. In the end, I was happy to see that my solution aligned well with the blog post.

Inversion of Control (IoC) and IoC containers are patterns that have always seemed a bit daunting to me. Keep in mind that I've been coding for almost 30 years, so I imagine I'm not the only experienced developer who never made the mental transition to this pattern. Martin Fowler, a well-known expert in this field, points out that IoC has several meanings, but the one that aligns with DI (a term he created to clarify this flavor of IoC) is about which piece of your application is in control of creating particular objects. Without IoC, this has always been a challenge.

Code download available at msdn.com/magazine/0216magcode.

When co-authoring the Pluralsight course "Domain-Driven Design Fundamentals" (bit.ly/PS-DDD) with Steve Smith (deviq.com), I was finally led to use the StructureMap library, which has become one of the most popular IoC containers among .NET developers since its inception in 2005. The bottom line is that I was a little late to the game. With Smith's guidance, I was able to understand how it works and its benefits, but I still didn't feel quite solid with it. So after the hint from Miller, I decided to refactor my earlier sample to leverage a container that makes it easier to inject object instances into logic that needs to use them.

But First, Let's Get DRY

An initial problem in my class that houses the GetAllNinjas class shown earlier, is that I repeat the using code:

```
using(var context=new NinjaContext)
```

in other methods in that class, such as:

```
public Ninja GetOneNinja(int id) {
  using (var context=new NinjaContext())
  {
    return context.Ninjas.Find(id);
  }
}
```

The Don't Repeat Yourself (DRY) principle helps me recognize that potential pitfall. I'll move the creation of the NinjaContext instance into a constructor and share a variable such as _context with the various methods:

```
NinjaContext _context;
public NinjaRepository() {
  _context = new NinjaContext();
}
```

However, this class, which should just focus on retrieving data, is still responsible for determining how and when to create the context. I want to move decisions about how and when to create the context higher up in the stream and just let my repository use the injected context. So I'll refactor again to pass in a context created elsewhere:

```
NinjaContext _context;
public NinjaRepository(NinjaContext context) {
  _context = context;
}
```

Now the repository is on its own. I don't have to keep mucking with it to create the context. The repository doesn't care about how the context is configured, when it's created or when it's disposed. This also helps the class follow another object-oriented principle, the Single Responsibility Principle, because it's no longer responsible for managing EF contexts in addition to making database requests. When working in the repository class, I can focus on queries. I can also test it more easily because my tests can drive those decisions and won't be tripped up by a repository that's designed to be used in a way that doesn't align with how I may want to use it in automated tests.

There's another problem in my original example, which is that I hardcoded the connection string into the DbContext. I justified that at the time because it was "just a demo" and getting the connection string from the executing app (the ASP.NET 5 application) to the EF6 project was complicated and I was focused on other things. However, as I refactor this project, I'll be able to leverage the IoC to pass in the connection string from the executing application. Watch for this further on in the article.

Let ASP.NET 5 Inject the NinjaContext

But where do I move the NinjaContext creation to? The controller uses the repository. I definitely don't want to introduce EF into the controller in order to pass it into a new instance of the repository. That would create a mess (something like this):

```
public class NinjaController : Controller {
  NinjaRepository _repo;
  public NinjaController() {
    var context = new NinjaContext();
    _repo = new NinjaRepository(context);
  }
  public IActionResult Index() {
    return View(_repo.GetAllNinjas());
  }
}
```

At the same time I'm forcing the controller to be aware of EF, this code ignores the problems of instantiating dependent objects that I just solved in the repository. The controller is directly instantiating the repository class. I just want it to *use* the repository, not worry about how and when to create it or when to dispose it. Just as I injected the NinjaContext instance into the repository, I want to inject a ready-to-use repository instance into the controller.

A cleaner version of the code inside the controller class looks more like this:

```
public class NinjaController : Controller {
  NinjaRepository _repo;
  public NinjaController(NinjaRepository repo) {
    _repo = repo;
  }
  public IActionResult Index() {
    return View(_repo.GetAllNinjas());
  }
}
```

Orchestrating Object Creation with IoC Containers

Because I'm working with ASP.NET 5, rather than pull in StructureMap, I'll take advantage of the ASP.NET 5 built-in support for DI. Not only are many of the new ASP.NET classes built to accept objects being injected, but ASP.NET 5 has a service infrastructure that can coordinate what objects go where—an IoC container. It also allows you to specify the scope of objects—when they should be created and disposed—that will be created and injected. Working with the built-in support is an easier way to get started.

Before using the ASP.NET 5 DI support to help me inject my NinjaContext and NinjaRepository as needed, let's see what this looks like injecting EF7 classes, because EF7 has built-in methods to wire it up to the ASP.NET 5 DI support. The startup.cs class that's part of a standard ASP.NET 5 project has a method called ConfigureServices. This is where you tell your application how you want to wire up the dependencies so it can create and then inject the proper objects into the objects that need them. Here's that method, with everything eliminated but a configuration for EF7:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddEntityFramework()
          .AddSqlServer()
          .AddDbContext<NinjaContext>(options =>
            options.UseSqlServer(
            Configuration["Data:DefaultConnection:ConnectionString"]));
}
```

Unlike my project, which uses my EF6-based model, the project where this configuration is being executed depends on EF7. These next few paragraphs describe what's happening in this code.

Because EntityFramework .MicrosoftSqlServer was specified in its project.json file, the project references all of the relevant EF7 assemblies. One of these, the EntityFramework.Core assembly, provides the AddEntityFramework extension method to IServiceCollection, allowing me to add in the Entity Framework service. The EntityFramework .MicrosoftSqlServer dll provides the AddSqlServer extension method that's appended to AddEntityFramework. This stuffs the SqlServer service into the IoC container so that EF will know to use that when it's looking for a database provider.

AddDbContext comes from the EF core. This code adds the specified DbContext instance (with the specified options) to the ASP.NET 5 built-in container. Any class that requests a DbContext in its constructor (and that ASP.NET 5 is constructing) will have the configured DbContext provided to it when it's created. So this code adds the NinjaContext as a known type that the service will instantiate as needed. In addition, the code specifies that when constructing a NinjaContext, it should use the string found in the configuration code (which in this case is coming from an ASP.NET 5 appsettings.json file, created by the project template) as a SqlServer configuration option. Because ConfigureService runs in the startup code, when any code in the application expects a NinjaContext but no specific instance is provided, ASP.NET 5 will instantiate a new NinjaContext object using the specified connection string and pass that in.

So that's all very nicely built-in with EF7. Unfortunately, none of this exists for EF6. But now that you have an idea how the services work, the pattern for adding the EF6 NinjaContext to the application's services should make sense.

Adding Services Not Built for ASP.NET 5

In addition to services that are built to work with ASP.NET 5, which have nice extensions like AddEntityFramework and AddMvc, it's possible to add other dependencies. The IServicesCollection interface provides a plain vanilla Add method, along with a set of methods to specify the lifetime of the service being added: AddScoped, AddSingleton and AddTransient. I'll focus on AddScoped for my solution because it scopes the lifetime of the requested instance to each HTTP request in the MVC application where I want to use my EF6Model project. The application won't try to share an instance across requests. This will emulate what I was originally achieving by creating and disposing my NinjaContext within each controller action, because each controller action was responding to a single request.

Remember that I have two classes that need objects injected. The NinjaRepository class needs NinjaContext, and NinjaController needs a NinjaRepository object.

In the startup.cs ConfigureServices method I begin by adding:

```
services.AddScoped<NinjaRepository>();
services.AddScoped<NinjaContext>();
```

Now my application is aware of these types and will instantiate them when requested by another type's constructor.

When the controller constructor is looking for a NinjaRepository to be passed in as a parameter:

```
public NinjaController(NinjaRepository repo) {
    _repo = repo;
  }
```

but none has been passed in, the service will create a NinjaRepository on the fly. This is referred to as "constructor injection". When the NinjaRepository expects a NinjaContext instance and none has been passed in, the service will know to instantiate that, as well.

Dependency injection (DI) is all about loose coupling.

Remember the connection string hack in my DbContext, which I pointed out earlier? Now I can instruct the AddScoped method that constructs the NinjaContext about the connection string. I'll put the string in the appsetting.json file again. Here's the appropriate section of that file:

```
"Data": {
    "DefaultConnection": {
      "NinjaConnectionString":
      "Server=(localdb)\\mssqllocaldb;Database=NinjaContext;
      Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
```

Note that JSON doesn't support line wrapping, so the string that begins with Server= can't be wrapped in your JSON file. It's wrapped here only for readability.

I've modified the NinjaContext constructor to take in a connection string and use it in the DbContext overload, which also takes a connection string:

```
public NinjaContext(string connectionString):
    base(connectionString) { }
```

Now I can tell AddScoped that when it sees a NinjaContext, it should construct it using that overload, passing in the NinjaConnectionString found in appsettings.json:

```
services.AddScoped<NinjaContext>
(serviceProvider=>new NinjaContext
  (Configuration["Data:DefaultConnection:NinjaConnectionString"]));
```

Figure 1 **NinjaRepository Using an Interface**

```
public interface INinjaRepository
{
  List<Ninja> GetAllNinjas();
}

public class NinjaRepository : INinjaRepository
{
  NinjaContext _context;
  public NinjaRepository(NinjaContext context) {
    _context = context;
  }
  public List<Ninja> GetAllNinjas() {
    return _context.Ninjas.ToList();
  }
}
```

With this last change, the solution that I broke while refactoring now works from end to end. The start-up logic sets up the app for injecting the repository and context. When the app routes to the default controller (which uses the repository that uses the context), the needed objects are created on the fly and the data is retrieved from the database. My ASP.NET 5 application takes advantage of its built-in DI to interact with an older assembly where I used EF6 to build my model.

Interfaces for Flexibility

There is one last possible improvement, which is to take advantage of interfaces. If there's a possibility that I might want to use a different version of my NinjaRepository or NinjaContext class, I can implement interfaces throughout. I can't foresee needing to have a variation on NinjaContext, so I'll only create an interface for the repository class.

As shown in **Figure 1**, the NinjaRepository now implements an INinjaRepository contract.

The controller in the ASP.NET 5 MVC application now uses the INinjaRepository interface instead of the concrete implementation, NinjaRepository:

```
public class NinjaController : Controller {
  INinjaRepository _repo;

  public NinjaController(INinjaRepository repo) {
    _repo = repo;
  }
  public IActionResult Index() {
    return View(_repo.GetAllNinjas());
  }
}
```

I've modified the AddScoped method for the NinjaRepository to tell ASP.NET 5 to use the appropriate implementation (currently NinjaRepository) whenever the interface is required:

```
services.AddScoped<INinjaRepository, NinjaRepository>();
```

When it's time for a new version, or if I'm using a different implementation of the interface in a different application, I can modify the AddScoped method to use the correct implementation.

Learn By Doing, Don't Copy and Paste

I'm grateful that Miller gently challenged me to refactor my solution. Naturally, my refactoring didn't go as smoothly as it might appear based on what I've written. Because I didn't just copy from someone else's solution, I did some things incorrectly at first. Learning what was wrong and figuring out the correct code led me to success and was hugely beneficial to my understanding of DI and IoC. I hope my explanations will give you that benefit without your having to bump your head as much as I did. ■

Julie Lerman *is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other .NET 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 juliel.me/PS-Videos.*

Thanks to the following technical expert for reviewing this article:
Steve Smith (ardalis.com)

DOMAINS | MAIL | **HOSTING** | eCOMMERCE | SERVERS

12 MONTHS
from **$0.99** per month,
then starting at $6.99/month*

MANAGED WORDPRESS

100 % PERFORMANCE

- **NEW! Unlimited SSD** webspace
- **NEW! Unlimited SSD** databases
- **NEW! PHP 7 with OPcache**
- **Unlimited** traffic
- **Unlimited** e-mail accounts
- **2 GB RAM** guaranteed

100 % SECURITY

- **NEW! 1&1 DDoS Protection with NGINX** for greater performance, reliability, and maximum security
- **Geo-redundancy:** simultaneous storage in separate data centers
- 1&1 CDN with Railgun™
- 1&1 SiteLock

100 % RELIABILITY

- **NEW! WP Assistant** – unique guided installation and initial setup
- **Includes ready-to-use templates**
- **24/7 customer support**
- **1&1 WP Expert Support**
- 1&1 Community

1 TRIAL TRY FOR 30 DAYS ★★★★★

1 MONTH FLEXIBLE PAYMENT OPTIONS ★★★★★

1 CALL SPEAK WITH AN EXPERT 24/7 ★★★★★

1 (877) 461-2631

1&1®

1and1.com

* 1&1 Managed WP Basic is $0.99/month for the first 12 months, after which the regular pricing of $6.99/month applies. Visit www.1and1.com and www.1and1.com/Gtc for full promotional details. 1&1 and the 1&1 logo are trademarks of 1&1 Internet, all other trademarks are property of their respective owners. Rubik's Cube® used by permission of Rubik's Brand Ltd. © 2016 1&1 Internet Inc. All rights reserved.

Universal Windows Platform Apps for Web Developers

Tim Kulp

As an enterprise Web developer, you have a keen understanding of HTML, CSS and JavaScript. You can build out responsive Web applications that adapt to the screen size while maintaining functionality across your supported browsers or devices. The same skills you use to light up the Web can be used to build Universal Windows Platform (UWP) apps. Whether you're building for desktop, mobile or any Windows 10 platform, your lessons learned from cross-browser adaptive Web applications will give you a head start in UWP.

In this article, I explore how to use knowledge from Web development to go from building cross-browser applications to flexible UWP apps that run on any Windows device. To do this, I start by examining how the fundamentals of building responsive interfaces translate from CSS and HTML to UWP. Next, I examine using VisualStates and XAML Views to structure your app based on specific device capabilities. Finally, I use Adaptive Code to target devices based on how Plain Old JavaScript is used to target specific browsers.

This article discusses:

- Identifying key skills to build responsive UWP apps
- Mapping how to use what you already know in HTML/CSS/JavaScript to build UWP apps
- Using various responsive approaches to build apps for multiple form factors
- Using a single code base for cross-platform development

Technologies discussed:

HTML/CSS/JavaScript, XAML, C#, Universal Windows Platform

Why XAML for Web Developers?

In this article, I'll build parallels from Web development to XAML. Web developers might already have a strong HTML/CSS/JavaScript skillset and want to build UWP with that skillset. If you love HTML and JavaScript, feel free to stick with it. If you're new to UWP and unsure where to start, XAML is a great tool to help you due to its strongly typed nature.

A common example of this is using a grid-based layout in XAML versus HTML. In XAML, building a grid layout starts with adding a grid control, defining the columns and rows, then assigning each control within the grid to a specific cell or row. In HTML, there are numerous ways to build a grid layout such as:

- Using floats with defined height and width to create cells with clears to start a new row
- Using display:grid on the container element with each child element having a column and row defined
- Using a table with tr and td elements

Which is implemented is up to your knowledge of CSS or HTML and these approaches won't give you help through tools such as IntelliSense, as the grid control will in XAML. The strongly typed controls of XAML make it easier to know how to build a UI through IntelliSense, which is extremely helpful for developers who are new to UWP.

Strongly typed code also provides a lot of value when troubleshooting issues. In HTML/CSS/JavaScript, developers have great flexibility to bend the rules to the demands of the app using loosely typed code. This is great for building apps but can be a nightmare for supporting apps. Troubleshooting loosely typed code can be a challenge when types change or objects change dynamically. As enterprise developers, building apps is fun, but at some point, someone is going to have to keep this app running. The ability to easily navigate the functionality of an app through strongly typed objects and IntelliSense helps the support team understand the app.

If you're passionate about HTML/CSS/JavaScript, UWP gives you a platform to use your existing code to produce great apps. HTML/CSS/JavaScript and XAML are both great tools with lots of pros and cons. There are two articles about why one author prefers XAML to JavaScript (bit.ly/1NxUxqh) and another prefers JavaScript to XAML (bit.ly/1RSLZ2G). I love HTML for Web applications but I encourage you to explore XAML if you're new to UWP to learn the controls within UWP, to keep support costs down for your team through strongly typed code with rich IntelliSense integration and to have fun learning something new.

Building on What You Already Know

UWP has many similarities with the fundamentals of Web design. Basic ideas such as separation of concerns in Web development between HTML and JavaScript translate in UWP to XAML and the XAML.cs codebehind file. All logic goes into the codebehind, while all presentation is maintained in the XAML file (just as all logic goes in JavaScript while presentation is in HTML with help from CSS). Further, many modern Web applications leverage frameworks such as Knockout and AngularJS to implement data binding through the Model-View-ViewModel (MVVM) design pattern. Knowledge of these frameworks and MVVM in general is the basis for understanding data binding in UWP. While the syntax is different between Web development and UWP, when it comes to basic concepts, Web developers have a strong foundation for building apps that cross devices, browsers and capabilities.

There are differences between Web development and UWP that I won't cover in this article, such as state management and data storage. For this article, I'll focus on building the UI and ensuring that the app can interact with the device on which it's running.

Positioning: Float and Clear to RelativePanel

In HTML, you determine the positioning of each element by where it is in the Document Object Model. HTML is top-down with each element being rendered in order from the first element declared to the last. When CSS was introduced, it let elements have sophisticated layouts based on setting the element's display style (inline, block and so on), position (relative or absolute), as well as float and clear. Using float, Web developers could take an HTML element out of the top-down flow, and place the element to the left (float: left) or right (float: right) of the containing element.

Imagine a simple layout including header, main content, sidebar and footer. Using float instructs the browser to render the sidebar to the right edge of the container and to render the main content to the left edge of the container. Using float, elements will position beside each other to the left or right, depending on which float value is specified. Clear is used to stop the floating of elements and return to the standard top-down flow of HTML. **Figure 1** shows an example of using float to build a simple layout.

Figure 1 **Using Float to Build a Simple Layout**

```
div {
  width: 100%;
}
mainContent {
  width: 60%; float: left;
}
  sidebar{
  width: 40%; float: right;
}
clearer {
  clear: both;
}
CSS for design

<header>

</header>
<div>
  <section class="content"></section>
  <section class="sidebar"></section>
  <div class="clearer"></div>
</div>
<footer>

</footer>
HTML for design
```

Where Web developers use float and clear to create a layout, UWP provides a control called the RelativePanel, which, as the name suggests, lets a layout be defined using relative relationships to other controls. Like float, RelativePanels let developers manage how controls are positioned relative to an anchor control. CSS classes are used to determine positioning of the elements of the Web page. To replicate the same layout, use the RelativePanel and the RelativePanel's attached properties within the controls:

```
<RelativePanel>
  <!-- Header is the anchor object for the relative panel -->
  <TextBlock Text="Header" Name="tbHeader"></TextBlock>
  <TextBlock Text="Content" RelativePanel.Below="tbHeader"
    Name="tbContent"></TextBlock>
  <TextBlock Text="SideBar" RelativePanel.RightOf="tbContent"
    RelativePanel.Below="tbHeader" Name="tbSideBar"></TextBlock>
  <TextBlock Text="Footer" RelativePanel.Below="tbSideBar"
    Name="tbFooter"></TextBlock>
</RelativePanel>
```

In this code block, the positioning of each control is relative to the position of the anchor control (in this case, the anchor is the header TextBlock). Using RelativePanel, each control can be assigned to where the control should appear on the screen in relation to the other controls. Where Web developers would use float: left, UWP developers would use RelativePanel.LeftOf or RelativePanel.RightOf (for float: right) to position content. While this is similar to using float, there isn't a concept of using clear to return to the normal flow; instead, just note that something is below a previous element. This simplifies troubleshooting layout issues as managing floats and clears can get challenging for developers who do not have strong CSS skills. Using the RelativePanel provides a declarative way to specify where a control should appear in relation to other controls. When the RelativePanel is closed, the app will return to the normal rendering flow of XAML (which is top-down like HTML).

Scaling: Percentages to Pixels

Building a responsive Web application through resizing means using relative sizing for elements. Using the header, content, sidebar and footer page layout, imagine this UI was originally built for a desktop screen. Web designers would first identify the optimal pixel width for the page for this layout. In this example, page width will be 1000px. As each element is built in the design, the element is built to the pixel width keeping the 1000px container in mind. In HTML, the content section would be 800px wide while the sidebar section would be 200px wide. Using the formula: target / context = percentage, the content section is 80 percent of the context (whereas the context = the 1000px page) while the sidebar is 20 percent.

Using percentages in Web design lets the layout resize as the container resizes. In this case, if the 1000px page object were resized by the user to only 659px, the content and sidebar would resize to 527px and 131px, respectively. Similarly, building styles to use em instead of specific point or pixel sizes lets the font scale according to the context. These practices help ensure that a design maintains the proportional sizing independent of the window size.

While using percentages seems like simple math, there are other factors that are involved in how elements scale, such as pixel density of the device and orientation, which add an element of unpredictability to your design. UWP simplifies scaling by using the concept of the "effective pixel" for all measurements. An effective pixel is not the same as a single pixel. Effective pixels use the UWP scaling algorithm to know how to represent one effective pixel based on standard distance of the device from the user and pixel density.

As an example, a Surface Hub would have a much higher pixel density than a tablet or phone. UWP developers only need to build to effective pixels in tools such as Blend and let the scaling algorithm handle the complex calculations necessary to shrink or grow accordingly. One thing to keep in mind: Effective pixels must be in multiples of four. Based on how the scaling algorithm works, using multiples of four will ensure clean edges as the UI scales.

ViewStates and Media Queries

In Web development, CSS provides the layout information for an application. Building with percentages lets an application resize, but at some point, the design needs to break and reform to meet the changing display demands. A layout built for a mobile device such as a tablet isn't going to be the same as a layout built for an 80-inch-plus presentation device such as the Surface Hub. An older analogy for Web design is the case where a user would want to print a screen-based design. CSS let designers fix the screen-to-print dilemma with CSS media queries. As the user printed the design, the browser would use the print CSS instead of the screen CSS. As responsive Web design has grown, media queries have grown to support information that is more detailed. Here's an example CSS media query:

```
<link type="text/css" rel="stylesheet"
  href="styles/719style.css"
  media="screen and (max-device-width: 719px)"/>
```

In this query, the 719style.css file is applied if the media displaying the Web application is a screen with a device width less than or equal to 719px. As an example, this media query can be used to clear the float values and present the content and sidebar elements in a stacked design versus side-by-side. Using media queries lets Web developers customize the display based on screen size, resolution, orientation and many more options (see complete list for CSS3 at bit.ly/1riUA2h).

Just like media queries, ViewStates can be used to reposition or rearchitect controls based on the triggers.

In UWP, the ViewStateManager can be used as media queries to alter the application design based on defined parameters. The VisualStateManager contains one or more VisualStateGroups, a container object of multiple ViewStates. Each ViewState holds the setters (what properties get updated for each control) and triggers (what makes the setters change). ViewStateManager manages the triggers to know when to apply a specific ViewState's setter values. In the terms of CSS, the triggers are like media queries and the setters are the style values within the stylesheet referenced in the media query. See the example in **Figure 2**.

In this code, two VisualStates are set up. The LessThan720 ViewState is triggered when the window width is 1px to 719px. When the window expands to 720px or higher, then the GreaterThan720 ViewState is triggered. Each of these ViewStates work with the tbSideBar control's RelativePanel settings. If the window size is less than 720, then the screen isn't big enough to support a design where content is beside the sidebar. In this situation, the LessThan720 ViewState will be triggered, which will stack the sidebar below the main content TextBlock. This is similar to using the media query in that I can set float: none in a LessThan719.css file.

Just like media queries, ViewStates can be used to reposition or rearchitect controls based on the triggers. Managing ViewStates and ViewStateGroups can get complicated as an interface grows in complexity. Managing complex ViewState changes are easiest with Blend for Visual Studio 2015. Using Blend's editor, you can create new VisualStates and see the changes in the app design as you make them. Blend will handle writing all the XAML for you and ensure that you provide the necessary data to trigger the VisualState change. Microsoft Virtual Academy has great walk-through videos on using Blend for ViewState management at bit.ly/1P94e32.

Figure 2 **Reposition Controls Based on VisualState Triggers**

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ResponseStateGroup">
    <VisualState x:Name="LessThan720">
      <VisualState.Setters>
      <Setter Target="tbSideBar.(RelativePanel.Below)" Value="tbContent"/>
        <Setter Target="tbSideBar.(RelativePanel.RightOf)" Value=""/>
      </VisualState.Setters>
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="1"/>
      </VisualState.StateTriggers>
    </VisualState>
    <VisualState x:Name="GreaterThan720">
      <VisualState.Setters>
        <Setter Target="tbSideBar.(RelativePanel.Below)" Value="tbHeader"/>
        <Setter Target=" tbSideBar.(RelativePanel.RightOf)" Value="tbContent"/>
      </VisualState.Setters>
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="720"/>
      </VisualState.StateTriggers>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Using Views to Rearchitect the Experience

Sometimes a mobile site is a different UI due to a reduction in use cases or change in focus that the mobile site provides versus the full desktop experience. In this scenario, Web developers would tailor content to the mobile experience for a more streamlined experience, or to highlight the capabilities of the mobile device. Using various detection methods, Web developers can redirect users to a rearchitected experience tailored to their device, often known as the m.webapp.com site.

UWP provides the same capability through Views. Depending on the app, working with different device families might call for such a stark difference in UI for which the ViewStateManager might not be the right tool. Views let developers leverage the existing back-end code with a new XAML UI. You can simplify Using Views through using well-structured ViewModels whereas you can leverage a single ViewModel object by multiple Views. Knowledge of Knockout or AngularJS will help Web developers build proper ViewModels in UWP for tailored UXes for a specific device family.

You set up a View for a specific device by creating a folder within the app's Views folder.

Within the Views folder, create a new folder called DeviceFamily-Mobile. This tells the Modern Resource Technology to use the MainPage.xaml view found in the DeviceFamily-Mobile folder when the MainPage is requested on a device in the mobile device family (such as a phone). If the request for MainPage comes from any other device family, the response will be the standard MainPage. This functionality lets UWP developers build targeted UI for use cases that are unique to specific Windows device families.

Adaptive Code

In building Web applications, not all browsers are the same. In an enterprise, you might have a corporate standard, but when developing for external users, complexity comes from various OSes, browser types and versions. Web developers have many tricks up their sleeves to handle these differences gracefully. Libraries such as Modernizr (modernizr.com) have made handling these complexities easier, but enabling features based on device or browser is nothing new to Web developers.

Building UWP apps can have the same complexities as cross-browser functionality. Imagine a note-taking application where users can add pictures to their notes. The app could leverage the built-in camera functionality of a phone to take a picture, or could just let users view images that already exist on their devices.

The first step in using device-specific capabilities is ensuring the proper Extension APIs are included in the project. As an example, in the note-taking app, the hardware buttons would need to be accessible on a phone. To use those buttons, you must add a reference to the Windows Mobile Extensions for the UWP. This is done by adding a reference to the project (just like any other reference), then selecting Universal Windows, then Extensions. You will see a list of possible extensions such as Desktop, Mobile and Team (for Surface Hub) extensions. Select which extensions you need to add to the project and click OK.

Wherein JavaScript detecting the browser's capabilities would be through a series of navigator checks, in UWP the app checks for the presence of the needed API through the IsTypePresent method. In the note-taking app, check for the hardware button to use the camera through the following code:

```
string apiName = "Windows.Phone.UI.Input.HardwareButtons";
if (Windows.Foundation.Metadata.ApiInformation.IsTypePresent(apiName))
{
  Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
    HardwareButtons_CameraPressed;
}
```

This short code lets the app target specific device capabilities added through the Extension APIs. By wrapping the CameraPressed event handler declaration with the IsTypePresent method, you ensure that you don't try to register the event handler when the API isn't present. There are tools to help ensure that the API checks occur so that the app doesn't crash when the API isn't present. PlatformSpecific is an excellent NuGet package that simplifies identifying and wrapping any reference to an extension API that isn't first validated through the ApiInformation.IsTypePresent method. Learn more about this NuGet package from the PlatformSpecific GitHub site at bit.ly/1GvhkF0.

Just as in Web development, sometimes a specific version of the browser needs targeting for a client or corporate standard. In these situations, Web developers need to focus on a specific browser configuration that might not match what the rest of the Internet is using.

Similarly, UWP developers might need to target specific contracts of an extension API to maintain existing code. This is very useful in enterprise applications where the IT operations team could have a fast loop and slow loop for deploying updates to employee computers. The fast loop might be getting a new great feature of some extension API that must be implemented in the app immediately. In this situation, the slow loop users still need their functionality. Using IsApiContractPresent UWP can check that the extension API is available and that the specific version expected is available prior to executing the code:

```
if(Windows.Foundation.Metadata.ApiInformation.IsApiContractPresent(apiName, 3))
{
  newKillerFunction();
}
```

In this code segment, the app will only run the newKillerFunction if the apiName provided is at version 3. If the version is less than 3, then newKillerFunction will not run. If a greater version is present (as example, version 4), then newKillerFunction will execute.

Wrapping Up

UWP development leverages many of the skills and knowledge that Web developers use to build responsive, cross-browser Web applications. Designing layout, responding to differences in displays (both static and dynamic), as well as system capabilities, are all common practices to Web developers from dealing with the wild world of Web browsers. Applying these skills in UWP development will help in building rich UXes that adapt to screen size, device and capabilities. ■

Tim Kulp *is a senior technical architect living in Baltimore, Md. He is a Web, mobile and UWP developer, as well as author, painter, dad and "wannabe Mad Scientist Maker." Find him on Twitter: @seccode or via LinkedIn: linkedin.com/in/timkulp.*

Thanks to the following Microsoft technical expert for reviewing this article: *Kevin Hill*

Implementing a UWP App with the Official OneDrive SDK

Laurent Bugnion

In part one of this two-part series (msdn.com/magazine/mt614268), I talked about using the OneDrive APIs over HTTP directly from a Universal Windows Platform (UWP) app, using the HttpClient to place the calls. This gave me the chance to go quite deep into the fundamentals of REST and to see how a .NET application can take advantage of modern programming techniques such as the async/await keywords and the HttpClient library to place calls to modern Web APIs. I also showed how the authentication mechanism works with the OAuth protocol, which lets a user enter his credential in a hosted Web page so that the communication between the user and the credential service is completely transparent to the hosting application. I implemented this using a WebView hosted in a XAML page.

Implementing the authentication and the REST calls isn't very difficult, but it is work, especially when you want to keep your library up-to-date with all the latest changes and additions. This is why Microsoft has published the OneDrive SDK, which you can add to your application in an easy manner (using the NuGet Package Manager). This is a very useful component, and in the true "new Microsoft" manner, the library is open sourced and published on GitHub at bit.ly/1WX0Y03.

To illustrate this article, a complete sample can be found at 1drv.ms/1ObySnz.

This article discusses:

- How the authentication works with the official SDK, either with the logged-in user's credentials transparently, or with an authentication broker
- How the new app is linked with a reserved name in the Windows Store
- How various file operations are executed by building requests and sending them to the service

Technologies discussed:

OneDrive SDK, REST, OAuth, Windows 10, Universal Windows Platform

Code download available at:

1drv.ms/1ObySnz

Authentication with the New SDK

One of the biggest differences between the "low-level" implementation described in the previous article and the official SDK is the authentication mechanism. Previously, I implemented OAuth manually and had to create a number of parts that had to play together:

- A WebView to present the user with a Web page sent by the Microsoft authentication service, so that the username and password are sent directly to the service, without the application being aware of this.
- A XAML page hosting the WebView, and waiting to parse the authentication token returned by the service when the process was successful.
- A client-side authentication service that's responsible for deciding if the authentication token is available and valid, and for showing the authentication page if that's not the case.

The new SDK simplifies this greatly, for the developer and for the user.

Authentication for the Developer

For the developer, the authentication process is almost trivial now. The whole workflow is handled by the SDK. Even better, in the background, the SDK uses a feature of Windows Store apps named the OnlineIdAuthenticator. This class handles the authentication mechanism at the OS level and removes the need to parse the authentication token.

In addition to OnlineIdAuthenticator, the OneDrive SDK for Windows 10 also supports the WebAuthenticationBroker class, which provides a more "classic" way of logging in by entering username and password, but also takes care of parsing the authentication token and all dialog with the service. You'll see examples of these two authentication modes later in this article.

Authentication for the User

For the end user, the main advantage is in OnlineIdAuthenticator taking advantage of the logged-in Windows user's credentials, which are already available at the OS level. If the Windows user is logging into the OneDrive account associated with his Windows account (which is often the case), he doesn't even need to re-enter his username and password. The whole authentication mechanism is transparent for him.

Note: *In addition to Microsoft Accounts (MSA) for consumers, such as Outlook.com, Live.com or Hotmail.com addresses, the OneDrive SDK also supports Azure Active Directory (Azure AD) for business users. This is great for enterprises whose users are managed this way. MSA is available for the Microsoft .NET Framework 4.5.1 (such as Windows Presentation Foundation applications or Windows Forms), Windows Store and Windows Phone applications. Azure AD is available for Windows Forms, Windows Store 8.1 and UWP apps.*

Associating a New UWP App in the Windows Store

For the OnlineIdAuthenticator to work, it's necessary to register the new application with the Windows Store. Note that you don't need to actually submit your app to the Windows Store to use the OneDrive SDK. The registration will create some unique IDs for your application, which must be entered into the application manifest (Package.appxmanifest). When users attempt to authenticate themselves to the OneDrive service, these IDs are used to uniquely identify the application. As such, the process of registering your application with the Windows Store replaces the Client ID that you were getting earlier from the OneDrive developer center.

To register your application with the Windows Store, you need to go to the Windows Dev Center at dev.windows.com and log in with your Windows developer account to access the Dashboard.

Figure 1 **Implementing the Click Event Handler**

```
private IOneDriveClient _client;

public MainPage()
{
  InitializeComponent();

  AuthenticateButton.Click += async (s, e) =>
  {
    var scopes = new[]
    {
      "onedrive.readwrite",
      "onedrive.appfolder",
      "wl.signin"
    };

    _client = OneDriveClientExtensions.GetClientUsingOnlineIdAuthenticator(
      _scopes);

    var session = await client.AuthenticateAsync();
    Debug.WriteLine($"Token: {session.AccessToken}");
  };
}
```

Depending on your status, you might have to register for the account and provide payment information.

Once logged in on the Dashboard, look for the "Submit your app" link. This will take you to an overview page where you can see existing apps and submit the new one, getting it registered with the Windows Store. Creating a new Windows Store app requires a unique app name, which you will enter after clicking the Create a new app button. Then, click on Check availability to make sure the app name you chose is indeed unique. Because the name you chose will be reserved, it's good practice to use temporary names for test applications that will never be published. This way you avoid blocking another developer's app name. Also, make sure that you own the rights to the name that you're reserving to avoid legal actions from a brand owner. Finally, note that if you don't submit the application within a year, the name reservation will be lost. For the purpose of the sample, I reserved the name TestOneDriveSdk_001, which happened to be available.

Implementing and Testing the Authentication

Now is the time to create the application and to add the OneDrive SDK. First, using Visual Studio, create a new UWP application for Windows 10. The application doesn't have to have the same name as the one you reserved in the Windows Store, but it is good practice to use the same name.

After the application is created, open the NuGet Package Manager by right-clicking on the project in the Solution Explorer and selecting Manage NuGet Packages from the context menu. Select Browse on top of the Package Manager, search for the Microsoft.OneDriveSDK package and add it to the application.

Now you'll add authentication to the application with the following steps:

- Open MainPage.xaml and add a button to the page and name it AuthenticateButton.
- In MainPage.xaml.cs, implement the Click event handler as shown in **Figure 1**.

In the code in **Figure 1**, you recognize the scopes that you already used in the previous article's sample. These will provide you with read/write access to the files, as well as the Application folder. As a reminder, this special folder is created in the Apps folder directly under the root and will carry the same name as the application. This is a good place to save roaming settings, online documents and so on.

Before you run the application to test it, you still need to associate it with the Windows Store name that you reserved earlier. To do this, follow these steps:

- In Solution Explorer, right-click on the project and select Store/Associate App with the Store. This is what you need to do before publishing a UWP app to the Windows Store. In this case, it's done very early in development because the information generated is needed.
- In the dialog, Associate Your App with the Windows Store, press the Next button and then sign into the store. Make sure to use the same credentials you used to reserve the name.
- After signing in, you should see the application name that you reserved. Note that you can also reserve a name directly from this dialog if you don't want to use the Web-based dashboard.
- Select the name and click on Next, then on Associate.

Now you're ready to test the authentication mechanism. Run the application in debug mode and click on the Authenticate button. You should see a dialog on top of your app asking for your consent. Note, however, that you don't have to enter your Microsoft credentials. Instead, your Windows login credentials are used automatically.

After executing the code in **Figure 1**, you'll see the SessionToken (authentication token) in the Output window. You'll be able to use this token in subsequent API calls just like you did in the first article. If you run the application again later and press Authenticate again, you probably won't even have to confirm again because your consent is cached until you sign off. This provides a seamless sign-in experience for the user.

Why Can We Not Switch Accounts?

The sample shown in **Figure 1** uses the SDK's GetClientUsingOnlineIdAuthenticator extension method, which, under the covers, uses the OnlineIdAuthenticator class. As I explained, this object uses the logged-in user's credentials, which makes the login experience extremely easy for the large majority of users.

The downside of this approach, however, is that there's no way to select a different account when logging in. As such, this authentication method is great for basic users who have only one OneDrive account associated with their Microsoft accounts. But for power users who have more than one OneDrive account, the OnlineIdAuthenticator might be too limited. Thankfully, you can switch to a different method quite easily.

Using a WebAuthenticationBroker

If the user needs to sign in with his credentials explicitly (for example, to let him switch to a different account than the one he logged into Windows with), it's better to use the WebAuthenticationBroker class. Here, too, the OneDrive SDK is hiding most of the complexity and you can use the following code to create the OneDrive client:

```
var client = OneDriveClientExtensions.GetClientUsingWebAuthenticationBroker(
  "[CLIENT ID]",
  _scopes);
```

In the code shown in **Figure 1**, the [CLIENT ID] string must be replaced with your application's Client ID. As a reminder, this is a unique ID that you can retrieve in the Windows Dev Center in your application's details.

If you run the application with this minor change, you'll now see a different dialog. This dialog lets the user switch to a different account, which is good. However, the user needs to enter his

Figure 2 **Downloading a Text File's Content**

```
var builder = _client.Drive.Root
  .ItemWithPath("Public/Test/MyFile.txt");

var file = await builder
  .Request()
  .GetAsync();

var contentStream = await builder.Content
  .Request()
  .GetAsync();

Debug.WriteLine($"Content for file {file.Name}:");

using (var reader = new StreamReader(contentStream))
{
  Debug.WriteLine(reader.ReadToEnd());
}
```

password manually, which is less seamless than the previous workflow. As is usual in engineering, it's up to you to select the method best suited to the use case you try to solve.

Using the SDK to Access Folders and Files

Once the client is available and authenticated, the OneDrive SDK provides several methods to retrieve information about Drive, Folders and Files. In the previous article, you saw how the file structure is composed of Items, Folders, Files, Audio, Images, Photos and Videos. Until now, you had been getting a JSON response from the service and deserializing it manually into the corresponding C# classes. With the OneDrive SDK, this is no longer necessary because it'll take care of that step for you. To illustrate this, I'll rework the same sample application as in the previous article to use the official SDK instead.

Building the Requests

To send requests to the OneDrive REST API, the SDK uses an object hierarchy composed of so-called "request builders." For example, getting the user's Drive is done with _client.Drive and getting his Root folder is done with _client.Drive.Root. The most common request builders are listed later in this article.

There are, of course, many possible combinations. Once a request builder has been obtained, the actual request is created by using the Request method, and sent with one of the HTTP methods, for example, GetAsync or PostAsync. For instance, the code in **Figure 2** will get the metainformation for the file located at the path Public/Test/MyFile.txt and its content as a stream.

Listing a Few of the Most Common Requests

The most common requests are listed here as request builders:

- _client.Drive: Builds a request to access the OneDrive itself and get its properties (IDriveRequestBuilder). If you have multiple drives, you can also use _client.Drives, which is an IDrivesCollectionRequestBuilder.
- _client.Drive.Root: Builds a request to access the OneDrive's Root folder (IItemRequestBuilder).
- _client.Drive.Root.Children: Builds a request to get the root folder's children (IChildrenCollectionRequestBuilder). After the request is executed (with Request().GetAsync()), the result is an IChildrenCollectionPage, which contains a property named NextPageRequest. If the number of children was too large, the NextPageRequest property can be used to access the next page of items.
- _client.Drive.Root.ItemWithPath("Public/Test"): Builds a request to get the item at the path Public/Test within the root folder (IItemRequestBuilder).
- _client.Drive.Root.ItemWithPath("Public/Test/MyFile.txt").Content: Build a request to get the content of the file called MyFile.txt (IItemContentRequestBuilder).
- _client.Drive.Special.AppRoot: Builds a request to access the Application folder (IItemRequestBuilder).
- _client.Drive.Items[SomeId]: Builds a request to access an item by ID.

Each of these request builders can be seen in action in the sample illustrating this article.

Data entry errors. Consolidation headaches.
Quality data shouldn't be a pain.

Try a little Vitamin Q.

We supply Vitamin Q – "quality" – in data quality management solutions that profile, clean, enrich, and match your customer data – and keep it healthy over time. Add Vitamin Q to your data integration, business intelligence, Big Data, and CRM initiatives to ensure accurate data for greater insight and business value.

ACT NOW

Get a FREE 30-day trial!
www.MelissaData.com/vitaminQ

Solutions for 240+ Countries	10,000+ Customers Worldwide	30+ Years Strong	Data Quality & Mailing Solutions	Cloud • On-Premise • Services
		1985		

Germany
www.MelissaData.de

United Kingdom
www.MelissaData.co.uk

India
www.MelissaData.in

Australia
www.MelissaData.com.au

MELISSA DATA®
Your Partner in Global Data Quality
www.MelissaData.com | 1-800-MELISSA

Figure 3 **Selecting and Uploading a File's Content**

```
var picker = new FileOpenPicker
{
  SuggestedStartLocation = PickerLocationId.DocumentsLibrary
};

picker.FileTypeFilter.Add("*");
var file = await picker.PickSingleFileAsync();

using (var stream = await file.OpenStreamForReadAsync())
{
  var item = await _client.Drive.Special.AppRoot
    .ItemWithPath(file.Name)
    .Content.Request()
    .PutAsync<Item>(stream);

  // Save for the GetLink demo
  _savedId = item.Id;
}
```

Uploading a File

Uploading a file's content happens with a PUT request according to the principles of REST APIs. Apart from this difference, the mechanism to build a PUT request is very similar to the GET requests used earlier. In fact, most of the work needed is to actually acquire the Stream. For example, in a UWP app, this can be done with a FileOpenPicker loading the selected file from the Windows file system. The code in **Figure 3** shows a simple example (without error handling) uploading the selected file to the Application folder. A more complete example can be found in the code download. In this example, you access the meta information returned by the PutAsync method and save the item's ID so that you can easily access the saved item later; for instance, to get a link to this item.

Getting a Sharing Link

Once a request for an item has been created, a unique link to this item can be obtained from the OneDrive client with the CreateLink method that returns an IItemCreateLinkRequestBuilder. The following code shows how to do that using the _savedId that you saved earlier when you uploaded a file to OneDrive:

```
link = await _client.Drive
  .Items[_savedId]
  .CreateLink("view")
  .Request().PostAsync();
```

Of course, the CreateLink method can be called on any item request (for example, to get a link to a folder and so on). Note that the request created by the CreateLink method needs to be POSTed to the service. The CreateLink method requires one parameter that can be "view" or "edit." Depending on the value of this parameter, a read-only link or a read-write link will be created.

Figure 4 **Copying the Newly Uploaded Item to the Root**

```
var newLocation = await _client.Drive.Root.Request().GetAsync();

// Get the file to access its meta info
var file = await _client.Drive.Items[_savedId].Request().GetAsync();
var newName = Path.GetFileNameWithoutExtension(file.Name)
  + "-"
  + DateTime.Now.Ticks
  + Path.GetExtension(file.Name);

var itemStatus = await _client
  .Drive
  .Items[_savedId]
  .Copy(
    newName,
    new ItemReference
    {
      Id = newLocation.Id
    })
  .Request()
  .PostAsync();

var newItem = await itemStatus.CompleteOperationAsync(
  null,
  CancellationToken.None);

var successDialog = new MessageDialog(
  $"The item has been copied with ID {newItem.Id}",
  "Done!");
await successDialog.ShowAsync();
```

Updating an Item

Updating an item's metainformation happens with a request builder and the UpdateAsync method. For example, after you upload a file as shown earlier in this article, you can use its ID (which you saved in the _savedId attribute) to modify its name with the code shown here:

```
var updateItem = new Item
{
  Name = "[NewNameHere]"
};

var itemWithUpdates = await _client
  .Drive
  .Items[_savedId]
  .Request()
  .UpdateAsync(updateItem);
```

Moving an Item

Moving an item to a new location is a special case of updating its properties. In this case, you'll update its ParentReference property, which contains information about the item's location in OneDrive. If you modify this property with a new location and update the OneDrive item accordingly, the item will be moved.

In the following sample, you take the file that you uploaded earlier (having saved its ID in the _savedId attribute) and move it from its original location into the Root folder (of course in a real life application, some error handling needs to be added to this code):

```
var newLocation = await _client.Drive.Root.Request().GetAsync();

var updateItem = new Item
{
  ParentReference = new ItemReference
  {
    Id = newLocation.Id
  }
};

var itemWithUpdates = await _client
  .Drive
  .Items[_savedId]
  .Request()
  .UpdateAsync(updateItem);
```

Copying an Item

Copying an item is a little different than moving it. Interestingly, when you move or rename an item, the result of the asynchronous operation comes immediately, and is the Item instance with the new metainformation (such as the new location or the new name). When you copy an item, however, this can take a while and instead of waiting until the operation is completed, the SDK returns an instance of IItemCopyAsyncMonitor as soon as the copy operation begins.

This instance has one method called CompleteOperationAsync, which polls the result of the copy operation on OneDrive, updates an optional progress provider and returns the Item instance only when the copy operation is completed. This provides a very nice UX because it's possible to do another operation at the same time and notify the user when the copy operation is finished. Of course, just as with every long-lasting operation, it's possible to cancel the polling (however, this will not cancel the copy operation itself!). The code in **Figure 4** shows how the item that was uploaded to the Application folder can be copied to the Root folder. First, you retrieve the Root folder itself. Then, you get the file in order to construct a new unique name (to avoid collisions). Finally, the Copy request is created and executed by a POST to the server. Note how you then use the CompleteOperationAsync method to wait and notify the user when the copy operation is done.

Creating a New Folder

There are more operations possible in the SDK, but before I move on, I want to mention another interesting feature: creating a new folder. This is interesting because it acts on a collection (the parent folder's Children) by adding an item and then sending the request to OneDrive as shown in **Figure 5**. Note that in this case the method used (AddAsync) doesn't directly correspond to an HTTP method (GET, POST, PUT and so on). Once the folder is created, the newFolderCreated variable contains the necessary information, especially the folder's ID:

Signing Off and More

Finally, once the work with the client is finished, the user can choose to sign off. This is easy to do by using the OneDrive client's SignOutAsync method.

In addition to the methods and properties described in this article, there are a few more functionalities in the SDK. To make sure to get the latest and greatest documentation, you can check two different documentation sources:

- The OneDrive C# SDK on GitHub has quite a lot of documentation available. You can find it at bit.ly/1k0V2AL.
- The OneDrive API itself is documented at bit.ly/1QniW84.

Error Handling

If anything wrong happens with the service call, a OneDriveException will be thrown. However, the exception message doesn't contain information about the actual error. Instead, the error detail is contained in the OneDriveException's Error property (of type Microsoft.OneDrive.Sdk.Error). This is where you'll find the error message, as well as additional data to help you solve the issue.

Because errors can be nested, you can easily use the IsMatch method to look for a specific error code anywhere in the error hierarchy, for example with:

```
theException.IsMatch(OneDriveErrorCode.InvalidRequest.ToString());
```

Figure 5 **Creating a New Folder**

```
var newFolder = new Item
{
  Name = NewFolderNameText.Text,
  Folder = new Folder()
};

var newFolderCreated = _client.Drive
  .Special.AppRoot
  .Children
  .Request()
  .AddAsync(newFolder);

var successDialog = new MessageDialog(
  $"The folder has been created with ID {newFolderCreated.Id}",
  "Done!");
await successDialog.ShowAsync();
```

Getting and Building the SDK Source

While the SDK can be added using the NuGet Package Manager, it can be useful to get the source code; for example, to make changes or add features to the code. This can be done easily, either by downloading the source code, or (even better) by forking the source code from GitHub and modifying your branch.

The OneDrive SDK source code is available at bit.ly/1WX0Y03. In order to get the code and create a fork, you can use your favorite GitHub client, such as GitHub Desktop (desktop.github.com). Once you get the code on your local machine, you can build it in Visual Studio and add it to your application as a project reference, for example. From this point forward, you're at the same point as after adding the SDK through NuGet.

Wrapping Up

In the first article of this series, you saw how you can build a powerful library calling into a REST API by using the HttpClient, the async/await keywords and a JSON serializer. However, while these days this is much easier than it used to be, it's still a lot of work, especially when you want to keep your library up-to-date with new features.

In this article, I took the other approach, which is to use the OneDrive SDK built by Microsoft. You saw how the library can be added to a UWP application, how the authentication works (either by using the logged-in user's credentials transparently or by using the WebAuthenticationBroker and offering to the user the possibility to select a different account and so on). I also demonstrated the most useful item operations such as getting an item information by its path or ID, getting a folder's children, downloading, renaming, moving or copying a file, getting a unique share link to an item, and more.

With the OneDrive SDK available on GitHub in open source and Microsoft actively developing new features and fixing issues, it's easier than ever to access OneDrive from your UWP apps. This opens the door to a range of features such as roaming settings, sharing documents between devices and more. ■

Laurent Bugnion *is senior director for IdentityMine, one of the leading companies (and a gold partner) for Microsoft technologies. He is based in Zurich, Switzerland. His 2010 book, "Silverlight 4 Unleashed," published by Sams, is an advanced sequel to "Silverlight 2 Unleashed" (2008). He writes for several publications, is in his ninth year as a Microsoft MVP and is in his second year as a Microsoft Regional Director. He's the author of the well-known open source framework MVVM Light for Windows, WPF, Xamarin and of the popular Pluralsight reference course about MVVM Light. Reach him on his blog at galasoft.ch.*

Thanks to the following Microsoft technical expert for reviewing this article: *Gina Chernoby*

Progressive Enhancement with ASP.NET and React

Graham Mendick

Delivering Web content reliably is a game of chance. The random elements are the user's network speed and browser capabilities. Progressive enhancement, or PE, is a development technique that embraces this unpredictability. The cornerstone of PE is server-side rendering of the HTML. It's the only way to maximize your chances of success in the game of content delivery. For users with a modern browser, you layer on JavaScript to enhance the experience.

With the advent of data-binding libraries like AngularJS and Knockout, the Single-Page Application (SPA) came into its own. The SPA is the antithesis of PE because, by client rendering the HTML, it ignores the unpredictable nature of the Web. Visitors on slow networks face long loading times while users and search engines with less-capable browsers might not receive any content at all. But even these concerns haven't blunted the appeal of the SPA.

This article discusses:

- Progressively enhanced applications vs. Single-Page Applications
- Rendering and filtering on the server
- Rendering and filtering on the client
- Checking for HTML5 History support

Technologies discussed:

ASP.NET, React, JavaScript, HTML, C#, Node, Babel, Navigation, Gulp, Edge

Code download available at:

msdn.com/magazine/0216magcode

The SPA killed progressive enhancement. It was just too hard to turn a server-rendered application into a SPA through JavaScript enhancement.

Luckily, it turns out that PE isn't really dead. It's merely been sleeping, and a JavaScript library called React has just come along and woken it up. React provides the best of both worlds because it can run on the server and on the client. You can start with a server-rendered application and, at the flick of a switch, bring it to life as a client-rendered one.

The TodoMVC project (todomvc.com) offers the same Todo SPA built with different JavaScript data-binding libraries to help you decide which to choose. It's a great project, but the implementations suffer from being client-rendered only. In this article, I'll put this right by building a cut-down version as a progressively enhanced SPA using React and ASP.NET. I'll concentrate on the read-only functionality, so you'll be able to view your list of todos and filter them to show the active or completed ones.

Rendering on the Server

With the old approach to PE, I'd build an ASP.NET MVC application using Razor to render the todo list on the server. If I decided to enhance it into a SPA, I'd be back at square one—I'd have to re-implement the rendering logic in JavaScript. With my new approach to PE, I'll build an ASP.NET MVC application using React instead of Razor to render the todo list on the server. This way, it can double as the client-rendering code.

I'll start by creating a new ASP.NET MVC project called TodoMVC. Aside from the View layer, the code is unremarkable, so the Models folder holds a TodoRepository that returns an IEnumerable of todos, and inside the HomeController is an Index method that calls into the repository. From that point on, things start to look a bit different. Instead of passing the todo list to the Razor view, I'll pass it to React to produce the HTML on the server.

To run JavaScript on the server you need Node.js, which you can download from nodejs.org. Node.js comes with its own package manager called npm. I'll install React using npm just as I'd use NuGet to install a .NET package. I'll open a command prompt, cd into my TodoMVC project folder and run the "npm install react" command.

Next, I'll create a file called app.jsx in the Scripts folder (I'll explain the .jsx file extension shortly). This file will hold the React rendering logic, taking the place of the Razor view in a typical ASP.NET MVC project. Node.js uses a module-loading system so, to load the React module, I'll add a require statement at the start of app.jsx:

```
var React = require('react');
```

A React UI is made up of components. Each component has a render function that turns input data into HTML. The input data is passed in as properties. Inside app.jsx, I'll create a List component that takes in the todos and outputs them as an unordered list, with the title of each todo represented as a list item:

```
var List = React.createClass({
  render: function () {
    var todos = this.props.todos.map(function (todo) {
      return <li key={todo.Id}>{todo.Title}</li>;
    });
    return <ul>{todos}</ul>;
  }
});
```

The file has a .jsx extension because the React code is a mixture of JavaScript and an HTML-like syntax called JSX. I want to run this code on the server, but Node.js doesn't understand JSX so I must first convert the file into JavaScript. Converting JSX to JavaScript is known as transpiling, and an example transpiler is Babel. I could paste my app.jsx contents into the online Babel transpiler (babeljs.io/repl) and create an app.js file from the transpiled output. But it makes more sense to automate this step because app.jsx could change fairly often.

I'll use Gulp to automate the conversion of app.jsx into app.js. Gulp is a JavaScript task runner that comes with a variety of plug-ins to help you transform source files. Later on, I'll write a Gulp task that bundles up the JavaScript for the browser. For now, I need a task that passes app.jsx through the Babel transpiler so it can be used inside Node.js on the server. I'll install Gulp and the Babel plug-in from npm by running:

```
npm install gulp gulp-babel babel-preset-react
```

As you can see, by separating the package names with spaces, I can install multiple packages with a single command. I'll create a gulpfile.js inside the TodoMVC project folder and add the transpile task to it:

```
var babel = require('gulp-babel');

gulp.task('transpile', function(){
  return gulp.src('Scripts/app.jsx')
    .pipe(babel({ presets: ['react'] }))
    .pipe(gulp.dest('Scripts/'))
});
```

The task is made up of three steps. First, Gulp receives the app.jsx source file. Then, the file is piped through the Babel transpiler. Last, the output app.js file is saved to the Scripts folder. To make the task runnable, I'll use Notepad to create a package.json file in the TodoMVC project folder with a scripts entry pointing at it:

```
{
  "scripts": {
    "transpile": "gulp transpile"
  }
}
```

From the command line I'll run the transpile task using "npm run transpile." This generates an app.js file that can run inside Node.js because the JSX has been replaced with JavaScript.

Because I'm using React as the view layer, I want to pass the todos from the controller into the List component and have the HTML returned. In Node.js, the code inside app.js is private and can only be made public by explicitly exporting it. I'll export a getList function from app.jsx so the List component can be created externally, remembering to run the transpile task so that app.js is updated:

```
function getList(todos) {
  return <List todos={todos} />;
}
exports.getList = getList;
```

The HomeController is in C# and the getList function is in JavaScript. To call across this boundary, I'll use Edge.js (tjanczuk.github.io/edge), which is available from NuGet by running Install-Package Edge.js. Edge.js expects you to pass it a C# string containing Node.js code that returns a function with two parameters. The first parameter holds the data passed from the C# and the second parameter is a callback used to return the JavaScript data back to the C#. After running "npm install react-dom" to bring in React's server-rendering capability, I'll use Edge.js to create a function that returns the List component's HTML from the todos array passed in:

```
private static Func<object, Task<object>> render = Edge.Func(@"
  var app = require('../../Scripts/app.js');
  var ReactDOMServer = require('react-dom/server');

  return function (todos, callback) {
    var list = app.getList(todos);
    callback(null, ReactDOMServer.renderToString(list));
  }
");
```

From the Node.js code, Edge.js creates a C# Func, which I assign to a variable called "render" in the HomeController. Calling render with a list of todos will return the HTML. I'll add this call into the Index method, using the async/await pattern because calls into Edge.js are asynchronous:

```
public async Task<ActionResult> Index()
{
  var todos = new TodoRepository().Todos.ToList();
  ViewBag.List = (string) await render(todos);
  return View();
}
```

I added the HTML returned to the dynamic ViewBag so I can access it from the Razor view. Even though React is doing all the work, I still need one line of Razor to send the HTML to the browser and complete the server rendering:

```
<div id="content">@Html.Raw(ViewBag.List)</div>
```

This new approach to progressive enhancement might seem like more work compared to the old approach. But don't forget, with this new approach, the server-rendering code will become the client-rendering code. There won't be the duplicated effort required by the old approach when it comes to turning the server-rendered application into a SPA.

Filtering on the Server

The todos must be filterable so that either the active or completed ones can be displayed. Filtering means hyperlinks and hyperlinks mean routing. I've just substituted Razor for React, a JavaScript renderer that works on both client and server. Next, I'm going to apply the same treatment to routing. Rather than use the routing solution that ships with ASP.NET MVC, I'm going to replace it with the Navigation router (grahammendick.github.io/navigation), a JavaScript router that works on both client and server.

I'll run "npm install navigation" to bring in the router. You can think of the Navigation router as a state machine, where each state represents a different view within your application. In app.jsx, I'll configure the router with a state representing the todo "list" view. I'll assign that state a route with an optional "filter" parameter so that the filtering URLs look like "/active" and "/completed":

```
var Navigation = require('navigation');

var config = [
  { key: 'todoMVC', initial: 'list', states: [
    { key: 'list', route: '{filter?}' }]
  }
];
Navigation.StateInfoConfig.build(config);
```

With the old approach to PE, you'd put the filtering logic inside the controller. With the new approach, the filtering logic lives inside the React code so it can be reused on the client when I turn it into a SPA. The List component will take in the filter and check it against a todo's completed status to determine the list items to display:

```
var filter = this.props.filter;
var todoFilter = function(todo){
  return !filter || (filter === 'active' && !todo.Completed)
    || (filter === 'completed' && todo.Completed);
}
var todos = this.props.todos.filter(todoFilter).map(function(todo) {
  return <li key={todo.Id}>{todo.Title}</li>;
});
```

I'll change the HTML returned from the List component to include the filter hyperlinks below the filtered todo list:

```
<div>
  <ul>{todos}</ul>
  <ul>
    <li><a href="/">All</a></li>
    <li><a href="/active">Active</a></li>
    <li><a href="/completed">Completed</a></li>
  </ul>
</div>
```

The exported "getList" function needs an additional parameter so it can pass the new filter property into the List component. This is the last change to app.jsx to support filtering, so it's a good time to rerun the Gulp transpile task to generate a fresh app.js.

```
function getList(todos, filter) {
  return <List todos={todos} filter={filter} />;
}
```

The selected filter must be extracted from the URL. You might be tempted to register an ASP.NET MVC route so that the filter is passed into the controller. But this would duplicate the route already configured in the Navigation router. Instead, I'll use the Navigation router to extract the filter parameter. First, I'll remove all mention of route parameters from the C# RouteConfig class.

```
routes.MapRoute(
  name: "Default",
   url: "{*url}",
  defaults: new { controller = "Home", action = "Index" }
);
```

The Navigation router has a navigateLink function for parsing URLs. You hand it a URL and it stores the extracted data in a StateContext object. You can then access this data using the name of the route parameter as the key:

```
Navigation.StateController.navigateLink('/completed');
var filter = Navigation.StateContext.data.filter;
```

I'll plug this route parameter extraction code into the render Edge.js function so the filter can be retrieved from the current URL and passed into the getList function. But the JavaScript on the server can't access the URL of the current request, so it'll have to be passed in from the C#, along with the todos, via the function's first parameter:

```
return function (data, callback) {
  Navigation.StateController.navigateLink(data.Url);
  var filter = Navigation.StateContext.data.filter;
  var list = app.getList(data.Todos, filter);
  callback(null, ReactDOMServer.renderToString(list));
}
```

The corresponding change to the Index method of the HomeController is to pass an object into the render call that holds both the URL from the server-side request and the todo list.

```
var data = new {
  Url = Request.Url.PathAndQuery,
  Todos = todos
};
ViewBag.List = (string) await render(data);
```

With the filtering in place, the server-side phase of the build is complete. Starting with server-rendering guarantees, the todo list is viewable by all browsers and search engines. The plan is to enhance the experience for modern browsers by filtering the todo list on the client. The Navigation router will manage the browser history and ensure that a client-filtered todo list remains bookmarkable.

Rendering on the Client

If I'd built the UI with Razor, I'd be no closer now to the SPA finishing line than when I set out. Having to replicate the rendering logic in JavaScript is why old school PE fell out of favor. But, with React, it's quite the opposite because I can reuse all my app.js code on the client. Just as I used React to render the List component to HTML on the server, I'll use it to render that same component to the DOM on the client.

To render the List component on the client I need access to the todos. I'll make the todos available by sending them down in a JavaScript variable as part of the server render. By adding the todo list to the ViewBag in the HomeController, I can serialize them to a JavaScript array inside the Razor view:

```
<script>
  var todos = @Html.Raw(new JavaScriptSerializer().Serialize(ViewBag.Todos));
</script>
```

I'll create a client.js file inside the Scripts folder to hold the client rendering logic. This code will look the same as the Node.js code I passed into Edge.js to handle the server rendering, but tweaked to cater to the differences in environment. So, the URL is sourced from the browser's location object, rather than the server-side request, and React renders the List component into the content div, rather than to an HTML string:

```
var app = require('./app.js');
var ReactDOM = require('react-dom');
var Navigation = require('navigation');

Navigation.StateController.navigateLink(location.pathname);
var filter = Navigation.StateContext.data.filter;
var list = app.getList(todos, filter);
ReactDOM.render(list, document.getElementById('content'));
```

I'll add a line to app.jsx that tells the Navigation router I'm using HTML5 History rather than the hash history default. If I didn't do this, the navigateLink function would think that the URL had changed and update the browser hash to match:

```
Navigation.settings.historyManager = new Navigation.HTML5HistoryManager();
```

PRECISELY
PROGRAMMED
FOR SPEED

DynamicPDF—Comprehensive PDF Solutions for .NET Developers

ceTe Software's DynamicPDF products provide real-time PDF generation, manipulation, conversion, printing, viewing, and much more. Providing the best of both worlds, the object models are extremely flexible but still supply the rich features you need as a developer. Reliable and efficient, the high-performance software is easy to learn and use. If you do encounter a question with any of our components, simply contact ceTe Software's readily available, industry-leading support team.

DynamicPDF®
WWW.DYNAMICPDF.COM

TRY OUR PDF SOLUTIONS FREE TODAY!
www.DynamicPDF.com/eval or call 800.631.5006 | +1 410.772.8620

ceTesoftware

If I could add a client.js script reference directly to the Razor view, that would be the end of the changes needed for client rendering. Unfortunately, it's not quite that simple, because the require statements inside client.js are part of the Node.js module-loading system and aren't recognized by browsers. I'll use a Gulp plug-in called browserify to create a task that bundles client.js and all its required modules into a single JavaScript file, which I can then add to the Razor view. I'll run "npm install browserify vinyl-source-stream" to bring in the plug-in:

```
var browserify = require('browserify');
var source = require('vinyl-source-stream');

gulp.task('bundle', ['transpile'], function(){
  return browserify('Scripts/client.js')
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest('Scripts/'))
});
```

I don't want the bundle task to run unless it includes the latest changes to app.jsx. To ensure that the transpile task always runs first, I made it a dependency of the bundle task. You can see that the second parameter of a Gulp task lists its dependencies. I'll add an entry to the scripts section of package.json for the bundle task. Running the command "npm run bundle" will create bundle.js and I'll add a script reference pointing at it to the bottom of the Razor view:

```
<script src="~/Scripts/bundle.js"></script>
```

By server-rendering the HTML, I've built an application that starts faster than those at todomvc.com because they can't display any content until their JavaScript loads and executes. Similarly, once the JavaScript loads in my application, a client render runs. In contrast, this doesn't update the DOM at all, but allows React to attach to the server-rendered content so that subsequent todo list filtering can be handled on the client.

Filtering on the Client

If you were doing PE the old-fashioned way, you might implement filtering on the client by toggling class names to control the todo items' visibility. But without a JavaScript router to help out, it's all too easy to break browser history. If you neglect to update the URL, the filtered list won't be bookmarkable. By doing PE the modern way, I already have the Navigation router up and running on the client to keep the browser history intact.

To update the URL when a filter hyperlink is clicked, I need to intercept the click event and pass the hyperlink's href into the router's navigateLink function. There's a React plug-in for the Navigation router that will handle this for me, provided I build the hyperlinks in the prescribed way. For example, instead of writing <a href="/active">Active</a>, I must use the RefreshLink React component the plug-in provides:

```
var RefreshLink = require('navigation-react').RefreshLink;
<RefreshLink toData={{filter: 'active'}}>Active</RefreshLink>
```

After running "npm install navigation-react" to bring in the plug-in, I'll update the List component in app.jsx by replacing the three filter hyperlinks with their RefreshLink equivalents.

To keep the UI and URL in sync, I have to filter the todo list whenever the URL changes—not only when a filter hyperlink is clicked but also when the browser back button is pressed. Instead of adding separate event listeners, I can add a single listener to the Navigation router that will be called any time navigation happens. This navigation listener must be attached to the "list" state I created as part of the router configuration. First, I'll access this state from the Navigation router using the keys from the configuration:

```
var todoMVC = Navigation.StateInfoConfig.dialogs.todoMVC;
var listState = todoMVC.states.list;
```

A navigation listener is a function assigned to the state's "navigated" property. Whenever the URL changes, the Navigation router will call this function and pass in the data extracted from the URL. I'll replace the code in client.js with a navigation listener that re-renders the List component into the "content" div using the new filter. React will take care of the rest, updating the DOM to display the freshly filtered todos:

```
listState.navigated = function(data){
  var list = app.getList(todos, data.filter);
  ReactDOM.render(list, document.getElementById('content'));
}
```

In implementing the filtering, I accidentally removed the code from client.js that triggered the initial client render. I'll restore this functionality by adding a call to "Navigation.start" at the bottom of client.js. This effectively passes the current browser URL into the router's navigateLink function, which triggers the navigation listener and performs the client rendering. I'll rerun the bundle task to bring the latest changes into app.js and bundle.js.

The new approach to PE is modern day alchemy. It turns the base metal of a server-rendered application into SPA gold. But it takes a special kind of base metal for the transformation to work, one that's built from JavaScript libraries that run equally well on the server and in the browser: React and the Navigation router in the place of Razor and ASP.NET MVC routing. This is the new chemistry for the Web.

Cutting the Mustard

The aim of PE is an application that works in all browsers, while offering an improved experience for modern browsers. But, in building this improved experience, I've stopped the todo list from working in older browsers. The SPA conversion relies on the HTML5 History API, which Internet Explorer 9, for example, doesn't support.

PE isn't about offering the same experience to all browsers. The todo list doesn't have to be a SPA in Internet Explorer 9. In browsers that don't support HTML5 History, I can fall back to the server-rendered application. I'll change the Razor view to dynamically load bundle.js, so it's only sent to browsers that support HTML5 History:

```
if (window.history && window.history.pushState) {
  var script = document.createElement('script');
  script.src = "/Scripts/bundle.js";
  document.body.appendChild(script);
}
```

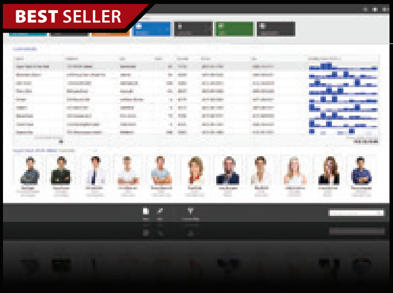
This check is called "cutting the mustard" because only those browsers that meet the requirements are considered worthy of receiving the JavaScript. The end result is the Web equivalent of the optical illusion where the same picture can either look like a rabbit or a duck. Take a look at the todo list through a modern browser and it's a SPA, but squint at it using an old browser and it's a traditional client-server application. ■

Graham Mendick *believes in a Web accessible to all and is excited by the new possibilities for progressive enhancement that isomorphic JavaScript has opened up. He's the author of the Navigation JavaScript router, which he hopes will help people to go isomorphic. Get in touch with him on Twitter: @grahammendick.*

Thanks to the following Microsoft technical expert for reviewing this article: *Steve Sanderson*

ComponentSource® /update/2016/02

Experience the brand new look at **www.componentsource.com**

BEST SELLER


DevExpress DXperience 15.2 | from $1,439.99

DevExpress

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

- WinForms Grid: New data-aware Tile View
- WinForms Grid & TreeList: New Excel-inspired Conditional Formatting
- .NET Spreadsheet: Grouping and Outline support
- ASP.NET: New Rich Text Editor-Word Processing control
- ASP.NET Reporting: New Web Report Designer

BEST SELLER

Help & Manual Professional | from $586.04

ec software

Help and documentation for .NET and mobile applications.

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

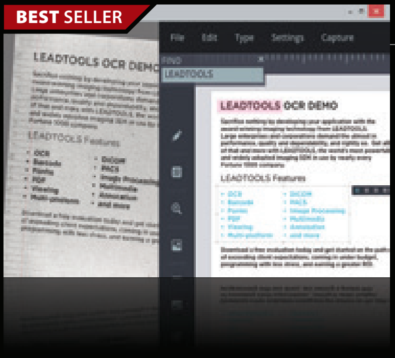
BEST SELLER

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

ASPOSE Your File Format APIs

Every Aspose .NET component in one package.

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

BEST SELLER


LEADTOOLS Document Imaging SDKs V19 | from $2,995.00 SRP

LEAD TECHNOLOGIES

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

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

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

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

European Headquarters
ComponentSource
2 New Century Place
East Street
Reading, Berkshire
RG1 4ET
United Kingdom

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

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

VISA DISCOVER
GSA Schedule Contract GS-35F-0188R

Customizable Scripting in C#

Vassili Kaplan

In this article I'll show you how to build a custom scripting language using C#—without using any external libraries. The scripting language is based on the Split-And-Merge algorithm for parsing mathematical expressions in C# I presented in the October 2015 issue of *MSDN Magazine* (msdn.com/magazine/mt573716).

By using custom functions, I can extend the Split-And-Merge algorithm to parse not only a mathematical expression, but also to parse a customizable scripting language. The "standard" language control flow statements (if, else, while, continue, break and so on) can be added as custom functions, as can other typical scripting language functionality (OS commands, string manipulation, searching for files and so on).

This article discusses:

- Modifying the Split-And-Merge algorithm
- The Interpreter class
- Variables and arrays
- Control flow
- Functions
- Internationalization

Technologies discussed:

C#

Code download available at:

msdn.com/magazine/0216magcode

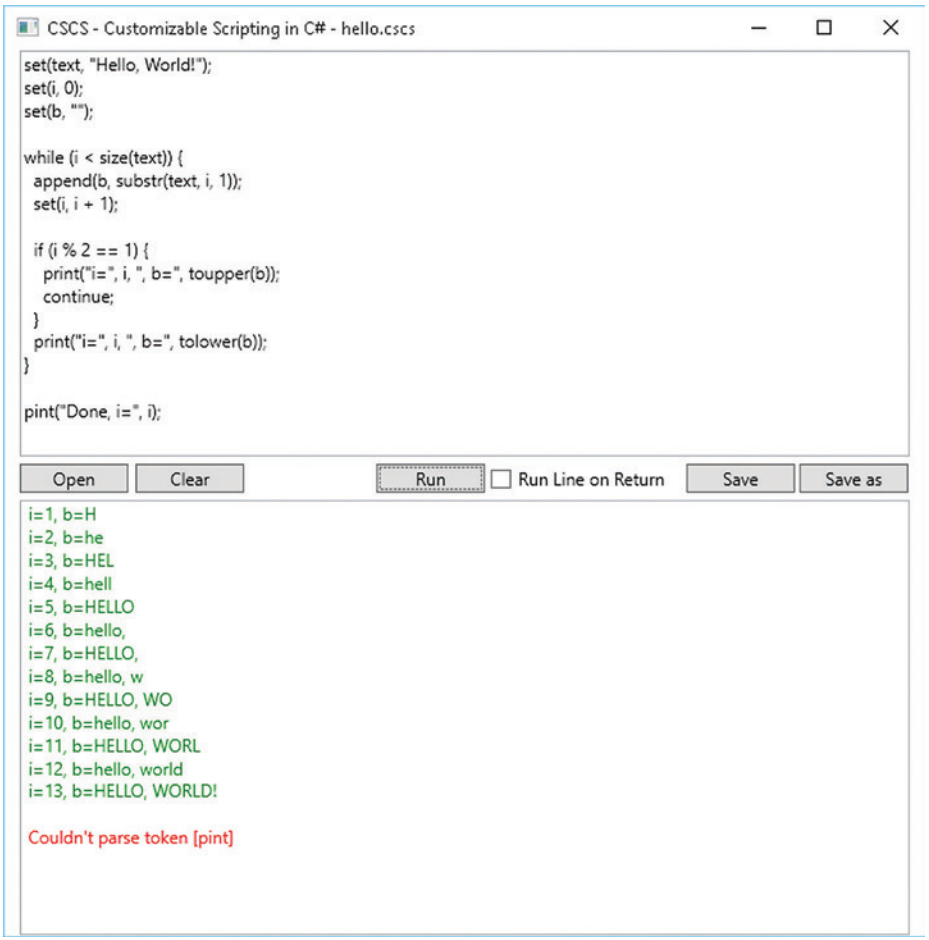
I'm going to call my language Customizable Scripting in C#, or CSCS. Why would I want to create yet another scripting language? Because it's an easy language to customize. Adding a new function or a new control flow statement that takes an arbitrary number of parameters takes just a few lines of code. Moreover, the function names and the control flow statements can be used in any non-English language scenario with just some configuration changes, which I'll also show in this article. And by seeing how the CSCS language is implemented, you'll be able to create your own custom scripting language.

By using custom functions, I can extend the Split-And-Merge algorithm to parse not only a mathematical expression, but also to parse a customizable scripting language.

The Scope of CSCS

It's fairly simple to implement a very basic scripting language, but brutally difficult to implement a five-star language. I'm going to limit the scope of CSCS here so you'll know what to expect:

- The CSCS language has if, else if, else, while, continue and break control flow statements. Nested statements are supported, as well. You'll learn how to add additional control statements on the fly.
- There are no Boolean values. Instead of writing "if (a)," you have to write "if (a == 1)."
- Logical operators aren't supported. Instead of writing "if (a ==1 and b == 2)," you write nested ifs: "if (a == 1) { if (b == 2) { … } }."
- Functions and methods aren't supported in CSCS, but they can be written in C# and registered with the Split-And-Merge Parser in order to be used with CSCS.
- Only "//"-style comments are supported.
- Variables and one-dimensional arrays are supported, all defined at the global level. A variable can hold a number, a string or a tuple (implemented as a list) of other variables. Multi-dimensional arrays are not supported.



```
set(text, "Hello, World!");
set(i, 0);
set(b, "");

while (i < size(text)) {
  append(b, substr(text, i, 1));
  set(i, i + 1);

  if (i % 2 == 1) {
    print("i=", i, ", b=", toupper(b));
    continue;
  }
  print("i=", i, ", b=", tolower(b));
}

pint("Done, i=", i);
```

```
i=1, b=H
i=2, b=he
i=3, b=HEL
i=4, b=hell
i=5, b=HELLO
i=6, b=hello,
i=7, b=HELLO,
i=8, b=hello, w
i=9, b=HELLO, WO
i=10, b=hello, wor
i=11, b=HELLO, WORL
i=12, b=hello, world
i=13, b=HELLO, WORLD!

Couldn't parse token [pint]
```

Figure 1 **"Hello, World!" in CSCS**

Figure 1 shows a "Hello, World!" program in CSCS. Due to a mistyping of "print," the program displays an error at the end: "Couldn't parse token [pint]." Note that all the previous statements executed successfully; that is, CSCS is an interpreter.

Modifications to the Split-And-Merge Algorithm

I've made two changes to the Split part of the Split-And-Merge algorithm. (The Merge part remains the same.)

The first change is that the result of parsing an expression can be now a number, a string or a tuple of values (each of which can be either a string or a number), rather than just a number. I created the following Parser.Result class to store the result of applying the Split-And-Merge algorithm:

```
public class Result
{
  public Result(double dRes = Double.NaN, string sRes = null, List<Result>
tRes = null)
  {
    Value  = dResult;
    String = sResult;
    Tuple  = tResult;
  }
  public double       Value  { get; set; }
  public string       String { get; set; }
  public List<Result> Tuple  { get; set; }
}
```

The second modification is that now the splitting part is performed not just until a stop-parsing character—) or \n—is found, but until any character in a passed array of stop-parsing characters is found. This is necessary, for example, when parsing the first argument of an If statement, where the separator can be any <, >, or = character.

You can take a look at the modified Split-And-Merge algorithm in the accompanying source code download.

The Interpreter

The class responsible for interpreting the CSCS code is called Interpreter. It's implemented as a singleton, that is, a class definition where there can be only one instance of the class. In its Init method, the Parser (see the original article mentioned earlier) is initialized with all the functions used by the Interpreter:

```
public void Init()
{
  ParserFunction.AddFunction(Constants.IF,       new IfStatement(this));
  ParserFunction.AddFunction(Constants.WHILE,    new WhileStatement(this));
  ParserFunction.AddFunction(Constants.CONTINUE, new ContinueStatement());
  ParserFunction.AddFunction(Constants.BREAK,    new BreakStatement());
  ParserFunction.AddFunction(Constants.SET,      new SetVarFunction());
...
}
```

In the Constants.cs file, the actual names used in CSCS are defined:

```
public const string IF          = "if";
public const string ELSE        = "else";
public const string ELSE_IF     = "elif";
public const string WHILE       = "while";
public const string CONTINUE    = "continue";
public const string BREAK       = "break";
public const string SET         = "set";
...
```

Any function registered with the Parser must be implemented as a class derived from the ParserFunction class and must override its Evaluate method.

The first thing the Interpreter does when starting to work on a script is to simplify the script by removing all white spaces (unless they're inside of a string), and all comments. Therefore, spaces or new lines can't be used as operator separators. The operator separator character and the comment string are defined in Constants.cs, as well:

```
public const char END_STATEMENT = ';';
public const string COMMENT     = "//";
```

Variables and Arrays

CSCS supports numbers (type double), strings or tuples (arrays of variables implemented as a C# list). Each element of a tuple can be either a string or a number, but not another tuple. Therefore, multidimensional arrays are not supported. To define a variable, the CSCS function "set" is used. The C# class SetVarFunction implements the functionality of setting a variable value, as shown in **Figure 2**.

Here are some examples of defining a variable in CSCS:

```
set(a, "2 + 3");  // a will be equal to the string "2 + 3"
set(b, 2 + 3);    // b will be equal to the number 5
set(c(2), "xyz"); // c will be initialized as a tuple of size 3 with c(0) = c(1) = ""
```

Figure 2 **Implementation of the Set Variable Function**

```
class SetVarFunction : ParserFunction
{
  protected override Parser.Result Evaluate(string data, ref int from)
  {
    string varName = Utils.GetToken(data, ref from, Constants.NEXT_ARG_ARRAY);
    if (from >= data.Length)
    {
      throw new ArgumentException("Couldn't set variable before end of line");
    }

    Parser.Result varValue = Utils.GetItem(data, ref from);

    // Check if the variable to be set has the form of x(i),
    // meaning that this is an array element.
    int arrayIndex = Utils.ExtractArrayElement(ref varName);
    if (arrayIndex >= 0)
    {
      bool exists = ParserFunction.FunctionExists(varName);
      Parser.Result  currentValue = exists ?
            ParserFunction.GetFunction(varName).GetValue(data, ref from) :
            new Parser.Result();

      List<Parser.Result> tuple = currentValue.Tuple == null ?
                                  new List<Parser.Result>() :
                                  currentValue.Tuple;
      if (tuple.Count > arrayIndex)
      {
        tuple[arrayIndex] = varValue;
      }
      else
      {
        for (int i = tuple.Count; i < arrayIndex; i++)
        {
          tuple.Add(new Parser.Result(Double.NaN, string.Empty));
        }
        tuple.Add(varValue);
      }

      varValue = new Parser.Result(Double.NaN, null, tuple);
    }

    ParserFunction.AddFunction(varName, new GetVarFunction(varName, varValue));

    return new Parser.Result(Double.NaN, varName);
  }
}
```

Note that there's no special declaration of an array: just defining a variable with an index will initialize the array if it's not already initialized, and add empty elements to it, if necessary. In the preceding example, the elements c(0) and c(1) were added, both initialized to empty strings. This eliminates, in my view, the unnecessary step that's required in most scripting languages of declaring an array first.

Note that there's no special declaration of an array: just defining a variable with an index will initialize the array if it's not already initialized, and add empty elements to it if necessary.

All CSCS variables and arrays are created using CSCS functions (like set or append). They're all defined with global scope and can be used later just by calling the variable name or a variable with an index. In C#, this is implemented in the GetVarFunction shown in **Figure 3**.

Only the set variable function must be registered with the Parser:

```
ParserFunction.AddFunction(Constants.SET, new SetVarFunction());
```

The get variable function is registered inside of the set variable function C# code (see the next-to-last statement in **Figure 2**):

```
ParserFunction.AddFunction(varName, new GetVarFunction(varName, varValue));
```

Figure 3 **Implementation of the Get Variable Function**

```
class GetVarFunction : ParserFunction
{
  internal GetVarFunction(Parser.Result value)
  {
    m_value = value;
  }

  protected override Parser.Result Evaluate(string data, ref int from)
  {
    // First check if this element is part of an array:
    if (from < data.Length && data[from - 1] == Constants.START_ARG)
    {
      // There is an index given - it may be for an element of the tuple.
      if (m_value.Tuple == null || m_value.Tuple.Count == 0)
      {
        throw new ArgumentException("No tuple exists for the index");
      }

      Parser.Result index = Utils.GetItem(data, ref from, true /* expectInt */);
      if (index.Value < 0 || index.Value >= m_value.Tuple.Count)
      {
        throw new ArgumentException("Incorrect index [" + index.Value +
          "] for tuple of size " + m_value.Tuple.Count);
      }

      return m_value.Tuple[(int)index.Value];
    }

    // This is the case for a simple variable, not an array:
    return m_value;
  }

  private Parser.Result m_value;
}
```

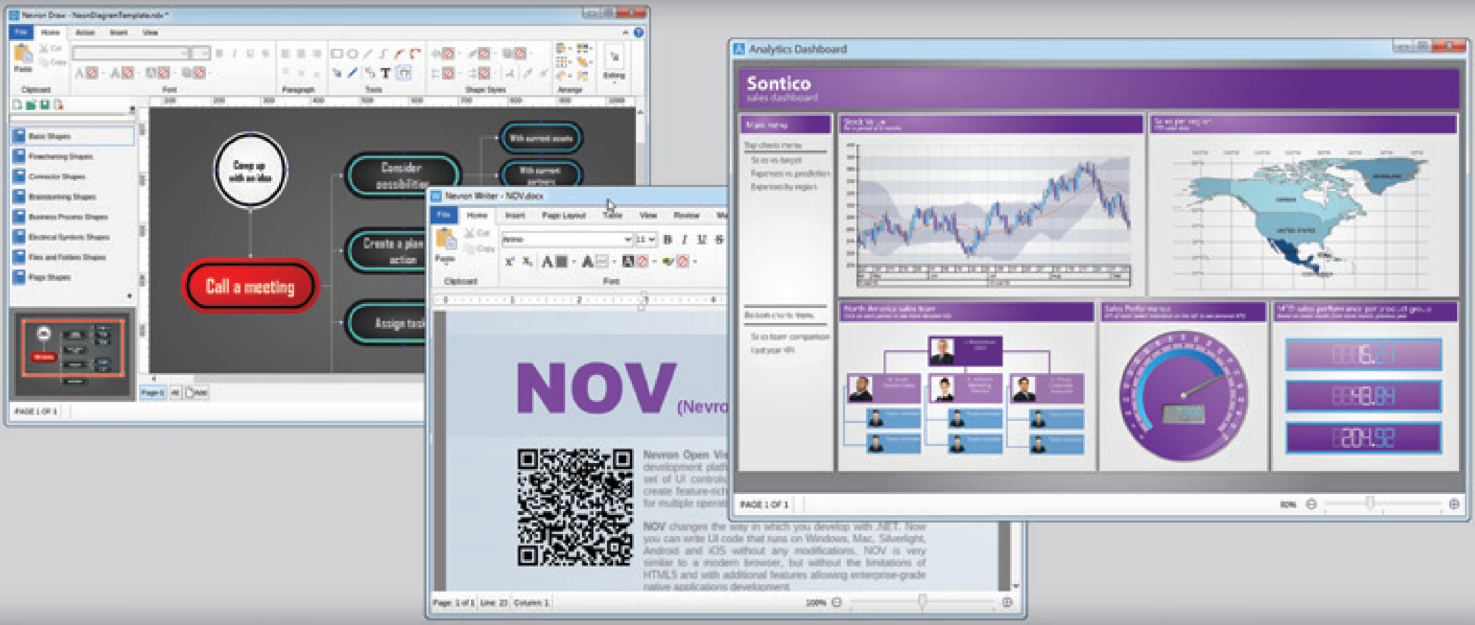
NEVRON www.nevron.com

NEVRON
OPEN VISION for .NET

Full support for VS2015

The Complete UI suite for .NET

free				new	new	new	new	soon
UI	RICH TEXT	BARCODE	GAUGE	DIAGRAM	MAP	CHART	SCHEDULER	GRID



NOV changes the way in which you develop with .NET. Now you can write UI code that runs on Windows, Mac, Silverlight, Android and iOS without any modifications. NOV is very similar to a modern browser, but without the limitations of HTML5 and with additional features allowing enterprise-grade native applications development.

develop cutting edge applications for WinForms, WPF, Silverlight, Xamarin.Mac & MonoMac using a single code base

DOWNLOAD FREE TRIAL

Learn more at **www.nevron.com** today

www.nevron.com | email@nevron.com | +1 888-201-6088 (Toll free, USA and Canada)

Visual Studio Partner

Figure 4 **Implementation of the If Statement**

```
internal Parser.Result ProcessIf()
{
  int startIfCondition = m_currentChar;
  Parser.Result result = null;

  Parser.Result arg1 = GetNextIfToken();
  string comparison  = Utils.GetComparison(m_data, ref m_currentChar);
  Parser.Result arg2 = GetNextIfToken();

  bool isTrue = EvalCondition(arg1, comparison, arg2);

  if (isTrue)
  {
    result = ProcessBlock();
    if (result is Continue || result is Break)
    {
      // Got here from the middle of the if-block. Skip it.
      m_currentChar = startIfCondition;
      SkipBlock();
    }
    SkipRestBlocks();

    return result;
  }

  // We are in Else. Skip everything in the If statement.
  SkipBlock();

  int endOfToken = m_currentChar;
  string nextToken = Utils.GetNextToken(m_data, ref endOfToken);

  if (ELSE_IF_LIST.Contains(nextToken))
  {
    m_currentChar = endOfToken + 1;
    result = ProcessIf();
  }
  else if (ELSE_LIST.Contains(nextToken))
  {
    m_currentChar = endOfToken + 1;
    result = ProcessBlock();
  }

  return result != null ? result : new Parser.Result();
}
```

Some examples of getting variables in CSCS are:

```
append(a, "+ 5"); // a will be equal to the string "2 + 3 + 5"
set(b, b * 2);    // b will be equal to the number 10 (if it was 5 before)
```

Control Flow: If, Else If, Else

The If, Else If and Else control flow statements are implemented internally as Parser functions, as well. They are registered by the Parser just like any other function:

```
ParserFunction.AddFunction(Constants.IF, new IfStatement(this));
```

Only the IF keyword must be registered with the Parser. ELSE_IF and ELSE statements will be processed inside of the IfStatement implementation:

```
class IfStatement : ParserFunction
{
  protected override Parser.Result Evaluate(string data, ref int from)
  {
    m_interpreter.CurrentChar = from;
    Parser.Result result = m_interpreter.ProcessIf();

    return result;
  }
  private Interpreter m_interpreter;
}
```

The real implementation of the If statement is in the Interpreter class, as shown in **Figure 4**.

It's explicitly stated that the If condition has the form: argument 1, comparison sign, argument 2:

```
Parser.Result arg1 = GetNextIfToken();
string comparison  = Utils.GetComparison(m_data, ref m_currentChar);
Parser.Result arg2 = GetNextIfToken();

bool isTrue = EvalCondition(arg1, comparison, arg2);
```

This is where optional AND, OR or NOT statements can be added.

Each element of a tuple can be either a string or a number, but not another tuple.

The EvalCondition function just compares the tokens according to the comparison sign:

```
internal bool EvalCondition(Parser.Result arg1, string comparison, Parser.Result arg2)
{
  bool compare = arg1.String != null ? CompareStrings(arg1.String,
comparison, arg2.String) :
                                       CompareNumbers(arg1.Value,
comparison, arg2.Value);

  return compare;
}
```

Here's the implementation of a numerical comparison:

```
internal bool CompareNumbers(double num1, string comparison, double num2)
{
  switch (comparison) {
    case "==": return num1 == num2;
    case "<>": return num1 != num2;
    case "<=": return num1 <= num2;
    case ">=": return num1 >= num2;
    case "<" : return num1 <  num2;
    case ">" : return num1 >  num2;
    default: throw new ArgumentException("Unknown comparison: " + comparison);
  }
}
```

The string comparison is similar and is available in the accompanying code download, as is the straightforward implementation of the GetNextIfToken function.

Figure 5 **Implementation of the ProcessBlock Method**

```
internal Parser.Result ProcessBlock()
{
  int blockStart = m_currentChar;
  Parser.Result result = null;

  while(true)
  {
    int endGroupRead = Utils.GoToNextStatement(m_data, ref m_currentChar);
    if (endGroupRead > 0)
    {
      return result != null ? result : new Parser.Result();
    }

    if (m_currentChar >= m_data.Length)
    {
      throw new ArgumentException("Couldn't process block [" +
                                  m_data.Substring(blockStart) + "]");
    }

    result = Parser.LoadAndCalculate(m_data, ref m_currentChar,
      Constants.END_PARSE_ARRAY);

    if (result is Continue || result is Break)
    {
      return result;
    }
  }
}
```

www.gnostice.com

Gnostice™
Smart needs...Smarter solutions...

One component suite for all Document requirements

WinForms, WPF, Web Forms, MVC, HTML5

Visual Studio MAGAZINE 2 READER'S 2015 CHOICE AWARD WINNER

View · Print · Edit · Convert · Digitize

PDF DOCX DOC RTF PPTX XLSX XLS TIFF MTIFF JPEG PNG BMP WMF EMF

Download your FREE 30-day trial today!!!

Gnostice XtremeDocumentStudio .NET enables you to develop rich and engaging user experiences. With XtremeDocumentStudio you can painlessly embed document viewing, printing, conversion, template-driven document creation, editing, mail-merge, digitization (OCR) and a host of other functionality in your applications. Format support includes PDF, DOCX, DOC, RTF, PPTX, XLSX, XLS, TIFF, MTIFF, JPEG, PNG, BMP, WMF & EMF. Platform support includes WinForms, WPF, Web Forms, MVC & HTML5. XtremeDocumentStudio .NET does not require external software or libraries such as Microsoft Word, Open XML SDK, Word Automation Services, Adobe PDF library or GhostScript.

XtremeDocumentStudio .NET

Figure 6 **Implementation of the While Loop**

```
internal void ProcessWhile()
{
  int startWhileCondition = m_currentChar;

  // A heuristic check against an infinite loop.
  int cycles = 0;
  int START_CHECK_INF_LOOP = CHECK_AFTER_LOOPS / 2;
  Parser.Result argCache1 = null;
  Parser.Result argCache2 = null;

  bool stillValid = true;
  while (stillValid)
  {
    m_currentChar = startWhileCondition;

    Parser.Result arg1 = GetNextIfToken();
    string comparison = Utils.GetComparison(m_data, ref m_currentChar);
    Parser.Result arg2 = GetNextIfToken();

    stillValid = EvalCondition(arg1, comparison, arg2);
    int startSkipOnBreakChar = m_currentChar;

    if (!stillValid)
    {
      break;
    }
    
    // Check for an infinite loop if same values are compared.
    if (++cycles % START_CHECK_INF_LOOP == 0)
    {
      if (cycles >= MAX_LOOPS || (arg1.IsEqual(argCache1) &&
        arg2.IsEqual(argCache2)))
      {
        throw new ArgumentException("Looks like an infinite loop after " +
          cycles + " cycles.");
      }
      argCache1 = arg1;
      argCache2 = arg2;
    }

    Parser.Result result = ProcessBlock();
    if (result is Break)
    {
      m_currentChar = startSkipOnBreakChar;
      break;
    }
  }

  // The while condition is not true anymore: must skip the whole while
  // block before continuing with next statements.
  SkipBlock();
}
```

When an if, else if, or else condition is true, all of the statements inside the block are processed. This is implemented in **Figure 5** in the ProcessBlock method. If the condition isn't true, all the statements are skipped. This is implemented in the SkipBlock method (see accompanying source code).

Note how the "Continue" and "Break" statements are used inside of the while loop. These statements are implemented as functions, as well. Here's Continue:

```
class Continue : Parser.Result  { }

class ContinueStatement : ParserFunction
{
  protected override Parser.Result
    Evaluate(string data, ref int from)
  {
    return new Continue();
  }
}
```

The implementation of the Break statement is analogous. They're both registered with the Parser like any other function:

```
ParserFunction.AddFunction(Constants.
CONTINUE,  new ContinueStatement());
ParserFunction.AddFunction(Constants.
BREAK,     new BreakStatement());
```

You can use the Break function to get out of nested If blocks or to get out of a while loop.

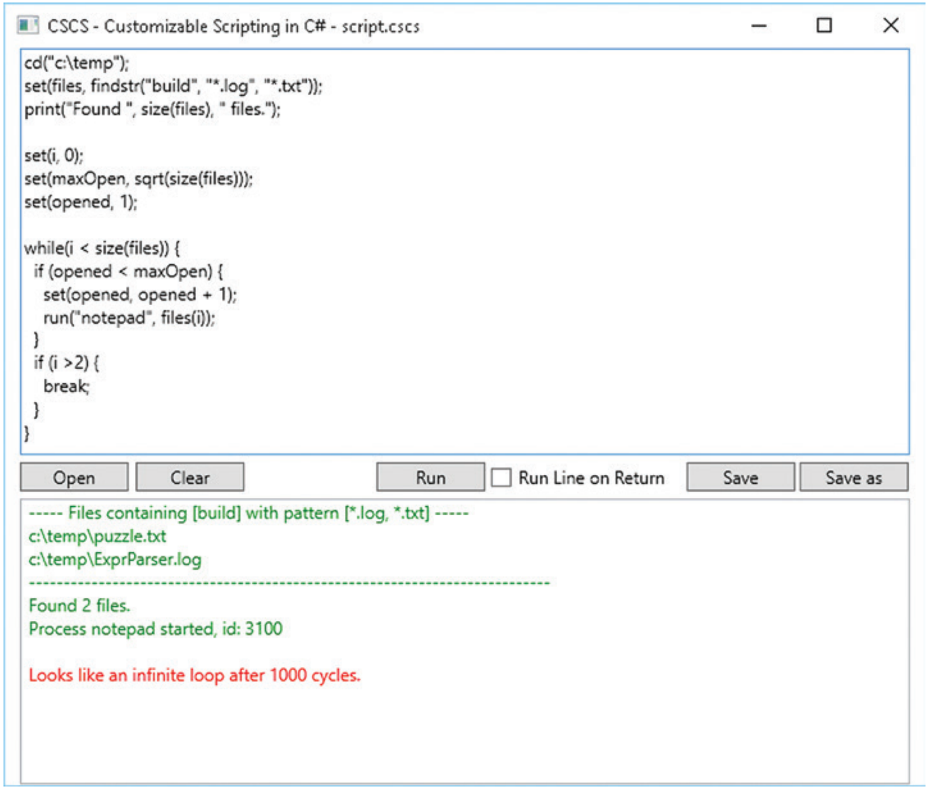
Control Flow: The While Loop

The while loop is also implemented and registered with the Parser as a function:

```
ParserFunction.AddFunction(Constants.
WHILE,     new WhileStatement(this));
```

Whenever the while keyword is parsed, the Evaluate method of the WhileStatement object is called:

```
class WhileStatement : ParserFunction
{
  protected override Parser.Result Evaluate(string data, ref int from)
  {
    string parsing = data.Substring(from);
    m_interpreter.CurrentChar = from;
    m_interpreter.ProcessWhile();

    return new Parser.Result();
  }
  private Interpreter m_interpreter;
}
```



```
cd("c:\temp");
set(files, findstr("build", "*.log", "*.txt"));
print("Found ", size(files), " files.");

set(i, 0);
set(maxOpen, sqrt(size(files)));
set(opened, 1);

while(i < size(files)) {
 if (opened < maxOpen) {
  set(opened, opened + 1);
  run("notepad", files(i));
 }
 if (i >2) {
  break;
 }
}
```

```
----- Files containing [build] with pattern [*.log, *.txt] -----
c:\temp\puzzle.txt
c:\temp\ExprParser.log
---------------------------------------------------------------
Found 2 files.
Process notepad started, id: 3100

Looks like an infinite loop after 1000 cycles.
```

Figure 7 **Detecting an Infinite While Loop in CSCS**

So the real implementation of the while loop is in the Interpreter class, as shown in **Figure 6**.

Note that the while loop proactively checks for an infinite loop after a certain number of iterations, defined in the configuration settings by the CHECK_AFTER_LOOPS constant. The heuristic is that if the exact same values in the while condition are compared over several loops, this could indicate an infinite loop. **Figure 7** shows a while loop where I forgot to increment the cycle variable ***i*** inside of the while loop.

Functions, Functions, Functions

In order for CSCS to do more useful things, more flesh needs to be added; that is, more functions must be implemented. Adding a new function to CSCS is straightforward: First implement a class deriving from the ParserFunction class (overriding the Evaluate method) and then register it with the Parser. Here's the implementation of the Print function:

```
class PrintFunction : ParserFunction
{
  protected override Parser.Result Evaluate(string data, ref int from)
  {
    List<string> args = Utils.GetFunctionArgs(data, ref from);
    m_interpreter.AppendOutput(string.Join("", args.ToArray()));

    return new Parser.Result();
  }

  private Interpreter m_interpreter;
}
```

The function prints any number of comma-separated arguments passed to it. The actual reading of the arguments is done in the GetFunctionArgs auxiliary function, which returns all the passed arguments as a list of strings. You can take a look at the function in the accompanying source code.

Figure 8 **Run Process Function Implementation**

```
class RunFunction : ParserFunction
{
  internal RunFunction(Interpreter interpreter)
  {
    m_interpreter = interpreter;
  }

  protected override Parser.Result Evaluate(string data, ref int from)
  {
    string processName = Utils.GetItem(data, ref from).String;

    if (string.IsNullOrWhiteSpace(processName))
    {
      throw new ArgumentException("Couldn't extract process name");
    }

    List<string> args = Utils.GetFunctionArgs(data, ref from);
    int processId = -1;

    try
    {
      Process pr = Process.Start(processName, string.Join("", args.ToArray()));
      processId = pr.Id;
    }
    catch (System.ComponentModel.Win32Exception exc)
    {
      throw new ArgumentException("Couldn't start [" + processName + "]:
        " + exc.Message);
    }

    m_interpreter.AppendOutput("Process " + processName + " started, id:
      " + processId);
    return new Parser.Result(processId);
  }

  private Interpreter m_interpreter;
}
```

The second and last step is to register the Print function with the Parser in the program initialization part:

```
ParserFunction.AddFunction(Constants.PRINT,     new PrintFunction(this));
```

The Constants.PRINT constant is defined as "print."

Figure 8 shows an implementation of a function that starts a new process.

In order for CSCS to do more useful things, more flesh needs to be added; that is, more functions must be implemented.

Here's how you can find files, start and kill a process, and print some values in CSCS:

```
set(b, findfiles("*.cpp", "*.cs"));
set(i, 0);
while(i < size(b)) {
  print("File ", i, ": ", b(i));
  set(id, run("notepad", b(i)));
  kill(id);
  set(i, i+ 1);
}
```

Figure 9 lists the functions that are implemented in the downloadable source code, along with a brief description. Most of the functions are wrappers over corresponding C# functions.

Internationalization

Note that you can register multiple labels (function names) corresponding to the same function with the Parser. In this way, it's possible to add any number of other languages.

Note that you can register multiple labels (function names) corresponding to the same function with the Parser. In this way, it's possible to add any number of other languages.

Adding a translation consists of registering another string with the same C# object. The corresponding C# code follows:

```
var languagesSection =
  ConfigurationManager.GetSection("Languages") as NameValueCollection;
string languages = languagesSection["languages"];

foreach(string language in languages.Split(",".ToCharArray());)
{
  var languageSection =
    ConfigurationManager.GetSection(language) as NameValueCollection;

  AddTranslation(languageSection, Constants.IF);
  AddTranslation(languageSection, Constants.WHILE);
...
}
```

The AddTranslation method adds a synonym for an already existing function:

```
public void AddTranslation(NameValueCollection languageDictionary, string
originalName)
{
  string translation = languageDictionary[originalName];

  ParserFunction originalFunction =
    ParserFunction.GetFunction(originalName);
  ParserFunction.AddFunction(translation, originalFunction);
}
```

Thanks to C# support of Unicode, most languages can be added this way. Note that the variable names can be in Unicode, as well.

All of the translations are specified in the configuration file. This is how the configuration file looks for Spanish:

```
<Languages>
  <add key="languages" value="Spanish" />
</Languages>
<Spanish>
    <add key="if"    value ="si" />
    <add key="else"  value ="sino" />
    <add key="elif"  value ="sinosi" />
    <add key="while" value ="mientras" />
    <add key="set"   value ="asignar" />
    <add key="print" value ="imprimir" />
 ...
</Spanish>
```

Here's an example of the CSCS code in Spanish:

```
asignar(a, 5);
mientras(a > 0) {
  asignar(expr, 2*(10 - a*3));
  si (expr > 0) {
    imprimir(expr, " es mayor que cero");
  }
  sino {
    imprimir(expr, " es cero o menor que cero");
  }
  asignar(a, a - 1);
}
```

Note that the Parser can now process control statements and functions in both English and Spanish. There's no limit to the number of languages you can add.

Wrapping Up

All of the CSCS elements—control flow statements, variables, arrays, and functions—are implemented by defining a C# class deriving from the ParserFunction base class and overriding its Evaluate method. Then you register an object of this class with the Parser. This approach provides the following advantages:

- Modularity: Each CSCS function and control flow statement resides in its own class, so it's easy to define a new function or a control flow statement or to modify an existing one.
- Flexibility: It's possible to have CSCS keywords and function names in any language. Only the configuration file needs to be modified. Unlike most other languages, in CSCS control flow statements functions and variable names don't have to be in ASCII characters.

Of course, at this stage the CSCS language is far from complete. Here are some ways to make it more useful:

- Creating multidimensional arrays. The same C# data structure as the one for one-dimensional arrays, List<Result>, can be used. However, more parsing functionality must be added when getting and setting an element of the multidimensional array.
- Enabling tuples to be initialized on one line.
- Adding logical operators (AND, OR, NOT and so forth), which would be very useful for if and while statements.
- Adding the capability to write functions and methods in CSCS. Currently, only functions previously written and compiled in C# can be used.
- Adding the capability to include CSCS source code from other units.
- Adding more functions that perform typical OS-related tasks. Because most such tasks can be easily implemented in C#, most would be just a thin wrapper over their C# counterparts.
- Creating a shortcut for the set(a, b) function as "a = b."

I hope you've enjoyed this glimpse of the CSCS language and seeing how you can create your own custom scripting language. ■

Figure 9 **CSCS Functions**

abs	gets the absolute value of an expression
append	appends a string or a number (converted then to a string) to a string
cd	changes a directory
cd..	changes directory one level up
dir	shows the contents of the current directory
enc	gets the contents of an environment variable
exp	exponential function
findfiles	finds files with a given pattern
findstr	finds files containing a string having a specific pattern
indexof	returns an index of a substring, or -1, if not found
kill	kills a process having a given process id number
pi	returns an approximation of the pi constant
pow	returns the first argument to the power of the second argument
print	prints a given list of arguments (numbers and lists are converted to strings)
psinfo	returns process information for a given process name
pstime	returns total processor time for this process; useful for measuring times
pwd	displays current directory pathname
run	starts a process with a given argument list and returns process id
setenv	sets the value of an environment variable
set	sets the value of a variable or of an array element
sin	returns the value of the sine of the given argument
size	returns the length of the string or the size of the list
sqrt	returns the square root of the given number
substr	returns the substring of the string starting from given index
tolower	converts a string to lowercase
toupper	converts a string to uppercase

Vassili Kaplan *is a former Microsoft Lync developer. He's passionate about programming in C# and C++. He currently lives in Zurich, Switzerland, and works as a freelancer for various banks. You can reach him at iLanguage.ch.*

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

DocuVieware

NEW Universal HTML5 Viewer & Document Management Kit

- zero-footprint solution
- super-easy integration
- fast & crystal clear rendering
- fully customizable UI look & feel
- mobile devices optimization
- annotations, thumbnails bookmarks & text search

supports nearly 100 formats

Powered by GdPicture.NET 11

FREE TRIAL
www.docuvieware.com

VISUAL STUDIO LIVE! is on a Campaign for Code in 2016, in support of developer education. First up on the campaign trail? Fabulous Las Vegas, where developers, software architects, engineers, designers and more will convene for five days of unbiased and cutting-edge education on the Microsoft Platform. Sharpen your skills in Visual Studio, ASP.NET, JavaScript, HTML5, Mobile, Database Analytics, and so much more.

EVENT PARTNERS Microsoft Magenic

PLATINUM SPONSOR Alachisoft

SUPPORTED BY Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA YOUR GROWTH. OUR BUSINESS.

LAS VEGAS • MARCH 7-11, 2016

BALLY'S HOTEL & CASINO

Turn the page for more Event details

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

DEVELOPMENT TOPICS INCLUDE:

- ALM/DevOps
- ASP.NET
- Cloud Computing
- Database & Analytics
- JavaScript / HTML5 Client
- Mobile Client
- Modern App Development
- UX/Design
- Visual Studio / .NET
- Windows Client

Be a responsible dev citizen. Register to code with us today!

REGISTER BY FEBRUARY 19 AND SAVE $300!

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

USE PROMO CODE VSLFEB4

VSLIVE.COM/LASVEGAS

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

AGENDA AT-A-GLANCE

Bally's Hotel & Casino will play host to Visual Studio Live!, and is offering a special reduced room rate to conference attendees.

ALM / DevOps	ASP.NET	Cloud Computing	Database and Analytics

START TIME	END TIME	**Pre-Conference Workshops: Monday, March 7, 2016** *(Separate entry fee required)*	
7:30 AM	9:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries	
9:00 AM	1:00 PM	**M01** Workshop: Service Oriented Technologies—Designing, Developing, & Implementing WCF and the Web API - *Miguel Ca stro*	
1:00 PM	2:00 PM	Lunch @ Le Village Buffet, Paris Las Vegas	
2:00 PM	6:00 PM	**M01** Workshop Continues	
7:00 PM	9:00 PM	Dine-A-Round	

START TIME	END TIME	**Day 1: Tuesday, March 8, 2016**	
7:00 AM	8:00 AM	Registration - Coffee and Morning Pastries	
8:00 AM	9:00 AM	Keynote: To Be Announced	
9:15 AM	10:30 AM	**T01** Developer Productivity in Visual Studio 2015 - *Robert Green*	**T02** Angular 101 - *Deborah Kurata*
10:45 AM	12:00 PM	**T06** How to Scale Entity Framework Apps with Distributed Caching - *Iqbal Khan*	**T07** Angular 2.0: A Comparison - *Deborah Kurata*
12:00 PM	1:00 PM	Lunch	
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors	
1:30 PM	2:45 PM	**T11** Native Mobile App Development for iOS, Android and Windows Using C# - *Marcel de Vries*	**T12** ASP.NET 5 in All its Glory - *Adam Tuliper*
3:00 PM	4:15 PM	**T16** Getting Started with Hybrid Mobile Apps with Visual Studio Tools for Cordova - *Brian Noyes*	**T17** MVC 6 - The New Unified Web Programming Model - *Marcel de Vries*
4:15 PM	5:30 PM	Welcome Reception	

START TIME	END TIME	**Day 2: Wednesday, March 9, 2016**	
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries	
8:00 AM	9:15 AM	**W01** Busy .NET Developer's Guide to Native iOS - *Ted Neward*	**W02** Getting Started with Aurelia - *Brian Noyes*
9:30 AM	10:45 AM	**W06** Busy Developer's Guide to Mobile HTML/JS Apps - *Ted Neward*	**W07** Securing Single Page Applications - *Brian Noyes*
11:00 AM	12:00 PM	General Session: To Be Announced	
12:00 PM	1:00 PM	Birds-of-a-Feather Lunch	
1:00 PM	1:30 PM	Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:15pm (Must be present to win)	
1:30 PM	2:45 PM	**W11** Optimizing Applications for Performance Using the Xamarin Platform - *Kevin Ford*	**W12** An Introduction to TypeScript - *Jason Bock*
3:00 PM	4:15 PM	**W16** Leverage Azure App Services Mobile Apps to Accelerate Your Mobile App Development - *Brian Noyes*	**W17** Take a Gulp, Make a Grunt, and Call me Bower - *Adam Tuliper*
4:30 PM	5:45 PM	**W21** Building Cross-Platform Apps Using CSLA.NET - *Rockford Lhotka*	**W22** JavaScript for the C# (and Java) Developer - *Philip Japikse*
7:00 PM	9:00 PM	Evening Out Event	

START TIME	END TIME	**Day 3: Thursday, March 10, 2016**	
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries	
8:00 AM	9:15 AM	**TH01** Building for the Internet of Things: Hardware, Sensors & the Cloud - *Nick Landry*	**TH02** Responsive Web Design with ASP.NET 5 - *Robert Boedigheimer*
9:30 AM	10:45 AM	**TH06** User Experience Case Studies—Good and Bad - *Billy Hollis*	**TH07** Tools for Modern Web Development - *Ben Hoelting*
11:00 AM	12:15 PM	**TH11** Pretty, Yet Powerful. How Data Visualization Transforms The Way We Comprehend Information - *Walt Ritscher*	**TH12** SASS and CSS for Developers - *Robert Boedigheimer*
12:15 PM	1:45 PM	Lunch @ Le Village Buffet, Paris Las Vegas	
1:45 PM	3:00 PM	**TH16** Exposing an Extensibility API for your Applications - *Miguel Castro*	**TH17** Hack Proofing your Web Applications - *Adam Tuliper*
3:15 PM	4:30 PM	**TH21** UWP Development for WPF and Silverlight Veterans - *Walt Ritscher*	**TH22** Increase Website Performance and Search with Lucene.Net Indexing - *Ben Hoelting*

START TIME	END TIME	**Post-Conference Workshops: Friday, March 11, 2016** *(Separate entry fee required)*
7:30 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries
8:00 AM	12:00 PM	**F01** Workshop: Upgrading Your Skills to ASP.NET 5 - *Mark Michaelis*
12:00 PM	1:00 PM	Lunch
1:00 PM	5:00 PM	**F01** Workshop Continues

Speakers and sessions subject to change

CONNECT WITH
VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

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

LAS VEGAS 2016 CAMPAIGN FOR CODE

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

Presented in partnership with **Magenic**

BONUS CONTENT! Modern Apps Live! is now a part of Visual Studio Live! Las Vegas at no additional cost!

JavaScript / HTML5 Client	Mobile Client	UX/Design	Visual Studio / .NET	Windows Client	Modern Apps Live!

Pre-Conference Workshops: Monday, March 7, 2016 *(Separate entry fee required)*

Pre-Conference Workshop Registration - Coffee and Morning Pastries		
M02 Workshop: DevOps in a Day - *Brian Randell*	**M03** Workshop: SQL Server for Developers - *Leonard Lobel*	**M04** Workshop: Modern App Technology Overview—Android, iOS, Cloud, and Mobile Web - *Allen Conway, Brent Edwards, Kevin Ford & Nick Landry*
Lunch @ Le Village Buffet, Paris Las Vegas		
M02 Workshop Continues	**M03** Workshop Continues	**M04** Workshop Continues
Dine-A-Round		

Day 1: Tuesday, March 8, 2016

Registration - Coffee and Morning Pastries		
Keynote: To Be Announced		
T03 Introduction to Next Generation of Azure Compute - Service Fabric and Containers - *Vishwas Lele*	**T04** Technical Debt—Fight it with Science and Rigor - *Brian Randell*	**T05** Defining Modern App Development - *Rockford Lhotka*
T08 Docker and Azure - *Steve Lasker*	**T09** Building Windows 10 Line of Business Applications - *Robert Green*	**T10** Modern App Architecture - *Brent Edwards*
Lunch		
Dessert Break - Visit Exhibitors		
T13 Knockout In 75 Minutes (Or Less...) - *Christopher Harrison*	**T14** Exploring T-SQL Enhancements: Windowing and More - *Leonard Lobel*	**T15** ALM with Visual Studio Online and Git - *Brian Randell*
T18 Building "Full Stack" Applications with Azure App Service - *Vishwas Lele*	**T19** Geospatial Data Types in SQL Server - *Leonard Lobel*	**T20** DevOps and Modern Applications - *Dan Nordquist*
Welcome Reception		

Day 2: Wednesday, March 9, 2016

Registration - Coffee and Morning Pastries		
W03 Exploring Microservices in a Microsoft Landscape - *Marcel de Vries*	**W04** Database Lifecycle Management and the SQL Server Database - *Brian Randell*	**W05** Reusing Logic Across Platforms - *Kevin Ford*
W08 Breaking Down Walls with Modern Identity - *Eric D. Boyd*	**W09** JSON and SQL Server, Finally Together - *Steve Hughes*	**W10** Coding for Quality and Maintainability - *Jason Bock*
General Session: To Be Announced		
Birds-of-a-Feather Lunch		
Dessert Break - Visit Exhibitors - Exhibitor Raffle @ 1:15pm (Must be present to win)		
W13 Real-world Azure DevOps - *Brian Randell*	**W14** Using Hive and Hive ODBC with HDInsight and Power BI - *Steve Hughes*	**W15** Start Thinking Like a Designer - *Anthony Handley*
W18 Lock the Doors, Secure the Valuables, and Set the Alarm - *Eric D. Boyd*	**W19** Introduction to Spark for C# Developers - *James McCaffrey*	**W20** Applied UX: iOS, Android, Windows - *Anthony Handley*
W23 Managing Windows Azure with PowerShell - *Mark Michaelis*	**W24** Introduction to R for C# Programmers - *James McCaffrey*	**W25** Leveraging Azure Services - *Brent Edwards*
Evening Out Event		

Day 3: Thursday, March 10, 2016

Registration - Coffee and Morning Pastries		
TH03 Unit Testing & Test-Driven Development (TDD) for Mere Mortals - *Benjamin Day*	**TH04** Effective Agile Software Requirements - *Richard Hundhausen*	**TH05** Building for the Modern Web with JavaScript Applications - *Allen Conway*
TH08 Unit Testing JavaScript - *Ben Dewey*	**TH09** Lessons Learned: Being Agile in a Waterfall World - *Philip Japikse*	**TH10** Building a Modern Android App with Xamarin - *Kevin Ford*
TH13 End-to-End Dependency Injection & Writing Testable Software - *Miguel Castro*	**TH14** Real World Scrum with Team Foundation Server 2015 & Visual Studio Online - *Benjamin Day*	**TH15** Building a Modern Windows 10 Universal App - *Nick Landry*
Lunch @ Le Village Buffet, Paris Las Vegas		
TH18 Async Patterns for .NET Development - *Ben Dewey*	**TH19** DevOps vs. ALM Different Measures of Success - *Mike Douglas*	**TH20** Panel: Decoding Mobile Technologies - *Rockford Lhotka*
TH23 Improving Quality for Agile Projects Through Manual and Automated UI Testing—NO CODING REQUIRED! - *Mike Douglas*	**TH24** Use Visual Studio to Scale Agile in Your Enterprise - *Richard Hundhausen*	**TH25** Analyzing Results with Power BI - *Scott Diehl*

Post-Conference Workshops: Friday, March 11, 2016 *(Separate entry fee required)*

Post-Conference Workshop Registration - Coffee and Morning Pastries		
F02 Workshop: Building Business Apps on the Universal Windows Platform - *Billy Hollis*	**F03** Workshop: Creating Awesome 2D & 3D Games and Experiences with Unity - *Adam Tuliper*	**F04** Workshop: Modern Development Deep Dive - *Jason Bock, Allen Conway, Brent Edwards & Kevin Ford*
Lunch		
F02 Workshop Continues	**F03** Workshop Continues	**F04** Workshop Continues

VSLIVE.COM/LASVEGAS

Azure Service Fabric, Q-Learning and Tic-Tac-Toe

Jesus Aguilar

Innovations in cloud computing are reducing the barriers to entry for distributed computing and machine learning applications from niche technologies requiring specialized and expensive infrastructure to a commodity offering available to any software developer or solution architect. In this article, I'll describe the implementation of a reinforcement learning technique that leverages the distributed computing and storage capabilities of Azure Service Fabric, the next iteration of the Azure Platform-as-a-Service offering. To demonstrate the potential of this approach, I'll show how you can make use of the Service Fabric and its Reliable Actors programming model to create an intelligent back end that can predict the next move in a game of tic-tac-toe. War Games, anyone?

Enter Q-Learning

Today we see innovative data-driven solutions such as recommendations, face recognition and fraud detection everywhere.

This article discusses:

- The Q-learning reinforcement learning technique
- Azure Service Fabric and its actor programming model
- Exposing an actor's functionality via an API controller
- Implementing the tic-tac-toe scenario

Technologies discussed:

Azure Service Fabric, Q-learning

Code download available at:

aka.ms/servicefabricqlearning

Software engineering teams use supervised and unsupervised learning techniques to implement these solutions. Despite the extensive capabilities of these approaches, there are cases where they are difficult to apply.

Reinforcement learning is a method that handles scenarios you can represent as a sequence of states and transitions. In contrast to other machine learning approaches, reinforcement learning doesn't attempt to generalize patterns by training a model from labeled information (supervised learning) or from unlabeled data (unsupervised learning). Instead, it focuses on problems you can model as a sequence of states and transitions.

Say you have a scenario you can represent as a sequence of states that lead to the final state (known as the absorbing state). Think of a robot making decisions to avoid obstacles, or artificial intelligence (AI) in a game designed to beat an opponent. In many cases, the sequence of states that lead to a particular situation is what determines the best next step for the agent/robot/AI character.

Q-learning is a reinforcement learning technique that uses an iterative reward mechanism to find optimal transitional pathways in a state machine model; it works remarkably well when the number of states and their transitions are finite. In this article, I'll present how I used Service Fabric to build an end-to-end Q-learning solution and show how you can create an intelligent back end that "learns" how to play tic-tac-toe. (Note that state machine scenarios are also referred to as Markov Decision Processes [MDPs]).

First, some basic theory about Q-learning. Consider the states and transitions depicted in **Figure 1**. Say you want to find, at any state, which state an agent needs to transition to next to arrive at the gold state—while minimizing the number of transitions. One way to tackle this problem is to assign a reward value to each state. The reward suggests the value of transitioning to a state toward your goal: getting the gold.

Simple, right? The challenge becomes how to identify the reward for each state. The Q-learning algorithm identifies rewards by recursively iterating and assigning rewards to states that lead to the absorbing (gold) state. The algorithm calculates a state's reward by *discounting* the reward value from a subsequent state. If a state has two rewards—which is possible if a state exists in more than one pathway—the highest prevails. The discount has an important effect on the system. By discounting the reward, the algorithm reduces the value of the reward for states that are far from the gold and assigns more weight to the states closest to the gold.

As an example of how algorithm calculates the reward, look at the state diagram in **Figure 1**. As you can see, there are three pathways to gold:

1→5→4→G
1→5→3→4→G
1→5→3→2→4→G

After running the algorithm using brute force transitioning (iterating through all the possible paths in the graph), the algorithm calculates and assigns the rewards for the valid pathways. Rewards are calculated with the discount factor of 0.9.

1(R=72)→5(R=81)→4(R=90)→G (R=100)
1(R=64)→5(R=72)→3(R=81)→4(R=90)→ G(R=100)
1(R=58)→5(R=64)→3(R=72)→2(R=81)→4(R=90)→ G(R=100)

Because some states have more than one reward, the highest value will prevail. **Figure 2** depicts the final reward assignment.

With this information, an agent can identify the optimal path to gold in any state by transitioning to the state with the highest reward. For instance, if the agent is in state 5, it has the choice to transition to states 3 or 4, and 4 becomes the choice because the reward is higher.

Azure Service Fabric

Service Fabric, the next iteration of the Azure Platform-as-a-Service offering, empowers developers to create distributed applications using two different top-level programming models: Reliable Actors and Reliable Services. These programming models allow you to maximize the infrastructure resources of a distributed platform. The platform handles the most difficult tasks associated with maintaining and running a distributed application—recovery from failures, distribution of services to ensure efficient resource utilization, rolling updates and side-to-side versioning, to mention a few.

Service Fabric provides you with a cluster, giving you a higher level of abstraction to use, rather than having to worry about the underlying infrastructure. Your code runs on the nodes of a Service Fabric cluster, and you can host multi-node clusters on a single machine for development purposes or on multiple servers (virtual or physical machines) for production workloads. The platform manages the lifecycle of your Actors and Services and the recovery from infrastructure failures.

Service Fabric introduces Reliable Actors and Services with stateful semantics. This capability translates into a fully integrated developer experience in which you can develop applications that persist data in a distributed and therefore highly available manner without having to include an external storage layer (for example, taking dependency on an external storage or caching layer) in your architecture.

Reinforcement learning is a method that handles scenarios you can represent as a sequence of states and transitions.

By implementing the Q-learning algorithm as a service within Service Fabric, you can benefit from having distributed computing and low-latency state storage capabilities, enabling you to execute the algorithm, persist the results, and expose the whole thing as reliable end points for clients to access. All these capabilities come together in a single solution with a unified programming and management stack. There's no need to add additional components to your architecture, such as an external storage, cache or messaging system. In short, you have a solution in which your compute, data and services reside within the same integrated platform. That's an elegant solution in my book!

Q-Learning and Reliable Actors

The actor model simplifies the design of massively concurrent applications. In the actor model, actors are the fundamental computing unit. An actor represents a boundary of functionality and state. You can think of an actor as an object entity living in a distributed system. Service Fabric manages the lifecycle of the actor. In the event of failure, Service Fabric re-instantiates the actor in a healthy node automatically. For example, if a stateful actor crashes

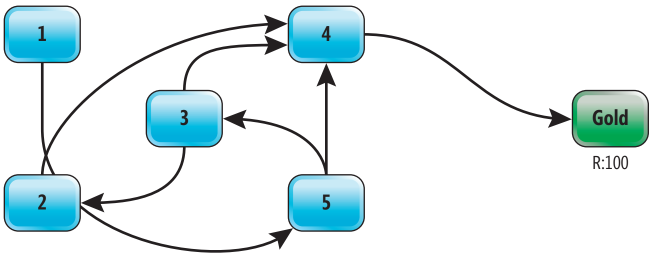


Figure 1 **A Sequence of States Leading to the Gold State**

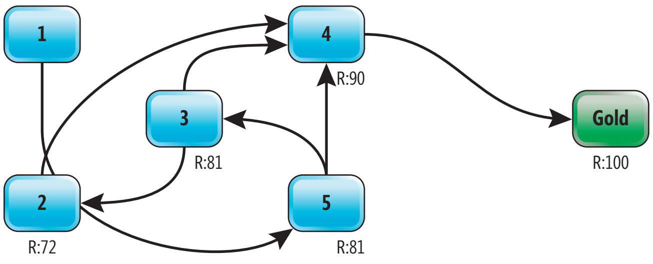


Figure 2 **Final Rewards**

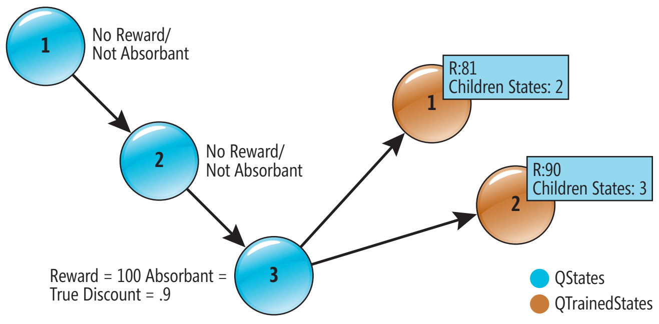


Figure 3 **Determining and Persisting Rewards**

for some reason or the node (think VM) it's running on fails, the actor is automatically re-created on another machine with all of its state (data) intact.

Service Fabric also manages how an instance of an actor is accessed. The platform guarantees that at any point in time, only one method on a particular actor is executing at a time. If there are two concurrent calls to the same actor, Service Fabric will queue one and let the other proceed. The implication is that inside an actor your code doesn't have to worry about race conditions, locks or synchronization.

Once the algorithm identifies an absorbing state with a reward, it will calculate rewards for all the states that lead to the absorbing state.

As I described earlier, the Q-learning algorithm iterates through states with the goal of finding absorbing states and states with rewards. Once the algorithm identifies an absorbing state with a reward, it will calculate rewards for all the states that lead to the absorbing state.

Using the actor model, I can model this functionality as an actor that represents a state in the context of the Q-learning algorithm (think of stages in the overall graph). In my implementation, the actor type that represents these states is QState. Once there's a transition to a QState actor containing a reward, the QState actor will create another actor instance of a different type (QTrainedState) for each of the QState actors in the pathway. QTrainedState actors maintain the maximum reward value and a list of the subsequent states that yield the reward. The list contains the state tokens (which uniquely identifies a state in the graph) of the subsequent states.

In **Figure 3**, I depict the logic of the algorithm using actors, for a very simple scenario where a state with state token 3 is an absorbing state, contains a reward of 100, and has only one pathway with two previous states (state token 1 and 2). Each circle represents an instance of an actor, QStates in blue and QTrainedStates in orange. Once the transition process reaches the QState with state token 3, the QState actor will create two QTrainedStates, one for each of the previous QStates. For the QTrainedState actor that represents state token 2, the suggested transition (for a reward of 90) is to state token 3, and for the QTrainedState actor that represents state token 1, the suggested transition (for a reward of 81) is to state token 2.

It's possible that multiple states will yield the same reward, so the QTrainedState actor persists a collection of state tokens as children states.

The following code shows the implementation of the interfaces for the QState and QTrainedState actors, called IQState and IQTrainedState. QStates have two behaviors: transitioning to other QStates and starting the transition process when no prior transition exists:

```
public interface IQState : IActor
{
  Task StartTrainingAsync(int initialTransitionValue);

  Task TransitionAsync(int? previousStateToken, int transitionValue);
}
public interface IQTrainedState:IActor
{
 .Task AddChildQTrainedStateAsync(int stateToken, double reward);

 .Task<List<int>> GetChildrenQTrainedStatesAsync();
}
```

Notice that the implementation of IQTrainedState surfaces the method GetChildrenQTrainedStatesAsync. This method is how the QTrainedState actor will expose the trained data containing the states with the highest reward value for any state in the system. (Note that all actors in the Service Fabric must implement an interface derived from IActor.)

QState Actor

After defining the interfaces, I can move to the implementation of the actors. I'll start with the QState actor and the TransitionAsync method, which is the cornerstone of the algorithm and where most of the work resides. TransitionAsync makes the transition to another state by creating a new instance of a QState actor and calling the same method again.

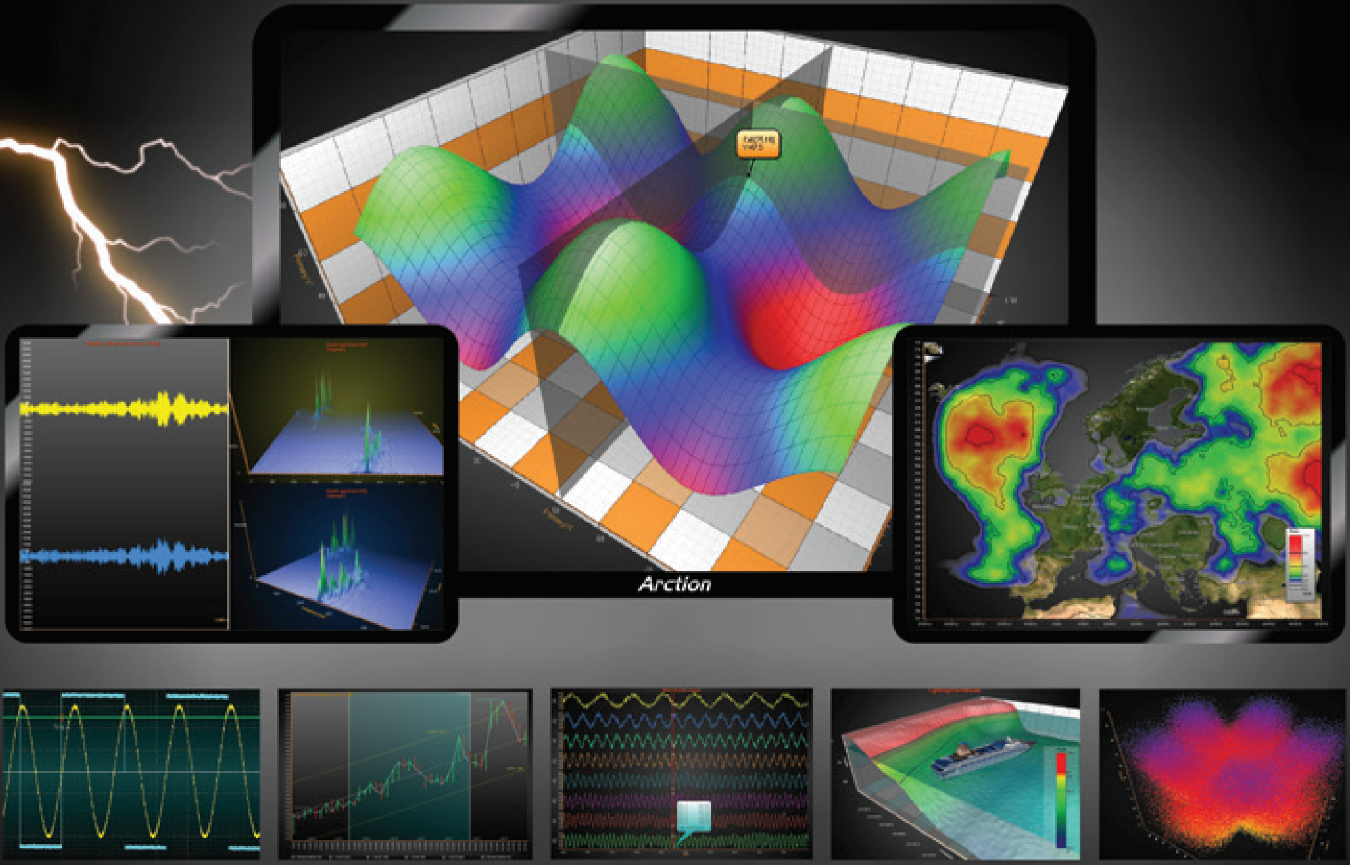
Figure 4 **TransitionAsync in the QState Class**

```
public abstract class QState : StatefulActor, IQState, IRemindable
{
  // ...
  public Task TransitionAsync(int? previousStateToken, int transitionValue)
  {
    var rwd = GetReward(previousStateToken, transitionValue);

    var stateToken = transitionValue;
    if (previousStateToken != null)
        stateToken = int.Parse(previousStateToken.Value + stateToken.ToString());

    var ts = new List<Task>();

    if (rwd == null || !rwd.IsAbsorbent)
      ts.AddRange(GetTransitions(stateToken).Select(p =>
        ActorProxy.Create<IQState>(ActorId.NewId(),
        "fabric:/QLearningServiceFab").TransitionAsync(stateToken, p)));

    if (rwd != null)
      ts.Add(RegisterReminderAsync("SetReward",
        Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(rwd))
        , TimeSpan.FromMilliseconds(0)
        , TimeSpan.FromMilliseconds(-1), ActorReminderAttributes.Readonly));

      return Task.WhenAll(ts);
  }
  // ...
}
```

WINDOWS 10 COMPATIBLE

The FASTEST rendering Data Visualization components for WPF and WinForms...

LightningChart



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

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2015
- Hundreds of examples

WPF charts performance comparison

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

WinForms charts performance comparison

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

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

Download a free 30-day evaluation from

www.LightningChart.com

Arction

Figure 5 **The SetRewardAsync and the ReceiveReminderAsync Methods**

```
public Task SetRewardAsync(int stateToken, double stateReward, double discount)
  {
    var t = new List<Task>();
    var reward = stateReward;

    foreach (var pastState in GetRewardingQStates(stateToken))
    {
      t.Add(ActorProxy
        .Create<IQTrainedState>(new ActorId(pastState.StateToken),
          "fabric:/QLearningServiceFab")
        .AddChildQTrainedStateAsync(pastState.NextStateToken, reward));

      reward = reward * discount;
    }

    return Task.WhenAll(t);

  }

public async Task ReceiveReminderAsync(string reminderName,
  byte[] context, TimeSpan dueTime, TimeSpan period)
  {
    await UnregisterReminderAsync(GetReminder(reminderName));

    var state = JsonConvert.DeserializeObject<JObject>(
      Encoding.UTF8.GetString(context));

    if (reminderName == "SetReward")
    {
      await SetRewardAsync(state["StateToken"].ToObject<int>(),
        state["Value"].ToObject<double>(),
        state["Discount"].ToObject<double>());
    }

    if (reminderName == "StartTransition")
    {
      await TransitionAsync(null, state["TransitionValue"].ToObject<int>());
    }
  }
```

You might wonder if by calling the method recursively you'd avoid the overhead of invoking the method through another actor instance. A recursive method call is a compute-intensive operation in a single node. In contrast, by instantiating another actor, you're taking advantange of the capabilities of Service Fabric by letting the platform distribute the processing across horizontal computing resources.

Reminders are new constructs introduced in the actor programming model that allow you to schedule asynchronous work without blocking the execution of a method.

To manage the assignment of the reward, I'll register a reminder. Reminders are new constructs introduced in the actor programming model that allow you to schedule asynchronous work without blocking the execution of a method.

Reminders are available only for stateful actors. For both stateless and stateful actors, the platform provides timers that enable similar patterns. One important consideration is that when the actor is used, the garbage collection process is delayed; nevertheless, the platform doesn't consider timer callbacks as usage. If the garbage collector kicks in, the timers will be stopped. Actors won't be garbage collected while a method is executed. To guarantee recurring execution, use reminders. More information can be found at bit.ly/1RmzKfr.

The goal, as it relates to the algorithm, is to perform the reward assignment without blocking the transition process. Typically, scheduling a work item in the thread pool with a callback will suffice; however, in the actor programming model, this approach is not a good idea as you'll lose the concurrency benefits of the platform.

The platform guarantees that only one method is executed at any time. This capability allows you to write your code without considering concurrency; that is, without having to worry about thread safety. As you'd expect, there's a trade-off: You must avoid creating tasks or threads to wrap operations inside the actor methods. The reminders allow you to implement background

Figure 6 **The QTrainedState Class**

```
public class QTrainedState : StatefulActor<QTrainedStateState>, IQTrainedState
{
  protected async override Task OnActivateAsync()
  {
    this.State =
      await ActorService.StateProvider.LoadStateAsync<QTrainedStateState>(
      Id, "qts") ??
      new QTrainedStateState() { ChildrenQTrainedStates = new HashSet<int>() };

    await base.OnActivateAsync();
  }

  protected async override Task OnDeactivateAsync()
  {
    await ActorService.StateProvider.SaveStateAsync(Id, "qts", State);

    await base.OnDeactivateAsync();
  }

  [Readonly]
  public  Task AddChildQTrainedStateAsync(int stateToken, double reward)
  {


    if (reward < State.MaximumReward)
    {
      return Task.FromResult(true);
    }

    if (Math.Abs(reward - State.MaximumReward) < 0.10)
    {
      State.ChildrenQTrainedStates.Add(stateToken);
      return Task.FromResult(true);
    }

      State.MaximumReward = reward;
      State.ChildrenQTrainedStates.Clear();
      State.ChildrenQTrainedStates.Add(stateToken);

      return Task.FromResult(true);
  }

  [Readonly]
  public Task<List<int>> GetChildrenQTrainedStatesAsync()
  {
    return Task.FromResult(State.ChildrenQTrainedStates.ToList());
  }
}
```

Figure 7 **The API Controller**

```
[Route("api/[controller]")]
public class QTrainerController : Controller
{
  [HttpGet()]
  [Route("[action]/{startTrans:int}")]
  public  async Task<IActionResult>  Start(int startTrans)
  {
    var actor = ActorProxy.Create<IQState>(ActorId.NewId(),
      "fabric:/QLearningServiceFab/");

    await actor.StartTrainingAsync(startTrans);

    return Ok(startTrans);
  }

  [HttpGet()]
  [Route("[action]/{stateToken}")]
  public async Task<int> NextValue(int stateToken)
  {
    var actor = ActorProxy.Create<IQTrainedState>(new ActorId(stateToken),
      "fabric:/QLearningServiceFab");

    var qs = await actor.GetChildrenQTrainedStatesAsync();

    return qs.Count == 0 ? 0 : qs[new Random().Next(0, qs.Count)];
  }
}
```

processing scenarios with the concurrency guarantees of the platform, as shown in **Figure 4**.

(Note that setting dueTime to TimeSpan.FromMilliseconds(0)) indicates an immediate execution.)

To complete the implementation of IQState, the following code implements the StartTransitionAsync method, where I use a reminder to avoid a blocking long-running call:

```
public Task StartTrainingAsync(int initialTransitionValue)
  {
    return RegisterReminderAsync("StartTransition",
      Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(new { TransitionValue =
      initialTransitionValue })), TimeSpan.FromMilliseconds(0),
      TimeSpan.FromMilliseconds(-1),
      ActorReminderAttributes.Readonly);
  }
```

To finalize the implementation of the QState class, I'll describe the implementation of the SetRewardAsync and the ReceiveReminderAsync methods, shown in **Figure 5**. The SetReward method creates or updates a stateful actor (the implementation of IQTrainedState). To locate the actor in subsequent calls, I use the state token as the actor id—actors are addressable.

QTrainedState Actor

The second actor in the solution is QTrainedState. The data in QTrainedState actor must be durable, therefore I implemented this actor as a stateful actor.

In Service Fabric, you implement a stateful actor by deriving your class from StatefulActor or the StatefulActor<T> base classes and implementing an interface derived from IActor. T is the type of the state instance, which must be serializable and a reference type. When you call a method of a class derived from StatefulActor<T>, the platform loads the state from the state provider, and once the call completes, the platform saves it automatically. In the case of the QTrainedState, I modeled the state (durable data) using the following class:

```
[DataContract]
public class QTrainedStateState
{
  [DataMember]
  public double MaximumReward { get; set; }

  [DataMember]
  public HashSet<int> ChildrenQTrainedStates { get; set; }
}
```

Figure 6 shows the complete implementation of the QTrainedState class, which implements the two methods of the IQTrainedState interface.

Surfacing the Actors

At this point, the solution has everything necessary to start the training process and persist the data. But I haven't yet discussed how clients will interact with these actors. At a high level, this interaction consists of starting the training process and querying the persisted data. Each of these interactions correlates nicely with an API operation, and a RESTful implementation facilitates the integration with clients.

In addition to having the two programming models, Service Fabric is a comprehensive orchestration and process management platform.

In addition to having the two programming models, Service Fabric is a comprehensive orchestration and process management platform. The failure recovery and resource management that exists for actors and services is also available to other processes. For instance, you can run Node.js or ASP.NET 5 processes, managed by Service Fabric, and benefit from these capabilities without further effort. So I can just use a standard ASP.NET 5 Web API application and create an API controller that exposes the relevant actor's functionality, as shown in **Figure 7**.

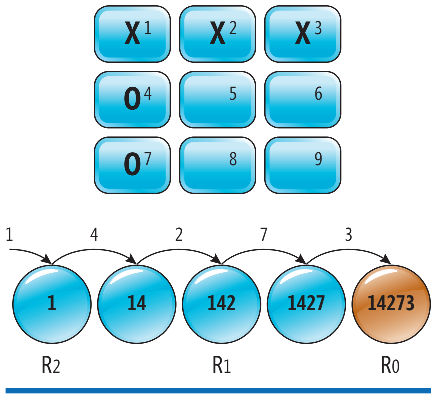


Figure 8 **The Tic-Tac-Toe Scenario**

And Tic-Tac-Toe?

What's left now is to make use of the solution with a concrete scenario. For this, I'll use a simple game: tic-tac-toe.

The goal is to train a set of QTrainedStates that you can query to predict the next move in a game of tic-tac-toe. One way to think about this is that the machine is acting as both players and learning from the outcomes.

Figure 9 **The GetReward Method**

```
internal override IReward GetReward(int? previousStateToken, int transitionValue)
{
  var game = new TicTacToeGame(previousStateToken,transitionValue);
  IReward rwd = null;

  if (game.IsBlock)
  {
    rwd = new TicTacToeReward() { Discount = .5, Value = 95, IsAbsorbent = false,
      StateToken = game.StateToken};
  }
  if (game.IsWin)
  {
    rwd = new TicTacToeReward() { Discount = .9, Value = 100, IsAbsorbent = true,
      StateToken = game.StateToken };
  }
  if (game.IsTie)
  {
    rwd = new TicTacToeReward() { Discount = .9, Value = 50, IsAbsorbent = true,
      StateToken = game.StateToken };
  }
  return rwd;
}
```

Going back to the implementation, notice that QState is an abstract class. The idea is to encapsulate the basic aspects of the algorithm and put the logic of the specific scenario in a derived class. A scenario defines three parts of the algorithm: how the transition between states occurs (policy); what states are absorbing and have an initial reward; and the states the algorithm will assign a reward with a discount. For each of these parts, the QState class has a method where you can implement these semantics to solve a specific scenario. These methods are GetTransitions, GetReward and GetRewardingQStates.

So the question becomes: How can you model a game of tic-tac-toe as a sequence of states and transitions?

Consider the game depicted in **Figure 8**, where each cell has a number assigned. You can think of each turn as a transition from one state to another in which the transition value is the cell where the player is making a play. Each state token is then a combination of the previous turns (cells) and the transition value. For the example in **Figure 8**, a transition from 1 to 14, and then to 142, and so on, models the steps of the game where the player that played the first turn wins. And in this case, all the states that lead to 14273 (the winning and absorbing state) must be assigned a reward: 1 and 142.

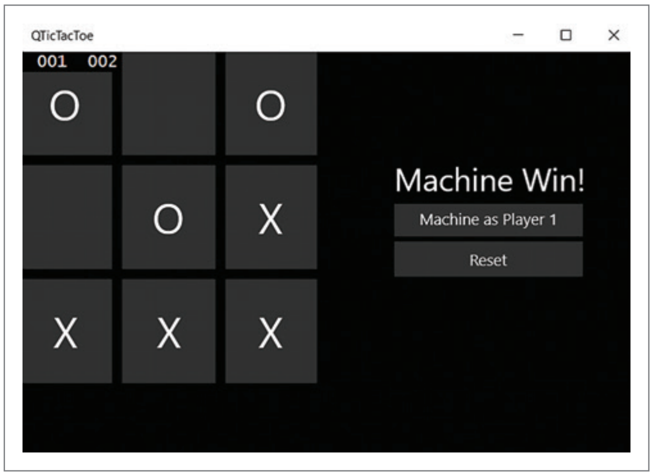


Figure 10 **A Game of Tic-Tac-Toe**

Going back to Q-learning, what I need to provide are all the final (absorbing) states, each with an initial reward. For tic-tac-toe, three types of states will yield a reward: a win, a tie or a block (referring to a point when your opponent is about to win, so you are forced to use your turn to block him). A win and a tie are absorbing, meaning the game ends; a block is not, however, and the game continues. **Figure 9** shows the implementation of the GetReward method for the game of tic-tac-toe.

Next, once a state is identified with a reward, I need to provide the algorithm with the states that led to the state with an initial reward so a discounted reward can be assigned. For win or block scenarios, these states are all the previous states (plays) of the winning or blocking player. For ties, all the states (plays) of both players must be assigned a reward:

```
internal override IEnumerable<IPastState> GetRewardingQStates(int stateToken)
{
  var game = new TicTacToeGame(stateToken);

  if (game.IsTie)
    return game.GetAllStateSequence();

  return game.GetLastPlayersStateSequence();
}
```

Finally, I need to implement a transition policy that determines how the algorithm will iterate through the states. For the game, I'll implement a transition policy where all possible combinations are explored:

```
internal override IEnumerable<int> GetTransitions(int stateToken)
{
  var game = new TicTacToeGame(stateToken);

  return game.GetPossiblePlays();
}
```

Playing Against the Machine

At this point, I can publish the solution and start the training by calling REST API and providing the initial transitions: 1 to 9.

Once the training finishes, you can use the API to create an application that can simply pass a state token and receive the suggested value. The source code for this article contains a Universal Windows Platform app that uses this back end. **Figure 10** shows the game.

Wrapping Up

By using Q-learning and Service Fabric, I was able to create an end-to-end framework that leverages a distributed platform to compute and persist data. To showcase this approach, I used the game of tic-tac-toe to create a back end that learns how to play the game, and does so at an acceptable level by only indicating when a win, a tie, or a block occurs and letting the machine learn by playing the game. ■

Jesus Aguilar *is a senior cloud architect at Microsoft in the Technical Evangelism and Development team where he partners with awesome companies that are born in the cloud, and helps them deliver compelling experiences at scale. He is passionate about software engineering and solution design, and you will catch his attention by using terms such as "Predictive Analytics," "Scalability," "Concurrency," "Design Patterns" and "<Choose any Letter>aaS." You can follow him on Twitter: @giventocode and check out his blog at giventocode.com.*

Thanks to the following Microsoft technical experts for reviewing this article: *Rob Bagby, Mike Lanzetta and Matthew Snider*

Extreme Performance Linear Scalability

Cache data, reduce expensive database trips, and scale your apps to extreme transaction processing (XTP) with NCache.

In-Memory Distributed Cache

- Extremely fast & linearly scalable with 100% uptime
- Mirrored, Replicated, Partitioned, and Client Cache
- Entity Framework & NHibernate Second Level Cache

ASP.NET Optimization in Web Farms

- ASP.NET Session State storage
- ASP.NET View State cache
- ASP.NET Output Cache provider

Full Integration with Microsoft Visual Studio

- NuGet Package for NCache SDK
- Microsoft Certified for Windows Server 2012 R2

Open Source

NCache™

FREE Download

sales@alachisoft.com US: +1 (925) 236 3830 www.alachisoft.com

Roach Infestation Optimization

Roach infestation optimization is a numerical optimization algorithm that's loosely based on the behavior of common cockroaches such as *Periplaneta americana*. Yes, you read that correctly. Let me explain.

In machine learning, a numerical optimization algorithm is often used to find a set of values for variables (usually called weights) that minimize some measure of error. For example, logistic regression classification uses a math equation where, if there are n predictor variables, there are n+1 weight values that must be determined. The process of determining the values of the weights is called training the model. The idea is to use a collection of training data that has known correct output values. A numerical optimization algorithm is used to find the values of the weights that minimize the error between computed output values and known correct output values.

There are many different numerical optimization algorithms. The most common are based on calculus derivatives, but there are also algorithms that are based on the behaviors of natural systems. These are sometimes called bio-inspired algorithms. This article explains a relatively new (first published in 2008) technique called roach infestation optimization (RIO). RIO optimization loosely models the foraging and aggregating behavior of a collection of roaches.

Roach infestation optimization is a numerical optimization algorithm that's loosely based on the behavior of common cockroaches.

A good way to get an idea of what RIO is and to see where this article is headed is to take a look at the demo program in **Figure 1**. The goal of the demo is to use RIO to find the minimum value of the Rastrigin function with eight input variables. The Rastrigin function is a standard benchmark function used to evaluate the effectiveness of numerical optimization algorithms. The function has a known minimum value of 0.0 located at x = (0, 0, . . 0) where the number of 0 values is equal to the number of input values.

The Rastrigin function is difficult for most optimization algorithms because it has many peaks and valleys that create local minimum values that can trap the algorithms. It's not possible to easily visualize the Rastrigin function with eight input values, but you can get a good idea of the function's characteristics by examining a graph of the function for two input values, shown in **Figure 2**.

The most common algorithms are based on calculus derivatives, but there are also algorithms that are based on the behaviors of natural systems.

The demo program sets the number of roaches to 20. Each simulated roach has a position that represents a possible solution to the minimization problem. More roaches increase the chance of finding the true optimal solution at the expense of performance. RIO typically uses 10 to 100 roaches.

RIO is an iterative process and requires a maximum loop counter value. The demo sets the maximum value to 10,000 iterations. The maximum number of iterations will vary from problem to problem, but values between 1,000 and 100,000 are common. RIO has an element of randomness and the demo sets the seed value for the random number generator to 6, because 6 gave representative demo output.

In the demo shown in **Figure 1**, the best (smallest) error associated with the best roach position found so far was displayed every 500 time units. After the algorithm finished, the best position found for any roach was x = (0, 0, 0, 0, 0, 0, 0, 0), which is, in fact, the correct answer. But notice if the maximum number of iterations had been set to 5,000 instead of 10,000, RIO would not have found the one global minimum. RIO, like almost all numerical optimization algorithms, is not guaranteed to find an optimal solution in practical scenarios.

This article assumes you have at least intermediate programming skills but doesn't assume you know anything about numerical

Code download available at msdn.com/magazine/0216magcode.

optimization or the roach infestation optimization algorithm. The demo program is coded using C#, but you shouldn't have too much difficulty refactoring the code to another language such as Visual Basic or JavaScript.

The complete demo code, with a few minor edits to save space, is presented in this article. The demo is also available in the code download that accompanies this article. The demo code has all normal error checking removed to keep the main ideas as clear as possible and the size of the code small.

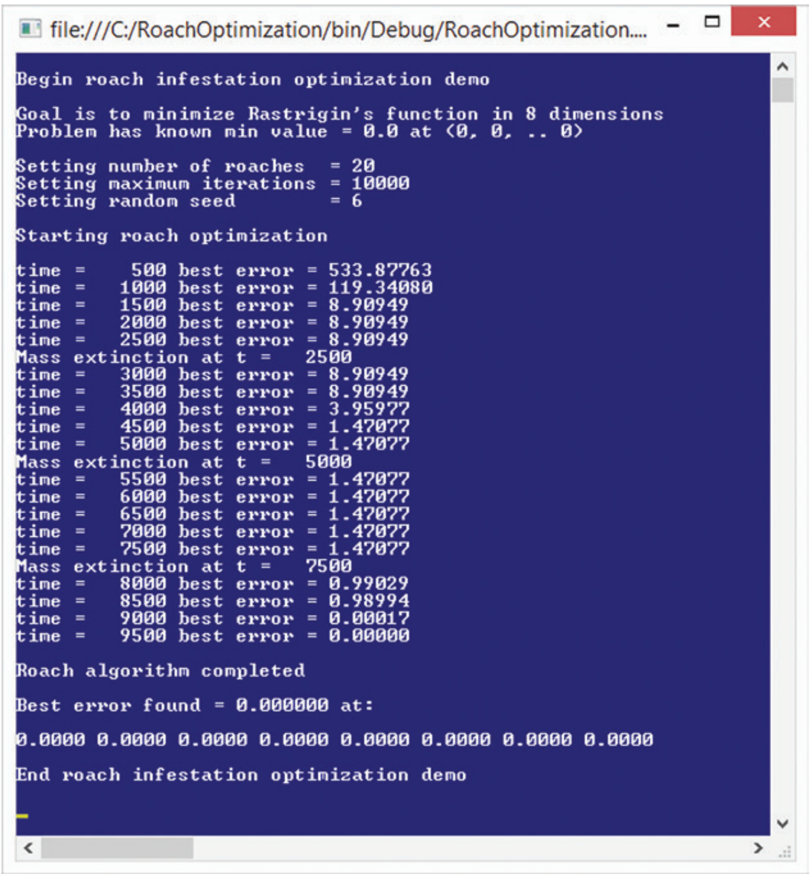
Overall Program Structure

The overall program structure is presented in **Figure 3**. To create the demo, I launched Visual Studio and created a new C# console application named RoachOptimization. The demo has no significant Microsoft .NET Framework dependencies so any recent version of Visual Studio will work.

After the template code loaded into the Visual Studio editor, in the Solution Explorer window I renamed file Program.cs to the more descriptive RoachOptimizationProgram.cs and Visual Studio automatically renamed class Program for me. At the top of the source code, I deleted all unnecessary using statements, leaving just the single reference to System.

I coded the demo using a mostly static-method approach rather than a full Object-Oriented Programming (OOP) approach. The demo has all the control logic in the Main method. It begins by setting up the algorithm input parameter values:

```
Console.WriteLine("Begin roach optimization demo");
int dim = 8;
int numRoaches = 20;
int tMax = 10000;
int rndSeed = 6;
```


file:///C:/RoachOptimization/bin/Debug/RoachOptimization....

```
Begin roach infestation optimization demo

Goal is to minimize Rastrigin's function in 8 dimensions
Problem has known min value = 0.0 at (0, 0, .. 0)

Setting number of roaches   = 20
Setting maximum iterations = 10000
Setting random seed         = 6

Starting roach optimization

time =    500 best error = 533.87763
time =   1000 best error = 119.34080
time =   1500 best error = 8.90949
time =   2000 best error = 8.90949
time =   2500 best error = 8.90949
Mass extinction at t =   2500
time =   3000 best error = 8.90949
time =   3500 best error = 8.90949
time =   4000 best error = 3.95977
time =   4500 best error = 1.47077
time =   5000 best error = 1.47077
Mass extinction at t =   5000
time =   5500 best error = 1.47077
time =   6000 best error = 1.47077
time =   6500 best error = 1.47077
time =   7000 best error = 1.47077
time =   7500 best error = 1.47077
Mass extinction at t =   7500
time =   8000 best error = 0.99029
time =   8500 best error = 0.98994
time =   9000 best error = 0.00017
time =   9500 best error = 0.00000

Roach algorithm completed

Best error found = 0.000000 at:

0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

End roach infestation optimization demo

_
```

Figure 1 **The Roach Infestation Optimization Algorithm in Action**

The Rastrigin function is difficult for most optimization algorithms because it has many peaks and valleys that create local minimum values that can trap the algorithms.

The dim variable specifies the number of input values for Rastrigin's function. In a non-demo machine learning scenario, the dim represents the number of weights in the prediction model. The number of roaches is set to 20. Variable tMax is the maximum number of iterations. RIO, like most bio-inspired algorithms, is probabilistic. Here, a random variable seed value is set to 6.

Next, the RIO parameters are echoed to the console:

```
Console.WriteLine("Goal is to minimize Rastrigin's " +
  "function in " + dim + " dimensions");
Console.WriteLine("Problem has known min value = 0.0 " +
  "at (0, 0, .. 0) ");
Console.WriteLine("Setting number of roaches = " +
  numRoaches);
Console.WriteLine("Setting maximum iterations = " +
  tMax);
Console.WriteLine("Setting random seed = " + rndSeed);;
```

The roach optimization algorithm is called like so:

```
Console.WriteLine("Starting roach optimization ");
double[] answer = SolveRastrigin(dim, numRoaches,
  tMax, rndSeed);
Console.WriteLine("Roach algorithm completed");
```

The Main method concludes by displaying the results:

```
double err = Error(answer);
Console.WriteLine("Best error found = " +
  err.ToString("F6") + " at: ");
for (int i = 0; i < dim; ++i)
  Console.Write(answer[i].ToString("F4") + " ");
Console.WriteLine("");
Console.WriteLine("End roach optimization demo");
Console.ReadLine();
```

The roach optimization algorithm presented in this article is based on the 2008 research paper, "Roach Infestation Optimization," by T. Havens, C. Spain, N. Salmon and J. Keller. You can find the paper in several locations on the Web.

Understanding the Roach Optimization Algorithm

In RIO, there's a collection of simulated roaches. Each roach has a position in n-dimensions that represents a possible solution to a minimization problem. Simulated roach behavior is based on three

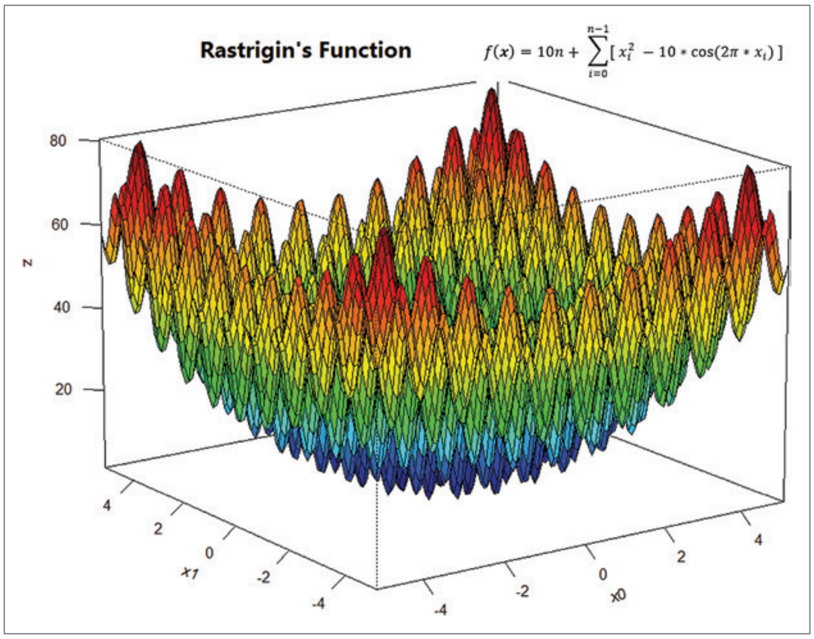


Figure 2 **The Rastrigin Function with Two Input Variables**

behaviors of real roaches. First, roaches tend to move toward dark areas. Second, roaches like to group together. Third, when roaches get hungry, they will leave their current location to search for food. Exactly how simulated roaches model these behaviors will become clear when the code is presented.

Expressed in very high-level pseudocode, the roach optimization algorithm is presented in **Figure 4**. At first glance, the algorithm seems quite simple; however, there are a lot of details that aren't apparent in the pseudo-code.

The personalBestPos field holds the best position found by the simulated roach at any point during its movement.

The Roach Class

The definition of program-defined class Roach begins as:

```
public class Roach
{
  public int dim;
  public double[] position;
  public double[] velocity;
  public double error;
...
```

The dim field is the problem dimension, which is 8 in the case of the demo. The position field is an array that conceptually represents the location of a roach, and also represents a possible solution to a minimization problem. The velocity field is an array of values that determine where the roach will move in the next time unit. For example, if dim = 2 and position = (5.0, 3.0) and velocity = (1.0, -2.0), the roach will move to (6.0, 1.0). The error field is the error associated with the current position.

The definition continues:

```
public double[] personalBestPos;
public double personalBestErr;
public double[] groupBestPos;
public int hunger;
private Random rnd;
```

The personalBestPos field holds the best position found by the simulated roach at any point during its movement. The personalBestError holds the error that corresponds to personalBestPos. The groupBestPos field holds the best position found by any of a group of neighbor roaches. The hunger field is an integer value that represents how hungry the roach is. The Random object rnd is used to initialize a roach to a random position.

The Roach class definition continues by defining a constructor:

```
public Roach(int dim, double minX, double maxX,
  int tHunger, int rndSeed)
{
  this.dim = dim;
  this.position = new double[dim];
  this.velocity = new double[dim];
  this.personalBestPos = new double[dim];
  this.groupBestPos = new double[dim];
...
```

The minX and maxX parameters are used to set limits for the components of the position vector. Parameter tHunger is a maximum hunger value. When a roach's hunger reaches tHunger, the roach will move to a new location. The constructor allocates space for the four array fields.

Next, the constructor initializes the Random object and then uses it to set the initial hunger value to a random value:

```
this.rnd = new Random(rndSeed);
this.hunger = this.rnd.Next(0, tHunger);
```

Next, the constructor sets the initial position, velocity, personal best location and group best location arrays to random values between minX and maxX:

Figure 3 **Roach Optimization Demo Program Structure**

```
using System;
namespace RoachOptimization
{
  class RoachOptimizationProgram
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Begin roach optimization demo");
      // Code here
      Console.WriteLine("End roach demo");
      Console.ReadLine();
    }

    static double[] SolveRastrigin(int dim, int numRoaches,
      int tMax, int rndSeed) { . . }

    public static double Error(double[] x) { . . }

    static double Distance(double[] pos1,
      double[] pos2) { . . }

    static void Shuffle(int[] indices,
      int seed) { . . }

  } // Program

  public class Roach
  {
    // Defined here
  }
}
```

Spreadsheets Made Easy.

WIN WPF

Windows Forms Silverlight WPF

Fastest Calculations

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

Powerful Controls

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

Comprehensive Charting

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

Scalable Reporting

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

Download your free fully functional evaluation at SpreadsheetGear.com

SpreadsheetGear

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

```
for (int i = 0; i < dim; ++i) {
  this.position[i] = (maxX - minX) * rnd.NextDouble() + minX;
  this.velocity[i] = (maxX - minX) * rnd.NextDouble() + minX;
  this.personalBestPos[i] = this.position[i];
  this.groupBestPos[i] = this.position[i];
}
```

The Roach constructor definition finishes like so:

```
...
    error = RoachOptimizationProgram.Error(this.position);
    personalBestErr = this.error;
  } // ctor
} // Roach
```

The error field is set by calling external method Error, defined in the calling program class. An alternative approach is to compute the error value before calling the constructor, then pass the error value in as a parameter to the constructor.

Implementing the Roach Algorithm

The RIO algorithm is contained in static method SolveRastrigin, whose definition begins as:

```
static double[] SolveRastrigin(int dim, int numRoaches,
  int tMax, int rndSeed)
{
  double C0 = 0.7;
  double C1 = 1.43;
  double[] A = new double[] { 0.2, 0.3, 0.4 };
...
```

Constants C0 and C1 are used when computing a roach's new velocity, as you'll see shortly. The values used, 0.7 and 1.43, come from particle swarm theory. You might want to investigate other values.

I discovered the probabilities that match real roach behavior weren't as effective as the artificial probability values used in the demo.

Roaches that are close to each other are called neighbors. Neighbors will sometimes, but not always, exchange information. The array named A holds probabilities. The first value, 0.2, is the probability that a roach with one neighbor will exchange information with that neighbor. The second value, 0.3, is the probability that a roach will exchange information if it has two neighbors. The third value, 0.4, is the probability of exchanging information if a roach has three or more neighbors.

These probability values, (0.2, 0.3, 0.4), are not the ones that were used in the source research paper. The original study used probability values of (0.49, 0.63, 0.65), which correspond to actual roach behavior as described in a biology research paper. I discovered the probabilities that match real roach behavior weren't as effective as the artificial probability values used in the demo. The definition of method SolveRastrigin continues with:

```
int tHunger = tMax / 10;
double minX = -10.0;
double maxX = 10.0;
int extinction = tMax / 4;
Random rnd = new Random(rndSeed);
```

Local variable tHunger determines when a roach will become hungry and leave its current location and neighbors. For example, if tMax is 10,000 as in the demo, then, when a roach's hunger value reaches tMax / 10 = 1,000, the roach will move to a new position.

Variables minX and maxX set limits on a roach's position vector. The values (-10.0, +10.0) are normal for machine learning weights and also correspond to usual limits for the Rastrigin function. For example, for a problem with dimension = 3, the position vector is an array of three cells, all of which will have values between -10.0 and +10.0.

Local variable extinction determines when all the roaches will die and be reborn at new positions. This mechanism is a restart and helps prevent the algorithm from getting stuck at a non-optimal solution.

Local Random object rnd is used by the algorithm for three purposes. The order in which the roaches are processed is randomized; the exchange of information between neighbor roaches occurs with a certain probability; and there's a random component to each roach's new velocity. Method SolveRastrigin continues:

```
Roach[] herd = new Roach[numRoaches];
for (int i = 0; i < numRoaches; ++i)
  herd[i] = new Roach(dim, minX, maxX, tHunger, i);
```

The collection of simulated roaches is an array named herd. There are all kinds of interesting names for collections of animals, such as a pod of whales and a gaggle of geese. As a matter of fact, a collection of cockroaches is called an intrusion of roaches. (This information could make a useful bar bet for you.)

Notice that the loop index variable, i, is passed to the Roach constructor. The index variable acts as a random seed for the Random object that's part of the Roach class definition. Passing a loop index variable to be used as a random seed value is a common technique in machine learning. The method definition continues with:

```
int t = 0;  // Loop counter (time)
int[] indices = new int[numRoaches];
for (int i = 0; i < numRoaches; ++i)
  indices[i] = i;

double bestError = double.MaxValue;
double[] bestPosition = new double[numRoaches];
int displayInterval = tMax / 20;
```

The array named indices holds values (0, 1 2, . . numRoaches-1). The array will be shuffled in the main processing loop so that the order in which the roaches are processed is different every time. Local variables bestPosition and bestError hold the best position/

Figure 4 **The Roach Algorithm at a High Level**

```
initialize n roaches to random positions
loop tMax times
  compute distances between all roaches
  compute median distance
  for-each roach
    compute number neighbors
    exchange data with neighbors
    if not hungry
      compute new velocity
      compute new position
      check if new best position
    else if hungry
      relocate to new position
    end-if
  end-for
end loop
return best position found
```

SEE THE WORLD AS A DATABASE

ADO.NET ▪ JDBC ▪ ODBC ▪ SQL SSIS ▪ ODATA
MYSQL ▪ EXCEL ▪ POWERSHELL

Work With Relational Data, Not Complex APIs or Services

Whether you are a developer using ADO.NET, JDBC, OData, or MySQL, or a systems integrator working with SQL Server or Biztalk, or even an information worker familiar with ODBC or Excel – our products give you bi-directional access to live data through easy-to-use technologies that you are already familiar with. If you can connect to a database, then you will already know how to connect to Salesforce, SAP, SharePoint, Dynamics CRM, Google Apps, QuickBooks, and much more!

Give RSSBus a try today and see what mean:

visit us online at www.rssbus.com to learn more or download a free trial.

rssbus
INTEGRATION YOUR WAY

solution and associated error found by any roach at any time. Local variable displayInterval determines when a progress message will be displayed to the console. Next, an array-of-arrays style matrix is instantiated to hold the distance between all pairs of roaches:

```
double[][] distances = new double[numRoaches][];
for (int i = 0; i < numRoaches; ++i)
  distances[i] = new double[numRoaches];
```

For example, if distances[0][3] = 7.89 then distances[3][0] is also 7.89 and the distance between roach 0 and roach 3 is 7.89. Note that the redundant data isn't serious because in most cases you won't have a huge number of roaches. Next, the main processing loop starts:

```
while (t < tMax)
{
  if (t > 0 && t % displayInterval == 0) {
    Console.Write("time = " + t.ToString().PadLeft(6));
    Console.WriteLine(" best error = " +
      bestError.ToString("F5"));
  }
...
```

Then the distances between roaches is calculated:

```
for (int i = 0; i < numRoaches - 1; ++i) {
  for (int j = i + 1; j < numRoaches; ++j) {
    double d = Distance(herd[i].position,
                 herd[j].position);
    distances[i][j] = distances[j][i] = d;
  }
}
```

The distance values are calculated using helper method Distance, which will be presented shortly. The array indexing here is a bit tricky but I'll leave it to you to verify if you're curious. Next, the distance values are copied from the distances matrix into an array so they can be sorted and then the median distance can be determined:

```
double[] sortedDists =
  new double[numRoaches * (numRoaches - 1) / 2];
int k = 0;
for (int i = 0; i < numRoaches - 1; ++i) {
  for (int j = i + 1; j < numRoaches; ++j) {
    sortedDists[k++] = distances[i][j];
  }
}
```

The size of the array is best explained by example. Suppose there are n = 4 roaches. Then the distances matrix will have size 4x4. The values on the diagonal, [0][0], [1][1], [2][2] and [3][3] will be 0.0 and shouldn't be included. That leaves the 6 values at [0][1], [0][2], [0][3], [1][2], [1][3] and [2][3]. You don't need the identical distance values at symmetric indices [1][0], [2][0] and so on. So, there are n * (n-1) / 2 distinct distance values. Next, the median distance between roaches is calculated and the roach indices are randomized:

```
Array.Sort(sortedDists);
double medianDist = sortedDists[sortedDists.Length / 4];
Shuffle(indices, t); // t is used as a random seed
```

Here, because I divide by 4, the distance is one-fourth from the beginning of the sorted distances array so the result really isn't a median, it's a quartile. The original research paper used the actual median (by dividing by 2), but I found that the quartile worked better than the median. The idea is that the median or quartile determines how many neighbors a roach has, which in turn influences how closely the roaches are grouped. Using the quartile keeps the roaches further apart, giving them a better chance to find a tricky global minimum value for the target function to minimize.

The roach indices are randomized using helper method Shuffle, which will be presented shortly. Notice that the time index variable, t, is passed to the Shuffle method and acts as a seed for the Shuffle random number generator. Next, the loop to process each roach begins:

```
for (int i = 0; i < numRoaches; ++i)  // Each roach
{
  int idx = indices[i]; // Roach index
  Roach curr = herd[idx]; // Ref to current roach
  int numNeighbors = 0;
...
```

A reference to the current roach, herd[idx], is created and named curr. This is just for convenience. Next, the number of neighbors of the current roach is calculated:

```
for (int j = 0; j < numRoaches; ++j) {
  if (j == idx) continue;
  double d = distances[idx][j];
  if (d < medianDist) // Is a neighbor
    ++numNeighbors;
}
```

The condition j == idx is used to prevent the current roach from being counted as a neighbor to itself. Next, the effective number of neighbors is determined:

```
int effectiveNeighbors = numNeighbors;
if (effectiveNeighbors >= 3)
  effectiveNeighbors = 3;
```

Recall that the purpose of calculating the number of neighbors is to determine the probability that neighbors will exchange information. But the probability of information exchange is the same for 3 or more neighbors. Next, the algorithm determines if information should be exchanged:

```
for (int j = 0; j < numRoaches; ++j) {
  if (j == idx) continue;
  if (effectiveNeighbors == 0) continue;
  double prob = rnd.NextDouble();
  if (prob > A[effectiveNeighbors - 1]) continue;
...
```

The current roach is compared against all other roaches. If the current roach has no neighbors then there's no information exchange. If the current roach has one or more neighbors, the A array of probabilities is used to decide if information should be exchanged or not. Next:

```
double d = distances[idx][j];
if (d < medianDist) { // a neighbor
  if (curr.error < herd[j].error) { // curr better than [j]
    for (int p = 0; p < dim; ++p) {
      herd[j].groupBestPos[p] = curr.personalBestPos[p];
      curr.groupBestPos[p] = curr.personalBestPos[p];
    }
  }
...
```

When information exchange between neighbor roaches occurs, the group best position and associated error of the better of the two roaches is copied to the worse roach. The second branch of the information exchange code is:

```
...
    else { // [j] is better than curr
      for (int p = 0; p < dim; ++p) {
        curr.groupBestPos[p] = herd[j].personalBestPos[p];
        herd[j].groupBestPos[p] = herd[j].personalBestPos[p];
      }
    }
  } // If a neighbor
} // j, each neighbor
```

After information exchange between neighbor roaches is taken care of, the current roach moves if it's not hungry. The first part of the move process is to calculate the new velocity of the current roach:

```
if (curr.hunger < tHunger) {
  for (int p = 0; p < dim; ++p)
    curr.velocity[p] = (C0 * curr.velocity[p]) +
     (C1 * rnd.NextDouble() * (curr.personalBestPos[p] -
       curr.position[p])) +
     (C1 * rnd.NextDouble() * (curr.groupBestPos[p] -
       curr.position[p]));
```

WASHINGTON, DC

JUNE 8-9 2016

ACQUIRE

Acquisition & Management Show

ACQUIRE is a new 2-day event that focuses on 3 key OMB spending categories—**Professional Services**, **Office Management** and **Information Technology**. Covering all aspects of the acquisition and management process, from setting policy and defining requirements to implementing and managing programs to end user experience, it's guaranteed to be **THE NEXT BIG THING.**

Start planning your presence now.
Exhibit & Sponsorship packages available.

Contact Stacy Money for pricing & details:
smoney@1105media.com | 415.444.6933

ACQUIREshow.com

Tracks include:

ACQUIRE
Office Management

ACQUIRE
Professional Services

ACQUIRE
Information Technology
FEDERAL IT ACQUISITION SUMMIT

ACQUIRE
Acquisition Management

ACQUIRE
Project Management

ACQUIRE
Industry Days
Washington Technology

ACQUIRE
Happy Fed
Federal SOUP

The new velocity has three components. The first component is the old velocity, which is sometimes called inertia in particle swarm terminology. Inertia acts to keep a roach moving in the same direction. The second component is the roach's best known position, which is sometimes called the cognitive term. The cognitive component prevents a roach from moving to bad positions. The third component is the best known position of the roach's neighbors. This component is more or less unique to RIO and doesn't have a standard name. This third term acts to keep groups of roaches together.

After the velocity of the current roach has been calculated, the roach is moved:

```
for (int p = 0; p < dim; ++p)
  curr.position[p] = curr.position[p] + curr.velocity[p];

double e = Error(curr.position);
curr.error = e;
```

After the current roach is moved, its new position is checked to see if it's a new best for the roach:

```
if (curr.error < curr.personalBestErr) {
  curr.personalBestErr = curr.error;
  for (int p = 0; p < dim; ++p)
    curr.personalBestPos[p] = curr.position[p];
}
```

Next, the new position is checked to see if it's a new global best, and the hunger counter is incremented:

```
  if (curr.error < bestError) {
    bestError = curr.error;
    for (int p = 0; p < dim; ++p)
      bestPosition[p] = curr.position[p];
  }
  ++curr.hunger;
} // If not hungry
```

The each-roach loop finishes by dealing with hungry roaches:

```
  else { // Roach is hungry
  {
    herd[idx] = new Roach(dim, minX, maxX, tHunger, t);
  }
} // j each roach
```

If a roach's hunger counter reaches the tHunger threshold, the roach moves to a new, random location. After all roaches have been processed, the algorithm finishes by checking if it's time for a global extinction, incrementing the main loop time counter and returning the best position found by any roach:

```
    if (t > 0 && t % extinction == 0) { // Extinction?
      Console.WriteLine("Mass extinction at t = " +
        t.ToString().PadLeft(6));
      for (int i = 0; i < numRoaches; ++i)
        herd[i] = new Roach(dim, minX, maxX, tHunger, i);
    }

    ++t;
  } // Main while loop

  return bestPosition;
} // Solve
```

Notice that the algorithm is contained in a method named SolveRastrigin rather than a more general name such as Solve. The idea here is that RIO is really a meta-heuristic, rather than a prescriptive algorithm, and needs to be customized to whatever minimization problem you're trying to solve.

The Helper Methods

Method SolveRastrigin calls helper methods Distance, Error and Shuffle. Helper method Distance returns the Euclidean distance (square root of the sum of squared term differences):

```
static double Distance(double[] pos1, double[] pos2)
{
  double sum = 0.0;
  for (int i = 0; i < pos1.Length; ++i)
    sum += (pos1[i] - pos2[i]) * (pos1[i] - pos2[i]);
  return Math.Sqrt(sum);
}
```

Based on a few limited experiments I've performed, RIO appears to solve some benchmark problems better than other algorithms, but is weaker on other problems.

Helper method Error returns the squared difference between the calculated value of Rastrigin's function at a given roach position x and the true minimum value of zero:

```
public static double Error(double[] x)
{
  double trueMin = 0.0; double rastrigin = 0.0;
  for (int i = 0; i < x.Length; ++i) {
    double xi = x[i];
    rastrigin += (xi * xi) - (10 * Math.Cos(2 * Math.PI * xi)) + 10;
  }
  return (rastrigin - trueMin) * (rastrigin - trueMin);
}
```

Method Shuffle randomizes the order of the values in an array using the Fisher-Yates mini-algorithm:

```
static void Shuffle(int[] indices, int seed)
{
  Random rnd = new Random(seed);
  for (int i = 0; i < indices.Length; ++i) {
    int r = rnd.Next(i, indices.Length);
    int tmp = indices[r]; indices[r] = indices[i];
    indices[i] = tmp;
  }
}
```

The original research version of RIO doesn't randomize the roach processing order, but I've found that this approach almost always improves the accuracy of bio-inspired optimization algorithms.

A Few Comments

So, just how effective is roach-inspired optimization compared to other numerical optimization techniques? Based on a few limited experiments I've performed, RIO appears to solve some benchmark problems better than other algorithms, but is weaker on other problems. I conclude that while RIO is not a fantastic new general-purpose optimization algorithm, it does have promise and could be useful for certain specific minimization problems. And for sure, roach infestation optimization has one of the most unusual names in all of computer science. ■

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

Thanks to the following Microsoft technical experts for reviewing this article: *Todd Bello, Marciano Moreno Diaz Covarrubias and Alisson Sol*

Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

JOIN US on the CAMPAIGN TRAIL in 2016!

MARCH 7 - 11
BALLY'S HOTEL & CASINO
LAS VEGAS, NV
vslive.com/lasvegas
See pages 44 – 47 for more details

MAY 16 - 19
HYATT AUSTIN, TX
vslive.com/austin
See pages 66 – 67 for more info

JUNE 13 - 16
HYATT CAMBRIDGE, MA
vslive.com/boston
See pages 68 – 69 for more info

AUGUST 8 - 12
MICROSOFT HQ, REDMOND, WA
vslive.com/redmond

SEPTEMBER 26 - 29
HYATT ORANGE COUNTY, CA –
A DISNEYLAND® GOOD
NEIGHBOR HOTEL
vslive.com/anaheim

OCTOBER 3 - 6
RENAISSANCE, WASHINGTON, D.C.
vslive.com/dc

PART OF LIVE! 360

DECEMBER 5 - 9
LOEWS ROYAL PACIFIC
ORLANDO, FL

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive
facebook.com – Search "VSLive"
linkedin.com – Join the "Visual Studio Live" group!

TURN THE PAGE FOR MORE EVENT DETAILS.

YOUR CODE COUNTS
Austin
CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!
Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

AUSTIN 2016 CAMPAIGN FOR CODE

VISUAL STUDIO LIVE! is bringing back its unique brand of practical, unbiased, Developer training to the deep heart of Texas. We're all set to convene in Austin this May, where your code not only counts, it's crucial! From May 16 – 19, we're offering four days of sessions, workshops and networking events—all designed to help you elevate your code-writing abilities to write winning applications across all platforms.

SUPPORTED BY Microsoft Visual Studio msdn magazine Visual Studio MAGAZINE

PRODUCED BY 1105 MEDIA YOUR GROWTH. OUR BUSINESS.

AUSTIN • MAY 16-19, 2016

HYATT REGENCY • AUSTIN, TX

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

TOPICS INCLUDE:

- Visual Studio / .NET
- Windows Client
- Mobile Client
- JavaScript / HTML5 Client
- ASP.NET
- Cloud Computing
- Database & Analytics
- ALM / DevOps
- UX / Design

Get out and code.
Register to join us today!

REGISTER NOW AND SAVE $300!

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

USE PROMO CODE VSLFEB5

VSLIVE.COM/AUSTIN

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!
BETTER
CODE
Boston
FOR ALL
Visual Studio LIVE!
EXPERT SOLUTIONS FOR .NET DEVELOPERS

BOSTON 2016 CAMPAIGN FOR CODE

For the first time in a decade, Boston will host **Visual Studio Live!** from June 13 – 16. Through four intense days of practical, unbiased, Developer training, join us as we dig in to the latest features of Visual Studio 2015, ASP.NET, JavaScript, TypeScript, Windows 10 and so much more. Code with industry experts, get practical answers to your current challenges, and immerse yourself in what's to come on the .NET horizon.

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

PRODUCED BY 1105 MEDIA INC. YOUR GROWTH. OUR BUSINESS.

BOSTON • JUNE 13-16, 2016

HYATT REGENCY • CAMBRIDGE, MA

CONNECT WITH VISUAL STUDIO LIVE!

twitter.com/vslive – @VSLive

facebook.com – Search "VSLive"

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

TOPICS INCLUDE:

- ➤ Visual Studio / .NET
- ➤ Windows Client
- ➤ Mobile Client
- ➤ JavaScript / HTML5 Client
- ➤ ASP.NET
- ➤ Cloud Computing
- ➤ Database & Analytics
- ➤ ALM / DevOps
- ➤ UX / Design

Life, liberty, and the pursuit of better code: register to join us today.

REGISTER NOW AND SAVE $300!

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

USE PROMO CODE VSLFEB5

VSLIVE.COM/BOSTON

How To Be MEAN: Inside MongoDB

Welcome back, "MEANies." (I decided that sounds better than "Nodeists," and, besides, starting with this piece, we're moving past just Node.)

The code has reached an important transition point—now that there are some tests in place to verify that functionality remains the same, it's safe to begin some serious refactoring. In particular, the current in-memory data storage system might be nice for quick demos, but it's not going to scale up particularly well over time. (Not to mention all it takes is one VM reboot/restart, and you've lost all the data that's not hardcoded in the startup code.) It's time to explore the "M" in MEAN: MongoDB.

Hu-MongoDB-ous

First off, it's important to recognize that according to popular lore, MongoDB actually gets its name from the word "humongous." Whether that bit of Internet trivia is true or not, it serves to underscore that MongoDB isn't built to provide the exact same feature set as your average relational database. To Mongo, scalability ranks high, and toward that end, MongoDB is willing to sacrifice a little consistency, trading off ACID transaction capabilities across the cluster in favor of "eventual consistency."

Like most databases, connecting to MongoDB will require a server DNS name or IP address, a database name and (optionally) a port to use.

This system might not ever reach the scale of bazillions of records; in fact, I'd be quite shocked if it ever came within shouting distance of needing it. However, MongoDB has another feature that ranks equally high on the "this is worth exploring" scale of intrigue and that's its data model: It's a document-oriented database. This means that instead of the traditional relational tables-and-columns schema-enforced model, MongoDB uses a data model of "schemaless" documents gathered into collections. These documents are represented in JSON, and as such each document is made up of name-value pairs, where the values can be traditional data types (strings, integers, floating-point values, Booleans and so on), as well as more "composite" data types (arrays of any of the data types just listed, or child objects that in turn can have name-value pairs). This means, offhand, that the data modeling will be a bit different than you might expect if your only experience is with a relational database; this application is small enough now that these differences won't be very overt, but it's something to keep in mind when working with more complex storage needs.

Figure 1 **Creating an Object in MongoDB**

```
// Go get your configuration settings
var config = require('./config.js');
debug("Mongo is available at ",config.mongoServer,":",config.mongoPort);

// Connect to MongoDB
var mongo = null;
var persons = null;
var mongoURL = "mongodb://" + config.mongoServer +
  ":" + config.mongoPort + "/msdn-mean";
debug("Attempting connection to mongo @",mongoURL);
MongoClient.connect(mongoURL, function(err, db) {
  if (err) {
    debug("ERROR:", err);
  }
  else {
    debug("Connected correctly to server");
    mongo = db;
    mongo.collections(function(err, collections) {
      if (err) {
        debug("ERROR:", err);
      }
      else {
        for (var c in collections) {
          debug("Found collection",collections[c]);
        }
        persons = mongo.collection("persons");
      }
    });
  }
});

// Create express instance
var app = express();
app.use(bodyParser.json());

// ...
```

Note: For a deeper look at MongoDB from a .NET developer's perspective, check out this column's 201 three-part series on MongoDB (bit.ly/1J7Dj0B).

Data Design

From a design perspective, seeing how the "persons" data model will map against MongoDB is straightforward: there will be a "persons" collection and each document inside that will be a JSON-based bundle of name-value pairs and so on.

And that's pretty much it. Seriously. This is part of the reason that document-oriented databases are enjoying such favor in the development community—the startup curve to getting data into them is ridiculously low, compared to their schema-based relational counterparts. This has its own drawbacks, too, of course—one typo and suddenly all the queries that are supposed to be based on "firstName" are suddenly coming back empty, because no document has a field "firstName"—but I'll look at a few ways to mitigate some of these later.

For now, let's look at getting some data into and out of MongoDB.

Data Access

The first step is to enable the application to talk to MongoDB; that involves, not surprisingly, installing a new npm package called "mongodb." So, by now, this exercise should seem almost automatic:

```
npm install --save mongodb
```

The npm tool will churn through its usual gyrations and when it returns, the Node.js MongoDB driver is installed into the node_modules directory. If you get a warning from npm about a kerberos package not being installed ("mongodb-core@1.2.28 requires a peer of kerberos@~0.0"), this is a known bug and seems fixable by simply installing kerberos directly via npm ("npm install kerberos"). There shouldn't be any problems beyond that, but of course, this is all subject to the next release of any of these packages—such is the joy of developing on the bleeding edge.

Next, the code will need to open a connection to the MongoDB instance. Where the instance resides, however, deserves a little discussion.

Data Location

As mentioned in the first article in this series, there's two easy options for MongoDB: one is to run it locally, which is great for the development experience but not so good for the production experience; and the other is to run it in the cloud, which is great for the production experience but not for development. (If I can't run the code while I'm on an airplane on my way to a conference, then it's not a great developer experience, in my opinion.) This is not an unusual state of affairs and the solution here is very much the same as it would be for any application: run it locally during development and from the cloud in production or testing.

Like most databases, connecting to MongoDB will require a server DNS name or IP address, a database name and (optionally) a port to use. Normally, in development, this will be "localhost," the database name and "27017" (the MongoDB default), but the settings for a MongoDB instance in the cloud will obviously be different than that. For example, the server and port settings for my Mongolab MongoDB instance called "msdn-mean" are "ds054308.mongolab.com" and "54308," respectively.

The easiest way to capture this divergence in the Node world is to create a standalone JS file (typically called config.js) and require it into the app.js code, like so:

```
// Load modules
var express = require('express'),
  bodyParser = require('body-parser'),
  debug = require('debug')('app'),
  _ = require('lodash');

// Go get your configuration settings
var config = require('./config.js');
debug("Mongo is available at",config.mongoServer,":",config.mongoPort);

// Create express instance
var app = express();
app.use(bodyParser.json());

// ... The rest as before
```

What remains, then, is for the config file to determine the environment in which this application is running; the usual way to do this in the Node.js environment is to examine an environment variable, "ENV," which will be set to one of "prod," "dev," or "test" (if a third, QA-centric, environment is in place). So the config code needs to examine the ENV environment variable and put the right values into the exported module object:

```
// config.js: Configuration determination
//
var debug = require('debug')('config');

debug("Configuring environment...");

// Use these as the default
module.exports = {
  mongoServer : "localhost",
  mongoPort : "27017"
};

if (process.env["ENV"] === "prod") {
  module.exports.mongoServer = "ds054308.mongolab.com";
  module.exports.mongoPort = "54308";
}
```

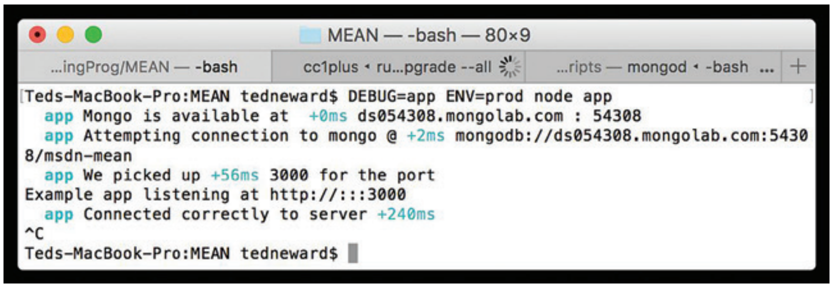
Note the use of the "process" object—this is a standard Node.js object, always implicitly present inside any Node.js-running application, and the "env" property is used to look up the "ENV" environment variable. (Sharp readers will note that the ExpressJS code does exactly the same thing when deciding what port to use; you could probably refactor that snippet to use the config.js settings, as well, but I'll leave that as an exercise to you, the reader.)

So far, so good. Actually, better; this has also implicitly created a nice separation of configuration code away from the main code base.

Let's start adding and removing data.

MongoDB + Node.js

Like most databases, you need to open a connection to MongoDB, hold on to that object, and use that for subsequent actions against the database. Thus, it would seem an obvious first step would be to create that object as the application is starting up and store it globally, as shown in **Figure 1**.



```
Teds-MacBook-Pro:MEAN tedneward$ DEBUG=app ENV=prod node app
  app Mongo is available at  +0ms ds054308.mongolab.com : 54308
  app Attempting connection to mongo @ +2ms mongodb://ds054308.mongolab.com:5430
8/msdn-mean
  app We picked up +56ms 3000 for the port
Example app listening at http://:::3000
  app Connected correctly to server +240ms
^C
Teds-MacBook-Pro:MEAN tedneward$
```

Figure 2 **Debug Printed Output**

Notice that the connect call takes a URL, and a callback—this callback takes an error object and the database connection object as its parameters, as is the Node.js convention. If the first is anything but undefined or null, it's an error, otherwise everything went swimmingly. The URL is a MongoDB-specific URL, using the "mongodb" scheme, but otherwise looking very much like a traditional HTTP URL.

However, there's a subtlety to this code that may not be apparent at first: The callback is invoked at some point well after the rest of the startup code completes, which becomes more obvious when you look at the debug-printed output, as shown in **Figure 2**.

See how the "Example app listening" message appears before the "Connected correctly to server" message from the callback? Given that this is happening on application startup, this concurrency issue isn't critical, but it's not going away, and this is, without question, one of the trickiest parts of working with Node.js. It's true that your Node.js code will never be executed simultaneously on two threads at the same time, but that doesn't mean you won't have some interesting concurrency problems going on here; they just look different than what you're used to as a .NET developer.

Also, just as a quick reminder, when this code is first run against a brand-new MongoDB database, the collections loop will be empty—MongoDB won't create the collections (or even the database!) until it absolutely has to, which usually occurs when somebody writes to it. Once an insert is done, then MongoDB will create the necessary artifacts and data structures to store the data.

Regardless, for the moment, we have a database connection. Time to update the CRUD methods to start using it.

Insert

The insertPerson will use the insert method on the MongoDB collection object and, again, you need a callback to invoke with the results of the database operation:

```
var insertPerson = function(req, res) {
  var person = req.body;
  debug("Received", person);
  // person.id = personData.length + 1;
  // personData.push(person);
  persons.insert(person, function(err, result) {
    if (err)
      res.status(500).jsonp(err);
    else
      res.status(200).jsonp(person);
  });
};
```

Notice the commented-out code (from the in-memory database version I'm migrating away from); I left it there specifically to prove a point. MongoDB will create an identifier field, "_id," that's the primary key for the document in the database, so my incredibly lame homegrown "id" generator code is not only no longer necessary, but entirely unwanted.

Also, notice that the last statement in the function is the insert method, with the associated callback. While it isn't necessary that this be the last statement in the function block, it's critical to understand that the insertPerson function will terminate before the database insert completes. The callback-based nature of Node.js is such that you don't want to return anything to the caller until you know the success or failure of the database operation—hence the calls to "res" don't happen anywhere outside the callback. (Skeptics should convince themselves of this by putting a debug call after the persons.insert call, and another one in the callback itself, and see the first one fire before the callback does.)

Retrieve All

Inserts require validation, so while I'm here, I'll refactor getAllPersons, which just needs a quick query to the collection to find all the documents in that collection:

```
var getAllPersons = function(req, res) {
  persons.find({}).toArray(function(err, results) {
    if (err) {
      debug("getAllPersons--ERROR:",err);
      res.status(500).jsonp(err);
    }
    else {
      debug("getAllPersons:", results);
      res.status(200).jsonp(results);
    }
  });
};
```

Before moving on, there's a couple of quick things to note: First, the find call takes a predicate document describing the criteria by which you want to query the collection, which in this case, I leave as empty; if this were a query by first name, that predicate document would need to look something like:

```
"{ 'firstName':'Ted' }"
```

Second, notice that the returned object from find isn't an actual result set yet, hence the need to call toArray to convert it into something of use. The toArray takes a callback and, again, each branch of the callback must make sure to communicate something back to the caller using res.status().jsonp.

Middleware

Before I can go on, recall from my previous columns that the getPerson, updatePerson and deletePerson functions all depend on the personId middleware function to look up a person by identifier. This means that that middleware needs to be updated to query the collection by its _id field (which is a MongoDB ObjectID, not a string!), instead of looking in the in-memory array, as shown in **Figure 3**.

The MongoDB Node.js driver documents a findOne method, which would seem to be more appropriate, but the driver documentation notes that as a deprecated method.

Figure 3 **Updating Middleware to Query the Collection**

```
app.param('personId', function (req, res, next, personId) {
  debug("personId found:",personId);
  if (mongodb.ObjectId.isValid(personId)) {
    persons.find({"_id":new mongodb.ObjectId(personId)})
      .toArray(function(err, docs){
        if (err) {
          debug("ERROR: personId:",err);
          res.status(500).jsonp(err);
        }
        else if (docs.length < 1) {
          res.status(404).jsonp(
            { message: 'ID ' + personId + ' not found'});
        }
        else {
          debug("person:", docs[0]);
          req.person = docs[0];
          next();
        }
      });
  }
  else {
    res.status(404).jsonp({ message: 'ID ' + personId + ' not found'});
  }
});
```

Notice that the middleware, if it gets an invalid ObjectId, doesn't call next. This is a quick way to save some lines of code in the various methods that are depending on finding persons from the database, because if it's not a legitimate ID, it can't possibly be there, so hand back a 404. The same is true if the results have zero documents (meaning that ID wasn't in the database).

Retrieve One, Delete and Update

Thus, the middleware makes getPerson trivial, because it handles all the possible error or document-not-found conditions:

```
var getPerson = function(req, res) {
  res.status(200).jsonp(req.person);
};
```

And deletePerson is almost as trivial:

```
var deletePerson = function(req, res) {
  debug("Removing", req.person.firstName, req.person.lastName);
  persons.deleteOne({"_id":req.person._id}, function(err, result) {
    if (err) {
      debug("deletePerson: ERROR:", err);
      res.status(500).jsonp(err);
    }
    else {
      res.person._id = undefined;
      res.status(200).jsonp(req.person);
    }
  });
};
```

Both of which make updatePerson pretty predictable:

```
var updatePerson = function(req, res) {
  debug("Updating",req.person,"with",req.body);

  _.merge(req.person, req.body);

    persons.updateOne({"_id":req.person._id}, req.person, function(err, result) {
    if (err)
      res.status(500).jsonp(err);
    else {
      res.status(200).jsonp(result);
    }
  });
};
```

The merge call, by the way, is the same Lodash function used before to copy the properties from the request body over to the person object that was loaded out of the database.

Wrapping Up

Wow. This has been a little heavier than some of the others in the series, but at this point, I have code that completely runs now against a MongoDB database, instead of the in-memory array I'd been using during the mocking out. But it's not perfect, not by a long shot. For starters, any typos in the code around those query predicates will create unanticipated runtime errors. More important, as .NET developers, we're accustomed to some kind of "domain object" to work with, particularly if there's some sort of validation on the various properties of the object that need to be done—it's not a good idea to spread that validation code throughout the Express parts of the code base. That's on the docket for next time. But for now … happy coding! ■

Ted Neward *is the CTO of iTrellis, a Seattle-based polytechnology consulting firm. He has written more than 100 articles, is an F# MVP, INETA speaker, and has authored or co-authored a dozen books. Reach him at ted@tedneward.com if you're interested in having him come work with your team, or read his blog at blogs.tedneward.com.*

Thanks to the following technical expert for reviewing this article: *Shawn Wildermuth*

dtSearch®

Instantly Search Terabytes of Text

dtSearch's document filters support popular file types, emails with multilevel attachments, databases, web data

Highlights hits in all data types; 25+ search options

With APIs for .NET, Java and C++. SDKs for multiple platforms. (See site for articles on faceted search, SQL, MS Azure, etc.)

Visit dtSearch.com for
- hundreds of reviews and case studies
- fully-functional enterprise and developer evaluations

The Smart Choice for Text Retrieval® since 1991

dtSearch.com 1-800-IT-FINDS

MARK MICHAELIS

Configuration in .NET Core

Those of you working with ASP.NET 5 have no doubt noticed the new configuration support included in that platform and available in the Microsoft.Extensions.Configuration collection of NuGet packages. The new configuration allows a list of name-value pairs, which can be grouped into a multi-level hierarchy. For example, you can have a setting stored in SampleApp:Users:InigoMontoya:MaximizeMainWindow and another stored in SampleApp:AllUsers:Default:MaximizeMainWindow. Any stored value maps to a string, and there's built-in binding support that allows you to deserialize settings into a custom POCO object. Those of you already familiar with the new configuration API probably first encountered it within ASP.NET 5. However, the API is in no way restricted to ASP.NET. In fact, all the listings in this article were created in a Visual Studio 2015 Unit Testing project with the Microsoft .NET Framework 4.5.1, referencing Microsoft.Extensions.Configuration packages from ASP.NET 5 RC1. (Go to GitHub.com/IntelliTect/Articles for the source code.)

The configuration API supports configuration providers for in-memory .NET objects, INI files, JSON files, XML files, command-line arguments, environment variables, an encrypted user store, and any custom provider you create. If you wish to leverage JSON files for your configuration, just add the Microsoft.Extensions.Configuration.Json NuGet package. Then, if you want to allow the command line to provide configuration information, simply add the Microsoft.Extensions.Configuration.CommandLine NuGet package, either in addition to or instead of other configuration references. If none of the built-in configuration providers are satisfactory, you're free to create your own by implementing the interfaces found in Microsoft.Extensions.Configuration.Abstractions.

Retrieving Configuration Settings

To familiarize yourself with retrieving configuration settings, take a look at **Figure 1**.

Accessing the configuration begins easily with an instance of the ConfigurationBuilder, a class available from the Microsoft.Extensions.Configuration NuGet package. Given the ConfigurationBuilder instance, you can add providers directly using IConfigurationBuilder extension methods like AddInMemoryCollection, as shown in **Figure 1**. This method takes a Dictionary<string,string> instance of the configuration name-value pairs, which it uses to initialize the configuration provider before adding it to the ConifigurationBuilder instance. Once the configuration builder is "configured," you invoke its Build method to retrieve the configuration.

Code download available at GitHub.com/IntelliTect/Articles.

As mentioned earlier, a configuration is simply a hierarchical list of name-value pairs in which the nodes are separated by a colon. Therefore, to retrieve a particular value, you simply access the Configuration indexer with the corresponding item's key:

```
Console.WriteLine($"Hello {Configuration["Profile:UserName"]}");
```

However, accessing a value isn't limited to only retrieving strings. You can, for example, retrieve values via the ConfigurationBinder's Get<T> extension methods. For instance, to retrieve the main window screen buffer size you can use:

```
Configuration.Get<int>("AppConfiguration:MainWindow:ScreenBufferSize", 80);
```

This binding support requires a reference to the Microsoft.Extensions.Configuration.Binder NuGet package.

Figure 1 **Configuration Basics Using the InMemoryConfigurationProvider and ConfigurationBinder Extension Methods**

```
public class Program
{
  static public string DefaultConnectionString { get; } =
    @"Server=(localdb)\\mssqllocaldb;Database=SampleData-0B3B0919-C8B3-481C-9833-
    36C21776A565;Trusted_Connection=True;MultipleActiveResultSets=true";

  static IReadOnlyDictionary<string, string>
DefaultConfigurationStrings{get;} =
    new Dictionary<string, string>()
    {
      ["Profile:UserName"] = Environment.UserName,
      [$"AppConfiguration:ConnectionString"] = DefaultConnectionString,
      [$"AppConfiguration:MainWindow:Height"] = "400",
      [$"AppConfiguration:MainWindow:Width"] = "600",
      [$"AppConfiguration:MainWindow:Top"] = "0",
      [$"AppConfiguration:MainWindow:Left"] = "0",
    };

  static public IConfiguration Configuration { get; set; }

  public static void Main(string[] args = null)
  {
    ConfigurationBuilder configurationBuilder =
      new ConfigurationBuilder();


      // Add defaultConfigurationStrings
      configurationBuilder.AddInMemoryCollection(
        DefaultConfigurationStrings);
      Configuration = configurationBuilder.Build();

      Console.WriteLine($"Hello {Configuration["Profile:UserName"]}");

      ConsoleWindow consoleWindow =
        Configuration.Get<ConsoleWindow>("AppConfiguration:MainWindow");
      ConsoleWindow.SetConsoleWindow(consoleWindow);
  }
}
```

Figure 2 **A Sample Configuration Object Graph**

```
class AppConfiguration
{
  public ProfileConfiguration Profile { get; set; }
   public string ConnectionString { get; set; }

  public WindowConfiguration MainWindow { get; set; }

  public class WindowConfiguration
  {
    public int Height { get; set; }
    public int Width { get; set; }
    public int Left { get; set; }
    public int Top { get; set; }
  }

  public class ProfileConfiguration
  {
    public string UserName { get; set; }
  }
}
public static void Main()
{
  // ...
  AppConfiguration appConfiguration =
    Program.Configuration.Get<AppConfiguration>(
      nameof(AppConfiguration));

  // Requires referencing System.Diagnostics.TraceSource in Corefx
  System.Diagnostics.Trace.Assert(
    600 == appConfiguration.MainWindow.Width);
}
```

Notice there's an optional argument following the key, for which you can specify a default value to return when the key doesn't exist. (Without the default value, the return will be assigned default(T), rather than throw an exception as you might expect.)

Configuration values are not limited to scalars. You can retrieve POCO objects or even entire object graphs. To retrieve an instance of the ConsoleWindow whose members map to the AppConfiguration:MainWindow configuration section, **Figure 1** uses:

```
ConsoleWindow consoleWindow =
  Configuration.Get<ConsoleWindow>("AppConfiguration:MainWindow")
```

Alternatively, you could define a configuration graph such as AppConfiguration, shown in **Figure 2**.

With such an object graph, you could define all or part of your configuration with a strongly typed object hierarchy that you can then use to retrieve your settings all at once.

Multiple Configuration Providers

The InMemoryConfigurationProvider is effective for storing default values or possibly calculated values. However, with only that provider, you're left with the burden of retrieving the configuration and loading it into a Dictionary<string,string> before registering it with the ConfigurationBuilder. Fortunately, there are several more built-in configuration providers, including three file-based providers (XmlConfigurationProvider, IniConfigurationProvider and JsonConfigurationProvider); an environment variable provider (EnvironmentVariableConfigurationProvider); and a command-line argument provider (CommandLineConfigurationProvider). Furthermore, these providers can be mixed and matched to suit your application logic. Imagine, for example, that you might specify configuration settings in the following ascending priority:

- InMemoryConfigurationProvider
- JsonFileConfigurationProvider for Config.json
- JsonFileConfigurationProvider for Config.Production.json
- EnvironmentVariableConfigurationProvider
- CommandLineConfigurationProvider

In other words, the default configuration values might be stored in code. Next, the config.json file followed by the Config.Production.json might override the InMemory specified values—where later providers like the JSON ones take precedence for any overlapping values. Next, when deploying, you may have custom configuration values stored in environment variables. For example, rather than hardcoding Config.Production.json, you might retrieve the environment setting from a Windows environment variable and access the specific file (perhaps Config.Test.Json) that the environment variable identifies. (Excuse the ambiguity in the term environment setting relating to production, test, pre-production or development, versus Windows environment variables such as %USERNAME% or %USERDOMAIN%.) Finally, you specify (or override) any earlier provided settings via the command line—perhaps as a onetime change to, for example, turn on logging.

To specify each of the providers, add them to the configuration builder (via the extension method AddX fluent API), as shown in **Figure 3**.

Figure 3 **Adding Multiple Configuration Providers—the Last One Specified Takes Precedence**

```
public static void Main(string[] args = null)
{
  ConfigurationBuilder configurationBuilder =
    new ConfigurationBuilder();


  configurationBuilder
    .AddInMemoryCollection(DefaultConfigurationStrings)
    .AddJsonFile("Config.json",
      true) // Bool indicates file is optional
    // "EssentialDotNetConfiguartion" is an optional prefix for all
    // environment configuration keys, but once used,
    // only environment variables with that prefix will be found
    .AddEnvironmentVariables("EssentialDotNetConfiguration")
    .AddCommandLine(
      args, GetSwitchMappings(DefaultConfigurationStrings));

  Console.WriteLine($"Hello {Configuration["Profile:UserName"]}");

  AppConfiguration appConfiguration =
    Configuration.Get<AppConfiguration>(nameof(AppConfiguration));
}

static public Dictionary<string,string> GetSwitchMappings(
  IReadOnlyDictionary<string, string> configurationStrings)
{
  return configurationStrings.Select(item =>
    new KeyValuePair<string, string>(
      "-" + item.Key.Substring(item.Key.LastIndexOf(':')+1),
      item.Key))
      .ToDictionary(
        item => item.Key, item=>item.Value);
}
```

For the JsonConfigurationProvider, you can either require the file to exist or make it optional; hence the additional optional parameter on AddJsonFile. If no parameter is provided, the file is required and a System.IO.FileNotFoundException will fire if it isn't found. Given the hierarchical nature of JSON, the configuration fits very well into the configuration API (see **Figure 4**).

The CommandLineConfigurationProvider requires you to specify the arguments when it's registered with the configuration builder. Arguments are specified by a string array of name-value pairs, with each pair of the format /<name>=<value>, in which the equals sign is required. The leading slash is also required but the second parameter of the AddCommandLine(string[] args, Dictionary<string,string> switchMappings), function allows you to provide aliases that must be prefixed with either a - or --. For example, a dictionary of values will allow a command line of "program.exe -LogFile="c:\program-data\Application Data\Program.txt" to load into the AppConfiguration:LogFile configuration element:

```
["-DBConnectionString"]="AppConfiguration:ConnectionString",
  ["-LogFile"]="AppConfiguration:LogFile"
```

Before finishing off the configuration basics, here are a few additional points to note:

- The CommandLineConfigurationProvider has several characteristics that are not intuitive from IntelliSense of which you need to be aware:
    - The CommandLineConfigurationProvider's switchMappings only allows a switch prefix of - or --. Even a slash (/) isn't allowed as a switch parameter. This prevents you from providing aliases for slash switches via switch mappings.
    - CommandLineConfigurationProviders doesn't allow for switch-based command-line arguments—arguments that don't include an assigned value. Specifying a key of "/Maximize," for example, isn't allowed.
    - While you can pass Main's args to a new CommandLineConfigurationProvider instance, you can't pass Environment.GetCommandLineArgs without first removing the process name. (Note that Environment.GetCommandLineArgs behaves differently when a debugger is attached. Specifically, executable names with spaces are split into individual arguments when there's no debugger attached. See itl.ty\GetCommandLineGotchas).
    - An exception will be issued when you specify a command-line switch prefix of - or -- for which there's no corresponding switch mapping.
- Although configurations can be updated (Configuration["-Profile:UserName"]="Inigo Montoya"), the updated value is not persisted back into the original store. For example, when you assign a JSON provider configuration value, the JSON file won't be updated. Similarly, an environment variable wouldn't get updated when its configuration item is assigned.
- The EnvironmentVariableConfigurationProvider optionally allows for a key prefix to be specified. In such cases, it will load only those environment variables with the specified prefix.

Figure 4 **JSON Configuration Data for the JsonConfigurationProvider**

```
{
  "AppConfiguration": {
    "MainWindow": {
      "Height": "400",
      "Width": "600",
      "Top": "0",
      "Left": "0"
    },
    "ConnectionString":
      "Server=(localdb)\\\\mssqllocaldb;Database=Database-0B3B0919-C8B3-481C-9833-
      36C21776A565;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

  In this way, you can automatically limit the configuration entries to those within an environment variable "section" or, more broadly, those that are relevant to your application.
- Environment variables with a colon delimiter are supported. For example, assigning SET AppConfiguration:ConnectionString=Console on the command line is allowed.
- All configuration keys (names) are case-insensitive.
- Each provider is located in its own NuGet package where the NuGet package name corresponds to the provider: Microsoft.Extensions.Configuration.CommandLine, Microsoft.Extensions.Configuration.EnvironmentVariables, Microsoft.Extensions.Configuration.Ini, Microsoft.Extensions.Configuration.Json and Microsoft.Extensions.Configuration.Xml.

Understanding the Object-Oriented Structure

Both the modularity and the object-oriented structure of the configuration API are well thought out—providing discoverable, modular and easily extensible classes and interfaces with which to work (see **Figure 5**).

Each type of configuration mechanism has a corresponding configuration provider class that implements IConfigurationProvider. In the majority of built-in provider implementations, the implementation is jump-started by deriving from ConfigurationBuilder rather than using custom implementations for all of the interface methods. Perhaps surprisingly, there's no direct reference to any of the providers in **Figure 1**. This is because instead of manually instantiating each provider and registering it with the ConfigurationBuilder class's Add method, each provider's NuGet pacakge includes a static extension class with IConfigurationBuilder extension methods. (The name of the extension class is generally identified by the suffix ConfigurationExtensions.) With the extension classes, you can start accessing the configuration data directly from ConfigurationBuilder (which implements IConfigurationBuilder) and directly call the extension method associated with your provider. For example, the JasonConfigurationExtensions class adds AddJsonFile extension methods to IConfigurationBuilder so that you can add the JSON configuration with a call to ConfigurationBuilder.AddJsonFile(fileName, optional).Build();.

For the most part, once you have a configuration, you have all you need to start retrieving values.

IConfiguration includes a string indexer, allowing you to retrieve any particular configuration value using the key to access the element for which you're looking. You can retrieve an entire set of settings (called a section) with the GetSection or GetChildren methods (depending on whether you want to drill down an additional level in the hierarchy). Note that configuration element sections allow you to retrieve the following:

- key: the last element of the name.
- path: the full name pointing from the root to the current location.
- value: the configuration value stored in the configuration setting.
- value as an object: via the ConfigurationBinder, you can retrieve a POCO object that corresponds to the configuration section you're accessing (and potentially its children). This is how

TECHMENTOR

IN-DEPTH TRAINING FOR IT PROS

MARCH 7 – 11, 2016
BALLY'S HOTEL & CASINO, LAS VEGAS, NV

REAL TOOLS FOR TODAY'S IT CHALLENGES

Join us for TechMentor, March 7 - 11, 2016, as we make our return to fabulous Las Vegas, NV.

What sets TechMentor apart, and makes it a viable alternative to huge, first-party conferences, is the immediately usable IT education, providing the tools you need today while preparing you for tomorrow. Zero marketing-speak, a strong emphasis on doing more with the technology you already own, and solid coverage of what's just around the corner.

HOT TOPICS COVERED:

- PowerShell
- Security
- DSC
- Hyper-V
- DevOps
- Windows Server
- Azure
- And More!

TRACK TOPICS INCLUDE:

- DataCenter
- Client Devices
- Dev Ops
- IT Soft Skills
- The Real Cloud

SAVE $300

REGISTER BEFORE FEBRUARY 10

USE PROMO CODE TMFEB1

TECHMENTOREVENTS.COM/LASVEGAS

the Configuration.Get<AppConfiguration>(nameof(AppConfiguration)) works in **Figure 3**, for example.

- IConfigurationRoot includes a Reload function that allows you to reload values in order to update the configuration. ConfigurationRoot (which implements IConfigurationRoot) includes a GetReloadToken method that lets you register for notifications of when a reload occurs (and the value might change).

Encrypted Settings

On occasion, you'll want to retrieve settings that are encrypted rather than stored in open text. This is important, for example, when you're storing OAuth application keys or tokens or storing credentials for a database connection string. Fortunately, the Microsoft.Extensions.Configuration system has built-in support for reading encrypted values. To access the secure store, you need to add a reference to the Microsoft.Extensions.Configuration.UserSecrets NuGet package. Once it's added, you'll have a new IConfigurationBuilder.AddUserSecrets extension method that takes a configuration item string argument called userSecretsId (stored in your project.json file). As you'd expect, once the UserSecrets configuration is added to your configuration builder, you can begin retrieving encrypted values, which only users with whom the settings are associated can access.

Obviously, retrieving settings is somewhat pointless if you can't also set them. To do this, use the user-secret.cmd tool as follows:

```
user-secret set <secretName> <value> [--project <projectPath>]
```

The --project option allows you to associate the setting with the userSecretsId value stored in your project.json file (created by default by the ASP.NET 5 new project wizard). If you don't have the user-secret tool, you'll need to add it via the developer command prompt using the DNX utility (currently dnu.exe).

For more information on the user secret configuration option, see the article, "Safe Storage of Application Secrets," by Rick Anderson and David Roth at bit.ly/1mmnG0L.

Wrapping Up

Those of you who have been with .NET for some time have likely been disappointed with the built-in support for configuration via System.Configuration. This is probably especially true if you're coming from classic ASP.NET, where configuration was limited to Web.Config or App.config files and then only by accessing the AppSettings node within that. Fortunately, the new open source Microsoft.Extensions.Configuration API goes well beyond what was originally available by adding a multitude of new configuration providers, along with an easily extensible system into which you can plug any custom provider you want. For those still living (stuck?) in a pre-ASP.NET 5 world, the old System.Configuration APIs still function, but you can slowly begin to migrate (even side-by-side) to the new API just by referencing the new packages. Furthermore, the NuGet packages can be used from Windows client projects like console and Windows Presentation Foundation applications. Therefore, the next time you need to access configuration data, there's little reason not to leverage the Microsoft.Extensions.Configuration API. ■

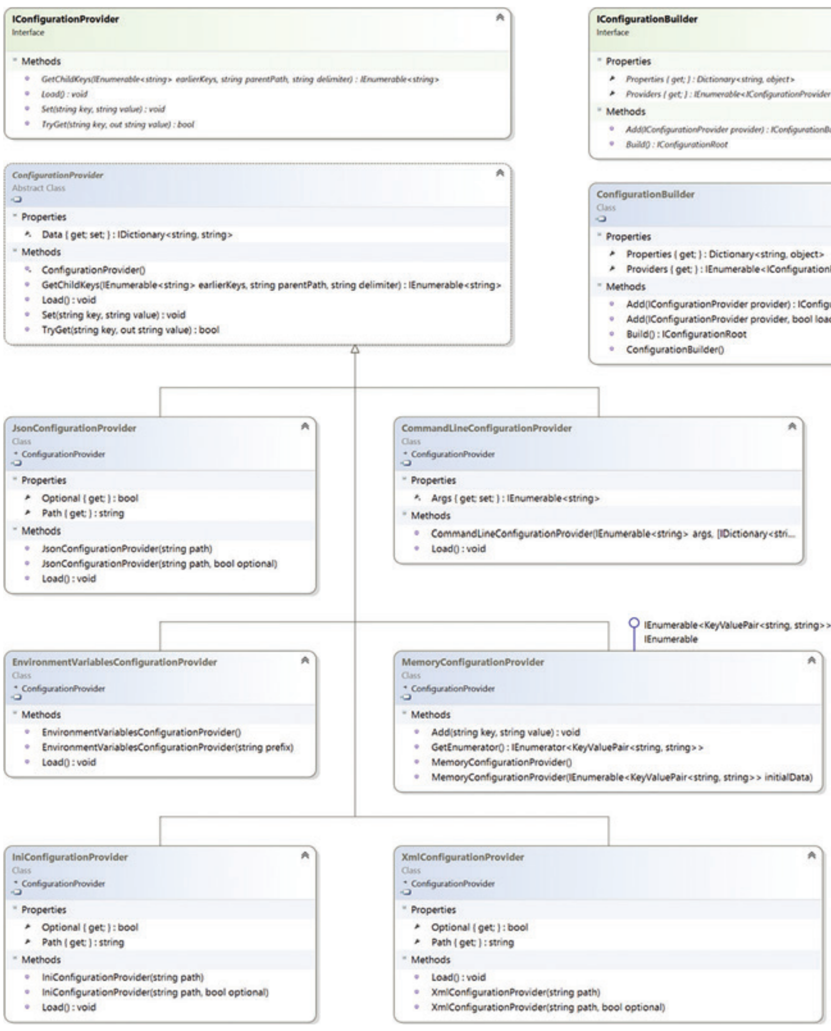

IConfigurationProvider
Interface
Methods
GetChildKeys(IEnumerable<string> earlierKeys, string parentPath, string delimiter) : IEnumerable<string>
Load() : void
Set(string key, string value) : void
TryGet(string key, out string value) : bool
IConfigurationBuilder
Interface
Properties
Properties { get; } : Dictionary<string, object>
Providers { get; } : IEnumerable<IConfigurationProvider>
Methods
Add(IConfigurationProvider provider) : IConfigurationBuilder
Build() : IConfigurationRoot
ConfigurationProvider
Abstract Class
Properties
Data { get; set; } : IDictionary<string, string>
Methods
ConfigurationProvider()
GetChildKeys(IEnumerable<string> earlierKeys, string parentPath, string delimiter) : IEnumerable<string>
Load() : void
Set(string key, string value) : void
TryGet(string key, out string value) : bool
ConfigurationBuilder
Class
Properties
Properties { get; } : Dictionary<string, object>
Providers { get; } : IEnumerable<IConfigurationProvider>
Methods
Add(IConfigurationProvider provider) : IConfigurationBuilder
Add(IConfigurationProvider provider, bool load) : IConfigurationBuilder
Build() : IConfigurationRoot
ConfigurationBuilder()
JsonConfigurationProvider
Class
ConfigurationProvider
Properties
Optional { get; } : bool
Path { get; } : string
Methods
JsonConfigurationProvider(string path)
JsonConfigurationProvider(string path, bool optional)
Load() : void
CommandLineConfigurationProvider
Class
ConfigurationProvider
Properties
Args { get; set; } : IEnumerable<string>
Methods
CommandLineConfigurationProvider(IEnumerable<string> args, [IDictionary<stri...
Load() : void
IEnumerable<KeyValuePair<string, string>>
IEnumerable
EnvironmentVariablesConfigurationProvider
Class
ConfigurationProvider
Methods
EnvironmentVariablesConfigurationProvider()
EnvironmentVariablesConfigurationProvider(string prefix)
Load() : void
MemoryConfigurationProvider
Class
ConfigurationProvider
Methods
Add(string key, string value) : void
GetEnumerator() : IEnumerator<KeyValuePair<string, string>>
MemoryConfigurationProvider()
MemoryConfigurationProvider(IEnumerable<KeyValuePair<string, string>> initialData)
IniConfigurationProvider
Class
ConfigurationProvider
Properties
Optional { get; } : bool
Path { get; } : string
Methods
IniConfigurationProvider(string path)
IniConfigurationProvider(string path, bool optional)
Load() : void
XmlConfigurationProvider
Class
ConfigurationProvider
Properties
Optional { get; } : bool
Path { get; } : string
Methods
Load() : void
XmlConfigurationProvider(string path)
XmlConfigurationProvider(string path, bool optional)

Figure 5 **Configuration Provider Class Model**

Mark Michaelis *is founder of IntelliTect, where he serves as its chief technical architect and trainer. For nearly two decades he has been a Microsoft MVP, and a Microsoft Regional Director since 2007. Michaelis serves on several Microsoft software design review teams, including C#, Microsoft Azure, SharePoint and Visual Studio ALM. He speaks at developer conferences and has written numerous books including his most recent, "Essential C# 6.0 (5th Edition)" (itl.tc/EssentialCSharp). Contact him on Facebook at facebook.com/Mark.Michaelis, on his blog at IntelliTect.com/Mark, on Twitter: @markmichaelis or via e-mail at mark@IntelliTect.com.*

Thanks to the following IntelliTect technical experts for reviewing this article: *Grant Erickson, Derek Howard, Phil Spokas and Michael Stokesbary*

VB6: Waking a Sleeping Giant

This issue begins my seventh year stirring up trouble in this space. To celebrate the occasion, I'm going to kick over my all-time favorite hornets' nest: the developers who continue to love Visual Basic 6, and those who love to hate it and them.

I've written twice (msdn.com/magazine/jj133828 and msdn.com/magazine/dn745870) about the unique place VB6 occupies in today's software development world, likening it to a cockroach, a bus and a knuckleball. I got more responses to these columns than anything else I've ever written. Today I'm going to exceed my usual pouring of oil on troubled fires. I'm about to set off a nuclear explosion, the radiation from which will mutate VB6 into immortality. Don't believe me? Read on, my friend.

I was having lunch with a client some weeks ago. He has a Silverlight-based solution that displays video from security cameras. But Microsoft has now deprecated Silverlight, encouraging developers to switch to HTML5 instead. "That's a pain in the ass," complained my client. "I was doing fine with what I had. Now I have to go learn another language and migrate all my code. My app isn't all that complicated, just a few video streams and some buttons. I wish there was some way to make that really easy."

Then it hit me: This is exactly what VB6 does with its current target of unmanaged Windows apps. How about we develop a version of VB6 that produces HTML5? The output would then run in any browser, on any OS, on any platform, desktop or mobile.

I wrote last May (msdn.com/magazine/dn973019) about the smallpox virus and how it managed to jump hosts—from human to computer—just before the last of the virus died in its final human victim. Here's the chance for VB to jump from its unmanaged Windows host and burst out and infect the entire software world; to do what Java promised and never delivered—writing code once, running absolutely everywhere. A *true* Universal app. From humble, old VB6. Who'd-a thunk it? Because it will go anywhere, I'll call the new language VB*.

The VB* programming model would conceptually resemble the ASP.NET Web Forms model, in which controls render their content as HTML. But that rendering requires ASP.NET on the server side, and VB* needs to avoid depending on any particular server. Therefore, VB* will compile down to independent pages of HTML5 elements and JavaScript code, just as VB6 compiles down to x86 assembler with Windows function calls. You'll be able to slap the page onto any HTML5 server and access it from any HTML5 client.

VB* will use the VB6 ultra-simple syntax and organization. We'll deliberately omit sophisticated functionality in return for easiest programming of simple cases. We probably won't, for example, surface threads into the VB* language. If it turns out that our VB* apps need background operations, some hot-shot programmer will write a background operation control that handles all those grotty details, as happened in VB6.

Therefore, VB* will need a design that supports two tiers of developers—the *uber*-geeks who write the controls, and the application programmers who consume them. We've done that twice before, with VBX controls and then OCX controls, so we can doubtless do it again.

Then it hit me: This is exactly what VB6 does with its current target of unmanaged Windows apps. How about we develop a version of VB6 that produces HTML5?

How could we develop and finance and release and support VB*? Open source? Maybe some tool vendor wants to take it on? How about a consortium? I'd love to help, for a fee of course. (Student: "Plattski, is it true you're a cynical, mercenary bastard who's only in this for the money?" Me: "How much will you pay me if I tell you?") I've even snagged the Web address vbstar.org to get things started.

I can hear the VB haters tuning up their chorus now: "It's not a real language! They're not real programmers! You should be shot for even suggesting it. VB* will never be able to do [this], or [that], or [the other]."

Maybe it won't. But as I wrote previously, "... the rapid (and therefore cheaper) development of limited (and therefore cheaper) applications by lower-skilled (and therefore cheaper) personnel is an important solution to a very large class of problems." If it's done right, VB* will become that solution.

Long live VB*! ■

DAVID S. PLATT *teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.*

Build Experiences
for Your Users with Enterprise-level Developer Tools and Solutions

Spread
Versatile Spreadsheet
Data and UI Components
spread.grapecity.com

Wijmo
A New Generation of
JavaScript Controls
wijmo.com

Xuni
Cross-Platform Native
Mobile UI Controls
goxuni.com

ComponentOne Studio
.NET Controls for Serious
Application Development
componentone.com

ActiveReports
Reporting Platform for
Essential Business Needs
activereports.grapecity.com

The GrapeCity family of products provides developers, designers, and architects the ultimate collection of easy-to-use tools for building sleek, high-performing, feature complete applications. With over 25 years of experience, we understand your needs and have developed a comprehensive line of products that includes innovative UI controls; cross-platform data visualization controls; and enterprise-level reporting, analysis, and spreadsheet controls for Windows, web, and mobile platforms.

Learn more and receive free 30-day trials at
tools.grapecity.com

GrapeCity®

Visual Studio
Microsoft
Partner