

# msdn magazine



ASP.NET Core: Better Living Through Middleware.....10

## When Only the Best Will Do

Our high-performance and feature-complete UI Controls and Libraries will help you build your best, without limits or compromise



 **DevExpress®**

Free 30-day Trial  
[devexpress.com/try](http://devexpress.com/try)

# 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](http://devexpress.com/try)**

# msdn

magazine



**ASP.NET Core: Better Living Through Middleware.....10**

Use Custom Middleware to Detect and Fix 404s in ASP.NET Core Apps <b>Steve Smith</b> .....	<b>10</b>
Scale Asynchronous Client-Server Links with Reactive <b>Peter Vogel</b> .....	<b>16</b>
Language-Agnostic Code Generation with Roslyn <b>Alessandro Del Sole</b> .....	<b>22</b>
Microsoft Azure Media Services and Power BI <b>Sagar Bhanudas Joshi</b> .....	<b>30</b>
Using Azure App Services to Convert a Web Page to PDF <b>Benjamin Perkins</b> .....	<b>36</b>
Speed Up Your Mobile Development Using an MBaaS Platform <b>Paras Wadehra</b> .....	<b>48</b>

## COLUMNS

### CUTTING EDGE

Building an Historical CRUD, Part 2  
Dino Esposito, page 6

### TEST RUN

Introduction to Prediction Markets  
James McCaffrey, page 54

### THE WORKING PROGRAMMER

How To Be MEAN: Passport  
Ted Neward, page 62

### ESSENTIAL .NET

Dependency Injection with .NET Core  
Mark Michaelis, page 66

### MODERN APPS

Playing with Audio in the UWP  
Frank La Vigne, page 74

### DON'T GET ME STARTED

The Joy of UX  
David Platt, page 80

# ASP.NET MVC REPORTING

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

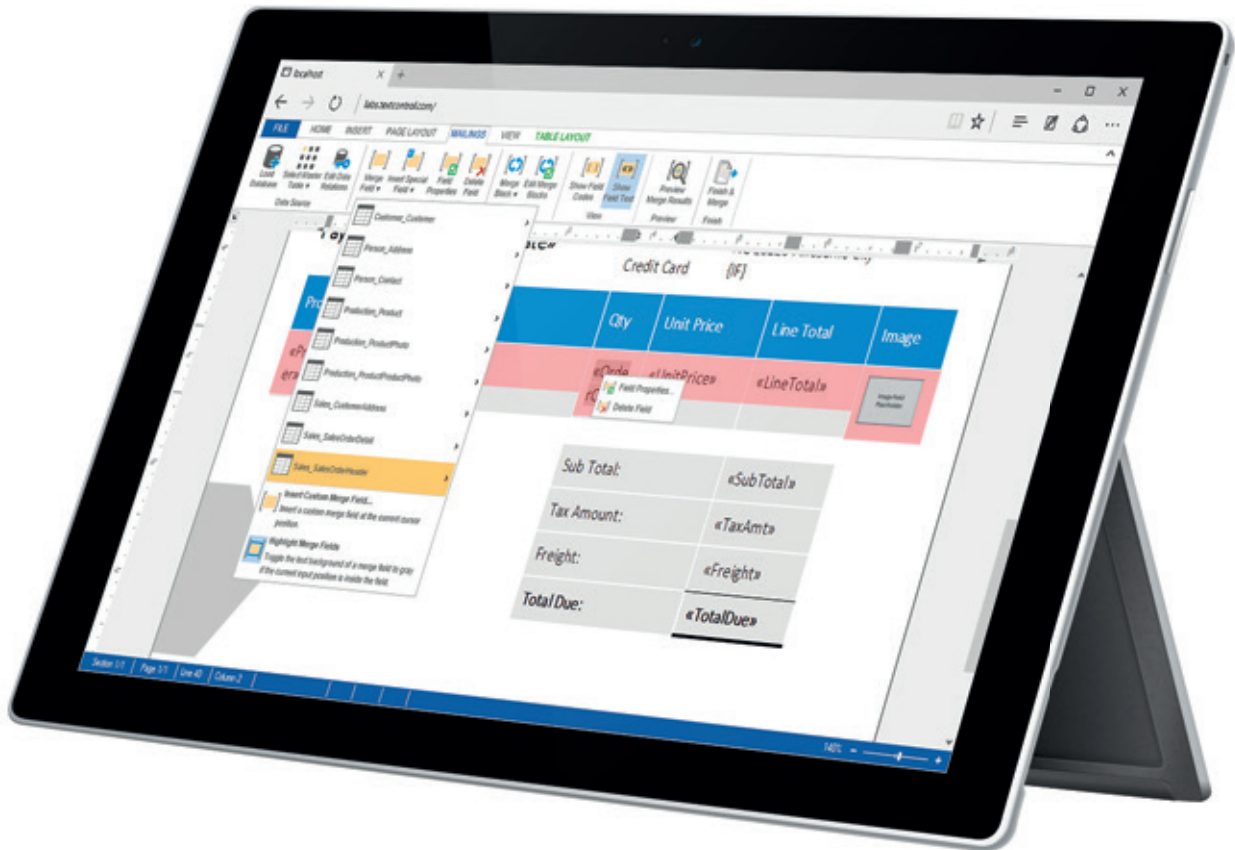
Now with full  
ASP.NET MVC support.



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.





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](http://www.textcontrol.com/html5)

**X13**  
released

Software • Training • Consulting

**TEXT CONTROL**

**General Manager** Jeff Sandquist

**Director** Dan Fernandez

**Editorial Director** Mohammad Al-Sabt [mmeditor@microsoft.com](mailto:mmeditor@microsoft.com)

**Site Manager** Kent Sharkey

**Editorial Director, Enterprise Computing Group** Scott Bekker

**Editor in Chief** Michael Desmond

**Features Editor** Sharon Terdeman

**Features Editor** Ed Zintel

**Group Managing Editor** Wendy Hernandez

**Senior Contributing Editor** Dr. James McCaffrey

**Contributing Editors** Dino Esposito, Frank La Vigne, Julie Lerman, Mark Michaelis, Ted Neward, David S. Platt, Bruno Terkaly

**Vice President, Art and Brand Design** Scott Shultz

**Art Director** Joshua Gould



**President**  
Henry Allain

**Chief Revenue Officer**  
Dan LaBianca

**Chief Marketing Officer**  
Carmel McDonagh

## ART STAFF

**Creative Director** Jeffrey Langkau

**Associate Creative Director** Scott Rovin

**Senior Art Director** Deirdre Hoffman

**Art Director** Michele Singh

**Assistant Art Director** Dragutin Cvijanovic

**Graphic Designer** Erin Horlacher

**Senior Graphic Designer** Alan Tao

**Senior Web Designer** Martin Peace

## PRODUCTION STAFF

**Print Production Coordinator** Lee Alexander

## ADVERTISING AND SALES

**Chief Revenue Officer** Dan LaBianca

**Regional Sales Manager** Christopher Kourtoglou

**Account Executive** Caroline Stover

**Advertising Sales Associate** Tanya Egenolf

## ONLINE/DIGITAL MEDIA

**Vice President, Digital Strategy** Becky Nagel

**Senior Site Producer, News** Kurt Mackie

**Senior Site Producer** Gladys Rama

**Site Producer** Chris Paoli

**Site Producer, News** David Ramel

**Director, Site Administration** Shane Lee

**Site Administrator** Biswarup Bhattacharjee

**Front-End Developer** Anya Smolinski

**Junior Front-End Developer** Casey Rysavy

**Executive Producer, New Media** Michael Domingo

**Office Manager & Site Assoc.** James Bowling

## LEAD SERVICES

**Vice President, Lead Services** Michele Imgrund

**Senior Director, Audience Development & Data Procurement** Annette Levee

**Director, Audience Development & Lead Generation Marketing** Irene Fincher

**Director, Client Services & Webinar Production** Tracy Cook

**Director, Lead Generation Marketing** Eric Yoshizuru

**Director, Custom Assets & Client Services** Mallory Bundy

**Senior Program Manager, Client Services & Webinar Production** Chris Flack

**Project Manager, Lead Generation Marketing** Mahal Ramos

**Coordinator, Lead Generation Marketing** Obum Ukabam

## MARKETING

**Chief Marketing Officer** Carmel McDonagh

**Vice President, Marketing** Emily Jacobs

**Senior Manager, Marketing** Christopher Morales

**Marketing Coordinator** Alicia Chew

**Marketing & Editorial Assistant** Dana Friedman

## ENTERPRISE COMPUTING GROUP EVENTS

**Vice President, Events** Brent Sutton

**Senior Director, Operations** Sara Ross

**Senior Director, Event Marketing** Merikay Marzoni

**Events Sponsorship Sales** Danna Vedder

**Senior Manager, Events** Danielle Potts

**Coordinator, Event Marketing** Michelle Cheng

**Coordinator, Event Marketing** Chantelle Wallace



**Chief Executive Officer**  
Rajeev Kapur

**Chief Operating Officer**  
Henry Allain

**Vice President & Chief Financial Officer**  
Michael Rafter

**Chief Technology Officer**  
Erik A. Lindgren

**Executive Vice President**  
Michael J. Valenti

**Chairman of the Board**  
Jeffrey S. Klein

**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](mailto: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.

Printed in the U.S.A. Reproductions in whole or part prohibited except by written permission. Mail requests to "Permissions Editor," c/o MSDN Magazine, 4 Venture, Suite 150, Irvine, CA 92618.

**LEGAL DISCLAIMER** The information in this magazine has not undergone any formal testing by 1105 Media, Inc. and is distributed without any warranty expressed or implied. Implementation or use of any information contained herein is the reader's sole responsibility. While the information has been reviewed for accuracy, there is no guarantee that the same or similar results may be achieved in all environments. Technical inaccuracies may result from printing errors and/or new developments in the industry.

**CORPORATE ADDRESS** 1105 Media, 9201 Oakdale Ave. Ste 101, Chatsworth, CA 91311 [www.1105media.com](http://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](mailto: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](mailto:1105reprints@parsintl.com). [www.magreprints.com/QuickQuote.asp](http://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: [jl@meritdirect.com](mailto:jl@meritdirect.com); Web: [www.meritdirect.com/1105](http://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](http://Redmondmag.com).

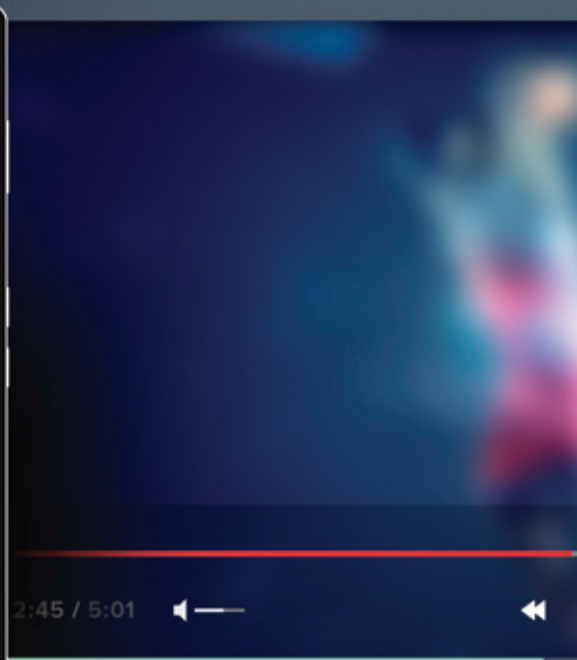
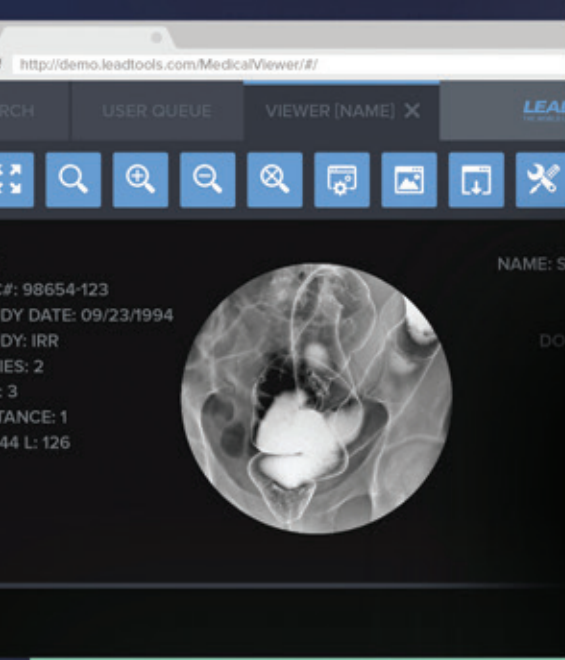
E-mail: To e-mail any member of the staff, please use the following form: [FirstInitial.LastName@1105media.com](mailto:FirstInitial.LastName@1105media.com)  
Irvine Office (weekdays, 9:00 a.m. – 5:00 p.m. PT)  
Telephone 949-265-1520; Fax 949-265-1528  
4 Venture, Suite 150, Irvine, CA 92618

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 contents herein do not necessarily express those of the publisher.



# IMAGING DEVELOPMENT MADE EASY



## Document

*Document Viewer & Converter*

*OCR, MICR, OMR & ICR*

*1D & 2D Barcode*

*Forms Recognition*

*Create, Save, Edit & View PDF*

*Annotations & TWAIN*

## Medical

*DICOM, CCOW & HL7*

*PACS Client & Server Framework*

*DICOMWeb (WADO)*

*Web & Desktop Viewers*

*Image Processing & Annotations*

*Medical 3D (MPR, MIP, VRT)*

## Multimedia

*Play, Capture, Convert & DVR*

*MPEG2-TS & RTSP*

*Media Streaming Server*

*MPEG-4, H.264, H.265 & more*

*Media Foundation & DirectShow*

*Distributed Transcoding*

.NET Windows API Java WinRT Linux iOS OS X Android JavaScript





## Cognitive Bias

In my last column ([msdn.com/magazine/mt703429](http://msdn.com/magazine/mt703429)), I described how efforts to control the coolant loss event at the Three Mile Island (TMI) nuclear plant in 1979 were shaped by received wisdom from another domain—in this case, training that plant operators received in the U.S. Navy's nuclear fleet. Faced with conflicting information from malfunctioning and dysfunctional systems and controls, operators chose to act first on water level readings in the cooling system's pressurizer tank over those in the reactor core itself. The incident resulted in a partial fuel meltdown and the worst nuclear reactor accident in U.S. history.

The lessons of Three Mile Island extend beyond the biases professionals bring with them as they transition among organizations, projects and job roles.

The lessons of TMI extend beyond the biases professionals bring with them as they transition among organizations, projects and job roles. In fact, TMI and nuclear incidents such as the 2011 Fukushima Daichii disaster in Japan reveal an important aspect about human nature in the face of crisis, and present cautionary lessons for software developers who must be responsive to deadlines, budget constraints, code flaws, security threats and a host of other stresses.

Arnie Gundersen is a nuclear industry veteran and chief engineer at Fairewinds Energy Education. During an April 2016 presentation in Japan, he noted that plant operators at TMI and Fukushima

each relied on instruments that falsely indicated “that there was a lot of water in the nuclear reactor, when in fact there was none.”

He went on to say: “Every reading that was true and really bad, they thought of as erroneous. Every reading that was erroneous but really good, they relied upon. That's a trend that I always see in emergency response. Operators want to believe the instruments that lead them to the conclusion they want to get to.”

Normalcy bias explains some of this. Humans are hardwired to underestimate the likelihood and impacts of a disaster and tend to, as Wikipedia notes, “interpret warnings in the most optimistic way possible, seizing on any ambiguities to infer a less serious situation.” This cognitive quirk occurs all over the place—in aircraft cockpits, financial institutions, government bodies and, yes, software development shops. Banks and financial firms, for instance, continued to engage in risky behavior ahead of the global financial collapse of 2008, despite clear indications of the impending downturn. In the minutes and hours before the Deepwater Horizon oil spill in 2010, operators failed to act on abnormal pressure and fluid readings in the well, which portended the calamitous blow out. After the explosion, British Petroleum downplayed the impact, estimating the flow rate of oil into the Gulf of Mexico at just 1,000 to 5,000 barrels per day, while the U.S. government's Flow Rate Technical Group (FRTG) placed that figure at 62,000 barrels.

Ignoring troubling indicators, downplaying damage, and choosing to believe information that supports positive outcomes—these are flawed responses that can make bad situations terrible. But the motivation to engage in them is strong. When I interviewed Gundersen, he drove the point home by citing author Upton Sinclair, who famously wrote: “It is difficult to get a man to understand something, when his salary depends on his not understanding it.”

For developers pressed by unrealistic ship schedules, inadequate budgets and ambitious software requirements, the ability to make clear-eyed judgments spells the difference between making tough decisions today and facing much more difficult ones down the road.

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

© 2016 Microsoft Corporation. All rights reserved.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, you are not permitted to reproduce, store, or introduce into a retrieval system *MSDN Magazine* or any part of *MSDN Magazine*. If you have purchased or have otherwise properly acquired a copy of *MSDN Magazine* in paper format, you are permitted to physically transfer this paper copy in unmodified form. Otherwise, you are not permitted to transmit copies of *MSDN Magazine* (or any part of *MSDN Magazine*) in any form or by any means without the express written permission of Microsoft Corporation.

A listing of Microsoft Corporation trademarks can be found at [microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx](http://microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx). Other trademarks or trade names mentioned herein are the property of their respective owners.

*MSDN Magazine* is published by 1105 Media, Inc. 1105 Media, Inc. is an independent company not affiliated with Microsoft Corporation. Microsoft Corporation is solely responsible for the editorial contents of this magazine. The recommendations and technical guidelines in *MSDN Magazine* are based on specific environments and configurations. These recommendations or guidelines may not apply to dissimilar configurations. Microsoft Corporation does not make any representation or warranty, express or implied, with respect to any code or other information herein and disclaims any liability whatsoever for any use of such code or other information. *MSDN Magazine*, MSDN and Microsoft logos are used by 1105 Media, Inc. under license from owner.





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

NEW  
v5.5

- 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



## Complete Suite of Accurate PDF Components

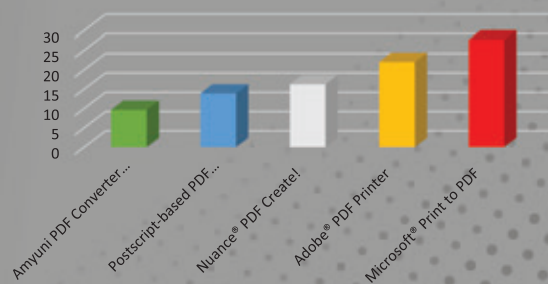
- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
- Includes our Microsoft WHQL certified PDF Converter printer driver
- Export PDF documents into other formats such as Jpeg, PNG, XAML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



## 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

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



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

All trademarks are property of their respective owners. © Amyuni Technologies Inc. All rights reserved.

All development tools available at  
[www.amyuni.com](http://www.amyuni.com)



## Building an Historical CRUD, Part 2

Conceptually speaking, an historical Create, Read, Update, Delete (CRUD) is the classic CRUD extended with an additional parameter: a date. An historical CRUD lets you add, update and delete records from a database and lets you query the state of the database at a given point in time. An historical CRUD gives your applications a built-in infrastructure for business intelligence analysis and advanced reporting features.

In last month's column ([msdn.com/magazine/mt703431](http://msdn.com/magazine/mt703431)), I introduced the theoretical foundation of historical CRUD systems. In this article, I'll give a practical demonstration.

### Presenting the Sample Scenario

For the purposes of this article, I'll consider a simple booking system. For example, take the system a company uses internally to let employees book meeting rooms. In the end, such software is a plain CRUD where a new record is created to reserve a slot. The same record is updated if the meeting is moved to a different time or deleted if the meeting is canceled.

If you code such a booking system as a regular CRUD, you know the latest state of the system, but lose any information about updated

and deleted meetings. Is this really a problem? It depends. It's probably not a problem if you simply look at the affects of meetings on the actual business. However, if you're looking for ways to improve the overall performance of employees, then an historical CRUD that tracks updates and deletions of records might reveal that too many times meetings are moved or canceled and that could be the sign of less-than-optimal internal processes or bad attitude.

**Figure 1** presents a realistic UI for a room-booking system. The underlying database is a SQL Server database with a couple of linked tables: Rooms and Bookings.

The sample application is designed as an ASP.NET MVC application. When the user clicks to place the request, a controller method kicks in and processes the posted information. The following code snippet gives a clear idea of the code handling the request on the server side:

```
[HttpPost]
public ActionResult Add(RoomRequest room)
{
    service.AddBooking(room);
    return RedirectToAction("index", "home");
}
```

The method belongs to a BookingController class and delegates to an injected worker service class the burden of organizing

the actual work. An interesting aspect of the method's implementation is that it redirects to the front page in **Figure 1** after creating the booking. There's no explicit view being created as the result of the add-booking operation. This is a side effect of choosing a Command Query Responsibility Segregation (CQRS) architecture. The add-booking command is posted to the back end; it alters the state of the system and that's it. Had the sample application used AJAX to post, there would've been no need to refresh anything and the command would've been a standalone operation with no visible link to the UI.

The core difference between a classic CRUD and an historical CRUD is that the latter keeps track of all operations that alter the state of the system since the beginning. To plan an historical CRUD, you should think of business operations as commands you give to the system and of a mechanism to track those commands down. Each command alters the state of the system and an historical CRUD

Corner Room		Small	
09:00	Available		
10:00	Available		
11:00	Booked : 53	View	Edit
12:00	Available		
13:00	Available		
14:00	Available		
15:00	Available		
16:00	Available		

**Book now**  

Room  
Corner Room

Time of the day  
10

Length  
1 hour

Name  
Name

Notes  
Notes

Place request

Figure 1 The Front-End UI for a Booking System

# DevExpress Spreadsheet Control

## Excel® Inspired UI Controls for Desktops and the Web

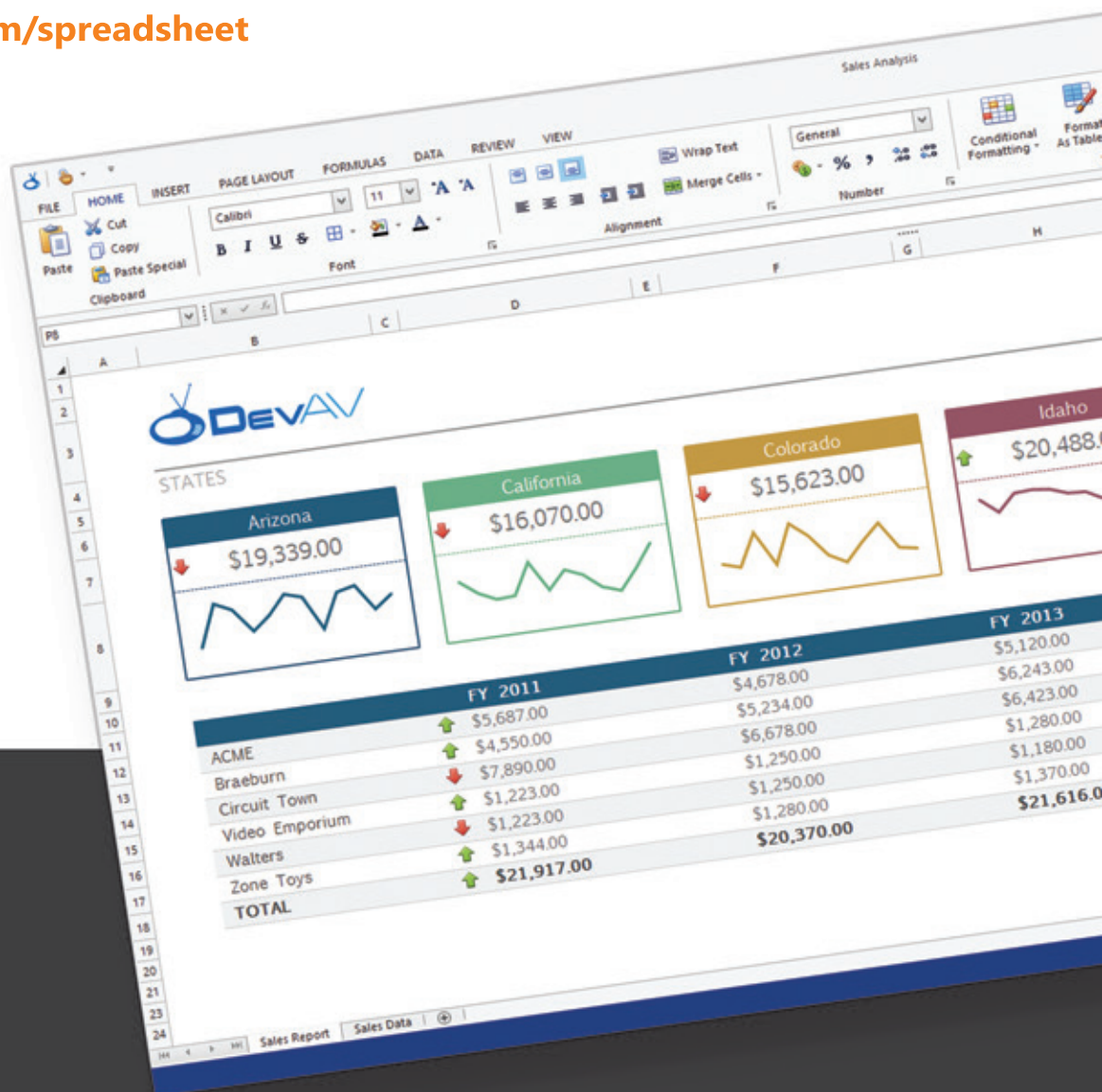
It's no secret...spreadsheets are an important part of business and if you need to incorporate the UX and functionality end-users have come to expect from Microsoft Excel®, DevExpress Spreadsheet is the tool for you. The component library ships with dozens of easy-to-implement features including chart support and fully integrates with other DevExpress components like our Office® inspired Ribbon.

WIN WPF ASP MVC

### Your Next Great Spreadsheet Starts @DevExpress

See how you can harness the power of spreadsheets in your next .NET app.

[devexpress.com/spreadsheet](http://devexpress.com/spreadsheet)





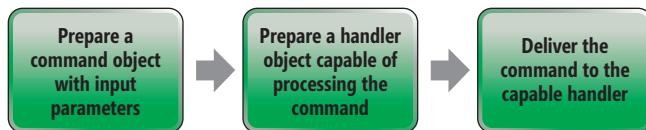


Figure 2 The Chain of Core Steps to Process a Command

keeps track of each state the system reaches. Any reached state is logged as an event. An event is the mere and immutable description of something that has happened. Once you have the list of events, you can create multiple projections of data on top of it, the most popular of which is just the current status of involved business entities.

In an application, events originate directly from the execution of user commands or indirectly from other commands or external input. In this sample scenario, you expect the user to click a button to post a booking request.

## Processing the Command

Here's a possible implementation of the `AddBooking` method of the application's controller:

```

public void AddBooking(RoomRequest request)
{
    var command = new RequestBookingCommand(request);
    var saga = new BookingSaga();
    var response = saga.AddBooking(command);

    // Do something based on the outcome of the command
}
  
```

The `RoomRequest` class is a plain data-transfer object populated by the ASP.NET MVC binding layer out of posted data. The `RequestBookingCommand` class, instead, stores the input parameters required to execute the command. In such a simple scenario, the two classes nearly coincide. How would you process the command? **Figure 2** presents the key three steps of processing a command.

The handler is a component that receives the command and processes it. A handler can be invoked through a direct in-memory call from within the worker service code or you can have a bus in the middle, as shown here:

```

public void AddBooking(RoomRequest request)
{
    var command = new RequestBookingCommand(request);

    // Place the command on the bus for
    // registered components to pick it up
    BookingApplication.Bus.Send(command);
}
  
```

A bus might bring a couple of benefits. One is that you can easily handle scenarios in which multiple handlers might be interested in the same command. Another benefit is that a bus might be configured to be a reliable messaging tool that ensures the delivery of the message over time and overcoming possible connectivity issues. In addition, a bus can just be the component that offers the ability to log the command.

The handler might be a simple one-off component that starts and ends in the same request or it can be a long-running workflow that takes hours or days to complete and may be suspended waiting for human approval at some point. Handlers that are not simple one-off task executors are often called sagas.

In general, you use a bus or a queue if you have specific requirements in terms of scalability and reliability. If you're just looking for

building an historical CRUD in lieu of a classic CRUD, you probably don't need to use a bus. Whether you use a bus or not, at some point the command might reach its one-off or long-running handler. The handler is supposed to carry out whatever tasks are expected. Most tasks consist of core operations on a database.

## Logging the Command

In a classic CRUD, writing information to a database would mean adding a record that lays out the values passed in. In an historical CRUD perspective, though, the newly added record represents the *created* event of a booking. The created event of a booking is an independent and immutable piece of information that includes a unique identifier for the event, a timestamp, a name and a list of arguments specific of the event. The arguments of a created event typically include all the columns you would fill in for a newly added Booking record in a classic Bookings table. The arguments of an *updated* event, instead, are limited to the fields that are actually updated. Subsequently, all updated events might not have the same content. Finally, the arguments of a *deleted* event are limited to the values that uniquely identify the booking.

Any operation of an historical CRUD is made up of two steps:

1. Log the event and its related data.
2. Ensure the current state of the system is immediately and quickly queryable.

In this way, the current state of the system is always available and up-to-date and all the operations that led to it are also available for any further analysis. Note that the "current state of the system" is just the only state you see in a classic CRUD system. To be effective in the context of a simple CRUD system, the step of logging the event and updating the state of the system should take place synchronously and within the same transaction, as shown in **Figure 3**.

As things are, every time you add, edit or delete a booking record you maintain the overall list of bookings up-to-date while knowing the exact sequence of events that led to the current state. **Figure 4** shows the two SQL Server tables involved in the sample scenario and their content after an insert and update.

The Bookings table lists all distinct bookings found in the system and for each returns the current state. The `LoggedEvents` table lists all the events for the various bookings in the order they were recorded. For example, the booking 54 has been created on a given date and modified a few days later. In the example, the `Cargo` column in the picture stores the JSON serialized stream of the command being executed.

Figure 3 Logging an Event and Updating the System

```

using (var tx = new TransactionScope())
{
    // Create the "regular" booking in the Bookings table
    var booking = _bookingRepository.AddBooking(
        command.RoomId, ...);
    if (booking == null)
    {
        tx.Dispose();
        return CommandResponse.Fail;
    }
    // Track that a booking was created
    var eventToLog = command.ToEvent(booking.Id);
    eventRepository.Store(eventToLog);
    tx.Complete();
    return CommandResponse.Ok;
}
  
```



## Using Logged Events in the UI

Let's say that an authorized user wants to see the details of a pending booking. Probably the user will get to the booking from a calendar list or through a time-based query. In both cases, the fundamental facts of the booking—when, how long and who—are already known and the details view might even be of little use. It might really be helpful, though, if you could show the entire history of the booking, as shown in **Figure 5**.

By reading through the logged events, you can build a view model that includes a list of states for the same aggregate entity—booking #54. In the sample application, when the user clicks to view the details of the booking a modal popup appears and some JSON is downloaded in the background. The endpoint that returns the JSON is shown here:

```
public JsonResult BookingHistory(int id)
{
    var history = _service.History(id);
    var dto = history.ToJavaScriptSlotHistory();
    return Json(dto, JsonRequestBehavior.AllowGet);
}
```

The History method on the worker service does most of the work here. The core part of this work is querying all events that related to the specified booking ID:

```
var events = new EventRepository().All(aggregateId);
foreach (var e in events)
{
    var slot = new SlotInfo();
    switch (e.Action)
    {
        :
    }
    history.Changelist.Add(slot);
}
```

As you loop through the logged events, an appropriate object is appended to the data-transfer object to be returned. Some transformation performed in the `ToJavaScriptSlotHistory` makes it quick and easy to display the delta between two consecutive states in the form you see in **Figure 5**.

It's remarkable, though, that while logging events even within a CRUD allows for such nice improvements in the UI, the largest value lies in the fact that you now know everything that ever happened within the system and can process that data to extract any custom projection of data you might need at some point. For example, you might create a statistics of the update and let analysts come to the conclusion that the entire process of requesting meeting rooms doesn't work in the company because people too often book and then update or delete. You can also easily track down what the situation was of the bookings at a specific date by simply querying events logged until then and calculating the subsequent state of things. In a nutshell, an historical CRUD opens up a whole new world of possibilities for applications.

## Wrapping Up

Historical CRUD is simply a smarter way of evolving plain CRUD applications. Yet, this discussion touched on buzzwords and patterns that have a lot more potential, such as CQRS, event sourcing, bus and queues, and message-based business logic. If you found this article helpful, I suggest you go back and read my July 2015 ([msdn.com/magazine/mt238399](http://msdn.com/magazine/mt238399)) and August 2015 ([msdn.com/magazine/mt185569](http://msdn.com/magazine/mt185569)) columns. In light of this example, you might find those articles even more inspiring! ■

Id	RequestId	Name	RoomId	StartingAt	Length	Notes
54	4f85063e-4ce5-450f-920d-57622454ad95	Joe	1	10	1	NULL
..	NULL	NULL	NULL	NULL	NULL	NULL

Id	Action	AggregateId	Carqo	TimeStamp
1062	AddBooking	54	("RoomId":1,"Hour":10,"Length":1,"UserName":"Paul","Notes":"","Name":"AddBooking")	2016-04-01 13:07:03.473
2058	EditBooking	54	("BookingId":54,"Hour":10,"Length":1,"UserName":"Joe","Name":"EditBooking")	2016-04-04 15:49:49.687
•	NULL	NULL	NULL	NULL

Figure 4 Bookings and LoggedEvents Tables Side by Side

**HISTORICAL CRUD**

Corner Room   Small

Time	Status
09:00	Available
10:00	Booked : 54
11:00	Available
12:00	Available
13:00	Available
14:00	Available
15:00	Available
16:00	Available

**Booking 54 :: History**

Action	When	Room Id	Starting at	Length	Name
Created	01 Apr 2016 15:07	1	10	1	Paul
Updated	04 Apr 2016 17:49				Joe

Close

Figure 5 Consuming Logged Events in the UI

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

**THANKS** to the following Microsoft technical expert for reviewing this article: Jon Arne Saeteras

# Use Custom Middleware to Detect and Fix 404s in ASP.NET Core Apps

Steve Smith

If you've ever lost something at a school or an amusement park, you may have had the good fortune of getting it back by checking the location's Lost and Found. In Web applications, users frequently make requests for paths that aren't handled by the server, resulting in 404 Not Found response codes (and occasionally humorous pages explaining the problem to the user). Typically, it's up to the user to find what they're looking for on their own, either through repeated guesses or perhaps using a search engine. However, with a bit of middleware, you can add a "lost and found" to your ASP.NET Core app that will help users find the resources they're looking for.

This article is based on ASP.NET Core 1.0 RC1.  
Some information may change when RC2 becomes available.

## This article discusses:

- Creating a separate middleware class
- Detecting and recording 404 responses
- Displaying Not Found requests
- Fixing 404s
- Configuring the middleware and adding support for storing data

## Technologies discussed:

ASP.NET Core 1.0, Entity Framework Core 1.0

## Code download available at:

[bit.ly/1VUcY0J](http://bit.ly/1VUcY0J)

## What Is Middleware?

The ASP.NET Core documentation defines middleware as “components that are assembled into an application pipeline to handle requests and responses.” At its simplest, middleware is a request delegate, which can be represented as a lambda expression, like this one:

```
app.Run(async context => {  
    await context.Response.WriteAsync("Hello world");  
});
```

If your application consists of just this one bit of middleware, it will return “Hello world” to every request. Because it doesn't refer to the next piece of middleware, this particular example is said to terminate the pipeline—nothing defined after it will be executed. However, just because it's the end of the pipeline doesn't mean you can't “wrap” it in additional middleware. For instance, you could add some middleware that adds a header to the previous response:

```
app.Use(async (context, next) =>  
{  
    context.Response.Headers.Add("Author", "Steve Smith");  
    await next.Invoke();  
});  
app.Run(async context =>  
{  
    await context.Response.WriteAsync("Hello world ");  
});
```

The call to `app.Use` wraps the call to `app.Run`, calling into it using `next.Invoke`. When you write your own middleware, you can choose whether you want it to perform operations before, after, or both before and after the next middleware in the pipeline. You can also short-circuit the pipeline by choosing not to call `next`. I'll show how this can help you create your 404-fixing middleware.

Figure 1 Middleware Class Template

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public Task Invoke(HttpContext httpContext)
    {
        return _next(httpContext);
    }
}

// Extension method used to add the middleware to the HTTP request pipeline.
public static class MyMiddlewareExtensions
{
    public static IApplicationBuilder UseMyMiddleware(this
    IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddleware>();
    }
}
```

If you're using the default MVC Core template, you won't find such low-level delegate-based middleware code in your initial Startup file. It's recommended that you encapsulate middleware in its own classes, and provide extension methods (named `UseMiddlewareName`) that can be called from Startup. The built-in ASP.NET middleware follows this convention, as these calls demonstrate:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
app.UseStaticFiles();
app.UseMvc();
```

The order of your middleware is important. In the preceding code, the call to `UseDeveloperExceptionPage` (which is only configured when the app is running in a development environment) should be wrapped around (thus added before) any other middleware that might produce an error.

## In a Class of Its Own

I don't want to clutter my Startup class with all of the lambda expressions and detailed implementation of my middleware. Just as with the built-in middleware, I want my middleware to be added to the pipeline with one line of code. I also anticipate that my

middleware will need services injected using dependency injection (DI), which is easily achieved once the middleware is refactored into its own class. (See my May article at [msdn.com/magazine/mt703433](http://msdn.com/magazine/mt703433) for more on DI in ASP.NET Core.)

Because I'm using Visual Studio, I can add middleware by using Add New Item and choosing the Middleware Class template. **Figure 1** shows the default content this template produces, including an extension method for adding the middleware to the pipeline via `UseMiddleware`.

Typically, I'll add `async` to the `Invoke` method signature, and then change its body to:

```
await _next(httpContext);
```

This makes the invocation asynchronous.

Once I've created a separate middleware class, I move my delegate logic into the `Invoke` method. Then I replace the call in `Configure` with a call to the `UseMyMiddleware` extension method. Running the app at this point should verify that the middleware still behaves as it did before, and the `Configure` class is much easier to follow when it consists of a series of `UseSomeMiddleware` statements.

## Detecting and Recording 404 Not Found Responses

In an ASP.NET application, if a request is made that doesn't match any handlers, the response will include a `StatusCode` set to 404. I can create a bit of middleware that will check for this response code (after calling `_next`) and take some action to record the details of the request:

```
await _next(httpContext);
if (httpContext.Response.StatusCode == 404)
{
    _requestTracker.Record(httpContext.Request.Path);
}
```

I want to be able to keep track of how many 404s a particular path has had, so I can fix the most common ones and get the most out of my corrective actions. To do that, I create a service called `RequestTracker` that records instances of 404 requests based on their path. `RequestTracker` is passed into my middleware through DI, as shown in **Figure 2**.

It's recommended that you encapsulate middleware in its own classes, and provide extension methods (named `UseMiddlewareName`) that can be called from Startup.

To add `NotFoundMiddleware` to my pipeline, I call the `UseNotFoundMiddleware` extension method. However, because it now depends on a custom service being configured in the services container, I also need to ensure the service is registered. I create an extension method on `IServiceCollection` called `AddNotFoundMiddleware` and call this method in `ConfigureServices` in Startup:

Figure 2 Dependency Injection Passing RequestTracker into Middleware

```
public class NotFoundMiddleware
{
    private readonly RequestDelegate _next;
    private readonly RequestTracker _requestTracker;
    private readonly ILogger _logger;

    public NotFoundMiddleware(RequestDelegate next,
        ILoggerFactory loggerFactory,
        RequestTracker requestTracker)
    {
        _next = next;
        _requestTracker = requestTracker;
        _logger = loggerFactory.CreateLogger<NotFoundMiddleware>();
    }
}
```

```
public static IServiceCollection AddNotFoundMiddleware(
    this IServiceCollection services)
{
    services.AddSingleton<INotFoundRequestRepository,
        InMemoryNotFoundRequestRepository>();
    return services.AddSingleton<RequestTracker>();
}
```

In this case, my `AddNotFoundMiddleware` method ensures an instance of my `RequestTracker` is configured as a `Singleton` in the services container, so it will be available to inject into the `NotFoundMiddleware` when it's created. It also wires up an in-memory implementation of the `INotFoundRequestRepository`, which the `RequestTracker` uses to persist its data.

Because many simultaneous requests might come in for the same missing path, the code in **Figure 3** uses a simple lock to ensure duplicate instances of `NotFoundRequest` aren't added, and the counts are incremented properly.

## Displaying Not Found Requests

Now that I have a way to record 404s, I need a way to view this data. To do this, I'm going to create another small middleware component that will display a page showing all of the recorded `NotFoundRequests`, ordered by how many times they've occurred. This middleware will check to see if the current request matches a particular path, and will ignore (and pass through) any requests that don't match the path. For matching paths, the middleware will return a page with a table containing the `NotFound` requests, ordered by their frequency. From there, the user will be able to assign individual requests a corrected path, which will be used by future requests instead of returning a 404.

**Figure 4** demonstrates how simple it is to have the `NotFoundPageMiddleware` check for a certain path, and to make updates based on querystring values using that same path. For security

Figure 3 RequestTracker

```
public class RequestTracker
{
    private readonly INotFoundRequestRepository _repo;
    private static object _lock = new object();

    public RequestTracker(INotFoundRequestRepository repo)
    {
        _repo = repo;
    }

    public void Record(string path)
    {
        lock(_lock)
        {
            var request = _repo.GetByPath(path);
            if (request != null)
            {
                request.IncrementCount();
            }
            else
            {
                request = new NotFoundRequest(path);
                request.IncrementCount();
                _repo.Add(request);
            }
        }
    }

    public IEnumerable<NotFoundRequest> ListRequests()
    {
        return _repo.List();
    }
    // Other methods
}
```

Figure 4 NotFoundPageMiddleware

```
public async Task Invoke(HttpContext httpContext)
{
    if (!httpContext.Request.Path.StartsWithSegments("/fix404s"))
    {
        await _next(httpContext);
        return;
    }
    if (httpContext.Request.Query.Keys.Contains("path") &&
        httpContext.Request.Query.Keys.Contains("fixedpath"))
    {
        var request = _requestTracker.GetRequest(httpContext.Request.Query["path"]);
        request.SetCorrectedPath(httpContext.Request.Query["fixedpath"]);
        _requestTracker.UpdateRequest(request);
    }
    Render404List(httpContext);
}
```

reasons, access to the `NotFoundPageMiddleware` path should be restricted to admin users.

As written, the middleware is hardcoded to listen to the path `/fix404s`. It's a good idea to make that configurable, so that different apps can specify whatever path they want. The rendered list of requests shows all requests ordered by how many 404s they've recorded, regardless of whether a corrected path has been set up. It wouldn't be difficult to enhance the middleware to offer some filtering. Other interesting features might be recording more detailed information, so you could see which redirects were most popular, or which 404s were most common in the last seven days, but these are left as an exercise for the reader (or the open source community).

**Figure 5** shows an example of what the rendered page looks like.

## Adding Options

I'd like to be able to specify a different path for the Fix 404s page within different apps. The best way to do so is to create an `Options` class and pass it into the middleware using DI. For this middleware, I create a class, `NotFoundMiddlewareOptions`, which includes a property called `Path` with a value that defaults to `/fix404s`. I can pass this into the `NotFoundPageMiddleware` using the `IOptions<T>` interface, and then set a local field to the `Value` property of this type. Then my magic string reference to `/fix404s` can be updated:

```
if (!httpContext.Request.Path.StartsWithSegments(_options.Path))
```

## Fixing 404s

When a request comes in that matches a `NotFoundRequest` that has a `CorrectedUrl`, the `NotFoundMiddleware` should modify the request to use the `CorrectedUrl`. This can be done by just updating the `path` property of the request:

```
string path = httpContext.Request.Path;
string correctedPath = _requestTracker.GetRequest(path)?.CorrectedPath;
if (correctedPath != null)
{
    httpContext.Request.Path = correctedPath; // Rewrite the path
}
await _next(httpContext);
```

With this implementation, any corrected URL will work just as if its request had been made directly to the corrected path. Then, the request pipeline continues, now using the rewritten path. This may or may not be the desired behavior; for one thing, search engine listings can suffer from having duplicate content indexed on multiple URLs. This approach could result in dozens of URLs all mapping to the same

# 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 ...



## 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](http://www.nsoftware.com)



underlying application path. For this reason, it's often preferable to fix 404s using a permanent redirect (status code 301).

If I modify the middleware to send a redirect, I can have the middleware short circuit in that case, because there's no need to run any of the rest of the pipeline if I've decided I'm just going to return a 301:

```
if(correctedPath != null)
{
    httpContext.Response.Redirect(httpContext.Request.PathBase + correctedPath +
        httpContext.Request.QueryString, permanent: true);
    return;
}
await _next(httpContext);
```

Take care not to set corrected paths that result in infinite redirect loops.

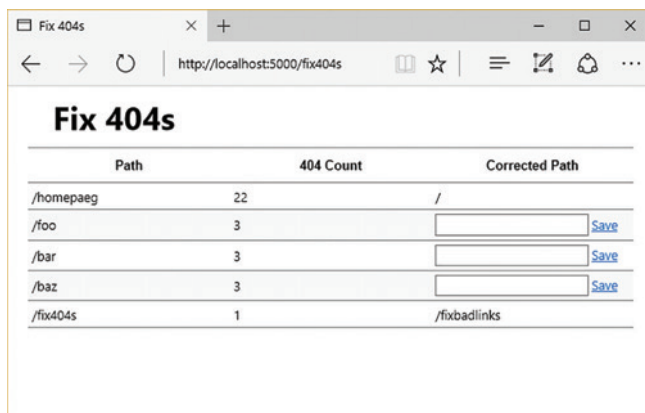
Ideally, the `NotFoundMiddleware` should support both path rewriting and permanent redirects. I can implement this using my `NotFoundMiddlewareOptions`, allowing one or the other to be set for all requests, or I can modify `CorrectedPath` on the `NotFoundRequest` path, so it includes both the path and the mechanism to use. For now I'll just update the options class to support the behavior, and pass `IOptions<NotFoundMiddlewareOptions>` into the `NotFoundMiddleware` just as I'm already doing for the `NotFoundPageMiddleware`. With the options in place, the redirect/rewrite logic becomes:

```
if(correctedPath != null)
{
    if (_options.FixPathBehavior == FixPathBehavior.Redirect)
    {
        httpContext.Response.Redirect(correctedPath, permanent: true);
        return;
    }
    if(_options.FixPathBehavior == FixPathBehavior.Rewrite)
    {
        httpContext.Request.Path = correctedPath; // Rewrite the path
    }
}
```

At this point, the `NotFoundMiddlewareOptions` class has two properties, one of which is an enum:

```
public enum FixPathBehavior
{
    Redirect,
    Rewrite
}

public class NotFoundMiddlewareOptions
{
    public string Path { get; set; } = "/fix404s";
    public FixPathBehavior FixPathBehavior { get; set; } = FixPathBehavior.Redirect;
}
```



Path	404 Count	Corrected Path
/homepaeg	22	/
/foo	3	<input type="text"/> Save
/bar	3	<input type="text"/> Save
/baz	3	<input type="text"/> Save
/fix404s	1	/fixbadlinks

Figure 5 The Fix 404s Page

## Configuring Middleware

Once you have Options set up for your middleware, you pass an instance of these options into your middleware when you configure them in Startup. Alternatively, you can bind the options to your configuration. ASP.NET configuration is very flexible, and can be bound to environment variables, settings files or built up programmatically. Regardless of where the configuration is set, the Options can be bound to the configuration with a single line of code:

```
services.Configure<NotFoundMiddlewareOptions>(
    Configuration.GetSection("NotFoundMiddleware"));
```

With this in place, I can configure my `NotFoundMiddleware` behavior by updating `appsettings.json` (the configuration I'm using in this instance):

```
"NotFoundMiddleware": {
  "FixPathBehavior": "Redirect",
  "Path": "/fix404s"
}
```

Note that converting from string-based JSON values in the settings file to the enum for `FixPathBehavior` is done automatically by the framework.

## Persistence

So far, everything is working great, but unfortunately my list of 404s and their corrected paths are stored in an in-memory collection. This means that every time my application restarts, all of this data is lost. It might be fine for my app to periodically reset its count of 404s, so I can get a sense of which ones are currently the most common, but I certainly don't want to lose the corrected paths I've set.

The first thing I need  
in order to use EF to save and  
retrieve `NotFoundRequests`  
is a `DbContext`.

Fortunately, because I configured the `RequestTracker` to rely on an abstraction for its persistence (`INotFoundRequestRepository`), it's fairly easy to add support for storing the results in a database using Entity Framework Core (EF). What's more, I can make it easy for individual apps to choose whether they want to use EF or the in-memory configuration (great for testing things out) by providing separate helper methods.

The first thing I need in order to use EF to save and retrieve `NotFoundRequests` is a `DbContext`. I don't want to rely on one that the app may have configured, so I create one just for the `NotFoundMiddleware`:

```
public class NotFoundMiddlewareDbContext : DbContext
{
    public DbSet<NotFoundRequest> NotFoundRequests { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        modelBuilder.Entity<NotFoundRequest>().HasKey(r => r.Path);
    }
}
```

Once I have the `DbContext`, I need to implement the repository interface. I create an `EfNotFoundRequestRepository`, which requests

an instance of the `NotFoundMiddlewareDbContext` in its constructor, and assigns it to a private field, `_dbContext`. Implementing the individual methods is straightforward, for example:

```
public IEnumerable<NotFoundRequest> List()
{
    return _dbContext.NotFoundRequests.AsEnumerable();
}

public void Update(NotFoundRequest notFoundRequest)
{
    _dbContext.Entry(notFoundRequest).State = EntityState.Modified;
    _dbContext.SaveChanges();
}
```

At this point, all that remains is to wire up the `DbContext` and EF repository in the app's services container. This is done in a new extension method (and I rename the original extension method to indicate it's for the InMemory version):

```
public static IServiceCollection AddNotFoundMiddlewareEntityFramework(
    this IServiceCollection services, string connectionString)
{
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<NotFoundMiddlewareDbContext>(options =>
            options.UseSqlServer(connectionString));

    services.AddSingleton<INotFoundRequestRepository,
        EfNotFoundRequestRepository>();
    return services.AddSingleton<RequestTracker>();
}
```

I chose to have the connection string passed in, rather than stored in `NotFoundMiddlewareOptions`, because most ASP.NET apps that are using EF will already be providing a connection string to it in the `ConfigureServices` method. If desired, the same variable can be used when calling `services.AddNotFoundMiddlewareEntityFramework(connectionString)`.

The last thing a new app needs to do before it can use the EF version of this middleware is run the migrations to ensure the database table structure is properly configured. I need to specify the middleware's `DbContext` when I do so, because the app (in my case) already has its own `DbContext`. The command, run from the root of the project, is:

```
dotnet ef database update --context NotFoundMiddlewareDbContext
```

If you get an error about a database provider, make sure you're calling `services.AddNotFoundMiddlewareEntityFramework` in `Startup`.

## Next Steps

The sample I've shown here works fine, and includes both an in-memory implementation and one that uses EF to store Not Found request counts and fixed paths in a database. The list of 404s and the ability to add corrected paths should be secured so that only administrators can access it. Finally, the current EF implementation doesn't include any caching logic, resulting in a database query being made with every request to the app. For performance reasons, I would add caching using the `CachedRepository` pattern.

The updated source code for this sample is available at [bit.ly/1VucY0J](http://bit.ly/1VucY0J). ■

**STEVE SMITH** is an independent trainer, mentor and consultant, as well as an ASP.NET MVP. He has contributed dozens of articles to the official ASP.NET Core documentation ([docs.asp.net](http://docs.asp.net)), and helps teams quickly get up to speed with ASP.NET Core. Contact him at [ardalis.com](http://ardalis.com).

THANKS to the following Microsoft technical expert for reviewing this article:  
*Chris Ross*

[msdnmagazine.com](http://msdnmagazine.com)



# 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](http://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](http://dtSearch.com) 1-800-IT-FINDS

# Scale Asynchronous Client-Server Links with Reactive

Peter Vogel

As **asynchronous** processing has become more common in application development, the Microsoft .NET Framework has acquired a wide variety of tools that support specific asynchronous design patterns. Often creating a well-designed asynchronous application comes down to recognizing the design pattern your application is implementing and then picking the right set of .NET components.

In some cases, the match requires integrating several .NET components. Stephen Cleary's article, "Patterns for Asynchronous MVVM Applications: Commands" ([bit.ly/233K0cr](http://bit.ly/233K0cr)), shows how to fully support the Model-View-ViewModel (MVVM) pattern in an asynchronous fashion. In other cases support requires just one component from the .NET Framework. I've discussed implementing the provider/consumers pattern using the BlockingCollection in my VisualStudioMagazine.com Practical .NET columns, "Create Simple, Reliable Asynchronous Apps with BlockingCollection" ([bit.ly/1Tu0pE6](http://bit.ly/1Tu0pE6)), and, "Create Sophisticated Asynchronous Applications with BlockingCollection" ([bit.ly/1SpYyD4](http://bit.ly/1SpYyD4)).

Another example is implementing the observer design pattern to monitor a long-running operation asynchronously. In this scenario,

an asynchronous method that returns a single Task object doesn't work because the client is frequently returning a stream of results. For these scenarios, you can leverage at least two tools from the .NET Framework: the ObservableCollection and Reactive Extensions (Rx). For simple solutions, the ObservableCollection (along with the async and await keywords) is all you need. However, for more "interesting" and, especially, event-driven problems, Rx provides you with better control over the process.

## Defining the Pattern

While the observer pattern is frequently used in UI design patterns—including Model-View-Controller (MVC), Model-View-Presenter (MVP) and MVVM—UIs should be considered as just one scenario from a larger set of scenarios where the observer pattern applies. The definition of the observer pattern (quoting from Wikipedia) is: "An object, called the subject, [that] maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods."

Really, the observer pattern is about getting results from long-running processes to the client *as soon as those results are available*. Without some version of the observer pattern, clients must wait until the last result is available and then have all the results sent to them in a single lump. In an increasingly asynchronous world, you want the observers to process results in parallel with the client as the results become available. To emphasize that you're talking about more than UIs when leveraging the observer pattern, I'll use "client" and "server" instead of "observer" and "subject," in the rest of this article.

## Problems and Opportunities

There are at least three issues and two opportunities with the observer pattern. The first issue is the lapsed-listener problem: Many implementations of the observer pattern require the server to hold a reference to all of its clients. As a result, clients can be

### This article discusses:

- Getting results from a long-running process as soon as those results are available using the observer pattern
- ObservableCollection provides an easy way to create applications that monitor long-running processes
- Upgrading to Reactive Extensions allows you to asynchronously accept output from event-driven processes

### Technologies discussed:

Rx (Reactive Extensions), ObservableCollection, Asynchronous Programming

### Code download available at:

[msdn.com/magazine/0616magcode](http://msdn.com/magazine/0616magcode)



held in memory by the server until the server exits. This obviously isn't an optimal solution for a long-running process in a dynamic system where clients connect and disconnect frequently.

The lapsed-listener problem, however, is just a symptom of the second, larger problem: Many implementations of the observer pattern require the server and the client to be tightly coupled, requiring both the server and the client to be present at all times. At the very least, the client should be able to determine if the server is present and choose not to attach; in addition, the server should be able to function even if there are no clients accepting results.

The third issue is performance-related: How long will it take for the server to notify all clients? Performance in the observer pattern is directly affected by the number of clients to be notified. Therefore, there is an opportunity to improve performance in the observer pattern by letting the client preemptively filter the results that come back from the server. This also addresses the scenarios where the server produces more results (or a wider variety of results) than the client is interested in: The client can indicate that it's only to be notified in specific cases. The second performance opportunity exists around recognizing when the server has no results or has finished producing results. Clients can skip acquiring resources required to process server events until the client is guaranteed there is something to process and clients can release those resources as soon as they know they've processed the last result.

## From Observer to Publish/Subscribe

Factoring in these considerations leads from simple implementations of the observer pattern to the related publish/subscribe model. Publish/subscribe implements the observer pattern in a loosely coupled way that lets servers and clients execute even if the other is currently unavailable. Publish/subscribe also typically implements client-side filtering by letting the client subscribe either to specific topics/channels ("Notify me about purchase orders") or to attributes associated with different kinds of content ("Notify me about any urgent requests").

One issue remains, however. All implementations of the observer pattern tend to tightly couple clients and servers to a specific message format. Changing the format of a message in most publish/subscribe implementations can be difficult because all of the clients must be updated to use the new format.

In many ways, this is similar to the description of a server-side cursor in a database. To minimize transmission costs, the database server doesn't return results as each row is retrieved. However, for large row sets, the database also doesn't return all the rows in a single batch at the end. Instead, the database server typically returns subsets from a cursor held on the server often as those subsets become available. With a database, the client and the server don't have to be simultaneously present: The database server can run when there are no clients present; a client can check to see if the server is accessible and, if not, decide what (if anything) else it can do. The filtering process (SQL) is also very flexible. However, if the database engine changes the format it uses to return rows, then all clients must, at the very least, be recompiled.

## Processing a Cache of Objects

As my case study for looking at a simple observer pattern implementation, I'm using as my server a class that searches an in-memory

cache of invoices. That server could, at the end of its processing, return a collection of all of the invoices. However, I'd prefer to have the client process the invoices individually and in parallel to the server's search process. That means I prefer a version of the process, which returns each invoice as it's found and lets the client process each invoice in parallel with the search for the next invoice.

A simple implementation of the server might look like this:

```
private List<Invoice> foundInvoices = new List<Invoice>();
public List<Invoice> FindInvoices(decimal Amount)
{
    foundInvoices.Clear();
    Invoice inv;
    // ...search logic to add invoices to the collection
    foundInvoices.Add(inv);
    // ...repeat until all invoices found
    return foundInvoices;
}
```

More sophisticated solutions might use yield return to return each invoice as it's found rather than assembling the list. Regardless, a client that calls the FindInvoices method will want to perform some critical activities before and after processing. For example, once the first item is found, the client might want to enable a MatchingInvoices list to hold the invoices at the client or acquire/initialize any resources required to process an invoice. As additional invoices are added, the client would need to process each invoice and, when the server signals that the final invoice is retrieved, release any resources that are no longer required because there are "no more" invoices to process.

During a database retrieval, for example, a read will block until the first row is returned. Once the first row is returned, the client initializes whatever resources are needed to process a row. The read also returns false when the final row is retrieved, letting the client release those resources because there are no more rows to process.

## Creating Simple Solutions with ObservableCollection

The most obvious choice for implementing the observer pattern in the .NET Framework is the ObservableCollection. The ObservableCollection will notify the client (through an event) whenever it's changed.

Rewriting my sample server to use the ObservableCollection class requires only two changes. First, the collection holding the results needs to be defined as an ObservableCollection and made public. Second, it's no longer necessary for the method to return a result: The server only needs to add invoices to the collection.

The new implementation of the server might look like this:

```
public List<Invoice> FindInvoices(decimal Amount)
{
    public ObservableCollection<Invoice> foundInvoices =
        new ObservableCollection<Invoice>();

    public void FindInvoices(decimal Amount)
    {
        foundInvoices.Clear();
        Invoice inv;
        // ...search logic to set inv
        foundInvoices.Add(inv);
        // ...repeat until all invoices are added to the collection
    }
}
```

A client that uses this version of the server only needs to wire up an event handler to the CollectionChanged event of the Invoice-Management's foundInvoices collection. In the following code I've had the class implement the IDisposable interface to support disconnecting from the event:

```

public class SearchInvoices: IDisposable
{
    InvoiceManagement invMgmt = new InvoiceManagement();

    public void SearchInvoices()
    {
        invMgmt.foundInvoices.CollectionChanged += InvoicesFound;
    }
    public void Dispose()
    {
        invMgmt.foundInvoices.CollectionChanged -= InvoicesFound;
    }
}

```

In the client, the `CollectionChanged` event is passed a `NotifyCollectionChangedEventArgs` object as its second parameter. That object's `Action` property specifies both what change was performed on the collection (the actions are: the collection was cleared, new items were added to the collection, existing items were moved/replaced/removed) and information about the changed items (a collection of any added items, a collection of items present in the collection prior to the new items being added, the position of the item that was moved/removed/replaced).

Simple code in the client that would asynchronously process each invoice as it's added to the collection in the server would look like the code in **Figure 1**.

While simple, this code might be inadequate for your needs, especially if you're handling a long-running process or working in a dynamic environment. From an asynchronous design point of view, for example, the code could capture the `Task` object returned by the `HandleInvoiceAsync` so that the client could manage the asynchronous tasks. You'll also want to make sure that the `CollectionChanged` event is raised on the UI thread even if `FindInvoices` runs on a background thread.

Because of where the `Clear` method is called in the server class (just before searching for the first `Invoice`) the `Action` property's `Reset` value can be used as a signal that the first item is about to be retrieved. However, of course, no invoices may be found in the search, so using the `Reset` `Action` might result in the client allocating resources that are never actually used. To actually handle "first item" processing, you'd need to add a flag to the `Add` `Action` processing to execute only when the first item was found.

In addition, the server has a limited number of options for indicating that the last `Invoice` is found so that the client can stop waiting for "the next one." The server could, presumably, clear the collection after finding the last item, but that would just force more complex processing into the `Reset` `Action` processing (have I been processing `Invoices`? If yes, then I've processed the last `Invoice`; if no, then I'm about to process the first `Invoice`).

While, for simple problems, `ObservableCollection` will be fine, any reasonably sophisticated implementation based on `ObservableCollection` (and any application that values efficiency) is going to require some complicated code, especially in the client.

## The Rx Solutions

If you want asynchronous processing then Rx (available through NuGet) can provide a better solution for implementing the

observer pattern by borrowing from the publish/subscribe model. This solution also provides a LINQ-based filtering model, better signaling for first/last item conditions and better error handling.

Rx can also handle more interesting observer implementations than are possible with an `ObservableCollection`. In my case study, after returning the initial list of invoices, my server might continue to check for new invoices that are added to the cache after the original search completes (and that match the search criteria, of course). When an invoice meeting the criteria does appear, the client will want to be notified about the event so that the new invoice can be added to the list. Rx supports these kinds of event-based extensions to the observer pattern better than `ObservableCollection`.

There are two key interfaces in Rx for supporting the observer pattern. The first is `IObservable<T>`, implemented by the server and specifying a single method: `Subscribe`. The server implementing the `Subscribe` method will be passed a reference to an object from a client. To handle the lapsed listener problem, the `Subscribe` method returns a reference to the client for an object that implements the `IDisposable` interface. The client can use that object to disconnect from the server. When the client does disconnect, the server is expected to remove the client from any of its internal lists.

The second is the `IObserver<T>` interface, which must be implemented by the client. That interface requires the client to implement and expose three methods to the server: `OnNext`, `OnCompleted` and `OnError`. The critical method here is `OnNext`, which is used by the server to pass a message to the client (in my case study that message would be new `Invoice` objects that will be returned as each one appears). The server can use the client's `OnCompleted` method to signal that there's no more data. The third method, `OnError`, provides a way for the server to signal to the client that an exception has occurred.

You're welcome to implement the `IObserver` interface yourself, of course (it's part of the .NET Framework). Along with the `ObservableCollection`, that may be all you need if you're creating a synchronous solution (I've written a column about that, too, "Writing Cleaner Code with Reactive Extensions" [bit.ly/10nfQtm]).

However, the Rx includes several packages that provide asynchronous implementations of these interfaces, including implementations for JavaScript and RESTful services. The Rx `Subject` class provides an

implementation of `IObservable` that simplifies implementing an asynchronous publish/subscribe version of the observer pattern.

## Creating an Asynchronous Solution

Creating a server to work with a `Subject` object requires very few changes to the original synchronous server-side code. I replace the old `ObservableCollection` with a `Subject` object that will pass each `Invoice` as it appears to any listening clients. I declare the `Subject` object as public so that clients can access it:

```

public class InvoiceManagement
{
    public IObservable<Invoice> foundInvoice =
        new Subject<Invoice>();
}

```

**Figure 1 Asynchronously Processing Invoices Using ObservableCollection**

```

private async void InvoicesFound(object sender,
    NotifyCollectionChangedEventArgs e)
{
    switch (e.Action)
    {
        case NotifyCollectionChangedAction.Reset:
        {
            // ...initial item processing
            return;
        }
        case NotifyCollectionChangedAction.Add:
        {
            foreach (Invoice inv in e.NewItems)
            {
                await HandleInvoiceAsync(inv);
            }
            return;
        }
    }
}

```

In the body of the method, instead of adding an invoice to a collection, I use the Subject's `OnNext` method to pass each invoice to the client as it's found:

```
public void FindInvoices(decimal Amount)
{
    inv = GetInvoicesForAmount(Amount) // Poll for invoices
    foundInvoice.OnNext(inv);
    // ...repeat...
}
```

In my client, I first declare an instance of the server class. Then, in a method marked as `async`, I call the Subject's `Subscribe` method to indicate that I want to start retrieving messages:

```
public class InvoiceManagementTests
{
    InvoiceManagement invMgmt = new InvoiceManagement();

    public async void ProcessInvoices()
    {
        invMgmt.foundInvoice.Subscribe<Invoice>();
    }
}
```

To filter the results to just the invoices I want, I can apply a LINQ statement to the Subject object. This example filters the invoices to the ones that are back ordered (to use Rx LINQ extensions you'll need to add a `using` statement for the `System.Reactive.Linq` namespace):

```
invMgmt.foundInvoice.Where(i => i.BackOrder == "BackOrder").Subscribe();
```

Once I've started listening to the subject, I can specify what processing I want to do when I receive an invoice. I can, for example, use `FirstAsync` to process just the first invoice returned by the service. In this example, I use the `await` statement with the call to `FirstAsync` so that I can return control to the main body of my application while processing the invoice. This code waits to retrieve that first invoice, then moves on to whatever code I use to initialize the invoice processing process and, finally, processes the invoice:

```
Invoice inv;
inv = await invMgmt.foundInvoice.FirstAsync();
// ...setup code invoices...
HandleInvoiceAsync(inv);
```

One caveat: `FirstAsync` will block if the server hasn't yet produced any results. If you want to avoid blocking, you can use `FirstOrDefaultAsync`, which will return null if the server hasn't produced any results. If there are no results, the client can decide what, if anything, to do.

The more typical case is that the client wants to process all the invoices returned (after filtering) and to do so asynchronously. In that case, rather than use a combination of `Subscribe` and `OnNext`, you can just use the `ForEachAsync` method. You can pass a method

or a lambda expression that processes incoming results. If you pass a method (which can't be asynchronous), as I do here, that method will be passed the invoice that triggered `ForEachAsync`:

```
invMgmt.foundInvoice.ForEachAsync(HandleInvoice);
```

The `ForEachAsync` method can also be passed a cancellation token to let the client signal that it's disconnecting. A good practice would be to pass the token when calling any of Rx \*Async methods to support letting the client terminate processing without having to wait for all objects to be processed.

The `ForEachAsync` won't process any result already processed by a `First` (or `FirstOrDefaultAsync`) method so you can use `FirstOrDefaultAsync` with `ForEachAsync` to check to see if the server has anything to process before processing subsequent objects. However, the Subject's `IsEmpty` method will perform the same check more simply. If the client has to allocate any resources required for processing results, `IsEmpty` allows the client to check to see if there's anything to do before allocating those resources (an alternative would be to allocate those resources on the first item processed in the loop). Using `IsEmpty` with a client that checks to see if there are any results before allocating resources (and starting processing) while also supporting cancellation would give code that looks something like **Figure 2**.

## Wrapping Up

If all you need is a simple implementation of the observer pattern, then `ObservableCollection` might do all you need to process a stream of results. For better control and for an event-based application, the Subject class and the extensions that come with Rx will let your application work in an asynchronous mode by supporting a powerful implementation of the publish/subscribe model (and I haven't looked at the rich library of operators that come with Rx). If you're working with Rx, it's worthwhile to download the Rx Design Guide ([bit.ly/1VOPxGS](http://bit.ly/1VOPxGS)), which discusses the best practices in consuming and producing observable streams.

Rx also provides some support for converting the message type passed between the client and the server by using the `ISubject<TSource, TResult>` interface. The `ISubject<TSource, TResult>` interface specifies two datatypes: an "in" datatype and an "out" datatype. Within the Subject class that implements this interface you can perform any operations necessary to convert the result returned from the server (the "in" datatype) into the result required by the client (the "out" datatype). Furthermore, the `in` parameter is covariant (it will accept the specified datatype or anything the datatype inherits from) and the `out` parameter is contravariant (it will accept the specified datatype or anything that derives from it), giving you additional flexibility.

We live in an increasingly asynchronous world and, in that world, the observer pattern is going to become more important—it's a useful tool for any interface between processes where the server process returns more than a single result. Fortunately, you have several options for implementing the observer pattern in the .NET Framework, including the `ObservableCollection` and Rx. ■

---

**PETER VOGEL** is a systems architect and principal in PH&V Information Services. PH&V provides full-stack consulting from UX design through object modeling and database design.

---

**THANKS** to the following Microsoft technical experts for reviewing this article: Stephen Cleary, James McCaffrey and Dave Sexton

**Figure 2 Code to Support Cancellation and Defer Processing Until Results are Ready**

```
CancellationTokenSource cancelSource = new CancellationTokenSource();

CancellationToken cancel;
cancel = cancelSource.Token;
if (!await invMgmt.foundInvoice.IsEmpty())
{
    // ...setup code for processing invoices...
    try
    {
        invMgmt.foundInvoice.ForEachAsync(HandleInvoice, cancel);
    }
    catch (Exception ex)
    {
        if (ex.GetType() != typeof(CancellationToken))
        {
            // ...report message
        }
    }
    // ...clean up code when all invoices are processed or client disconnects
}
```





# BEST FILE APIs

Open Create Convert Print Save

files from your *applications!*



XLS

DOC



PDF

Contact Us:

US: +1 888 277 6734

EU: +44 141 416 1112

AU: +61 2 8003 5926

[sales@aspose.com](mailto:sales@aspose.com)

SCAN FOR  
20% SAVINGS!



# BUSINESS FILE FORMATS



## ASPOSE.Cells

XLS, CSV, PDF, SVG, HTML, PNG  
BMP, XPS, JPG, SpreadsheetML...

## ASPOSE.Words

DOC, RTF, PDF, HTML, PNG  
ePUB, XML, XPS, JPG...

## ASPOSE.Pdf

PDF, XML, XSL-FO, HTML, BMP  
JPG, PNG, ePUB...

## ASPOSE.Slides

PPT, POT, POTX, XPS, HTML  
PNG, PDF...

## ASPOSE.BarCode

JPG, PNG, BMP, GIF, TIF, WMF  
ICON...

## ASPOSE.Tasks

XML, MPP, SVG, PDF, TIFF  
PNG...

## ASPOSE.Email

MSG, EML, PST, EMLX  
OST, OFT...

## ASPOSE.Imaging

PDF, BMP, JPG, GIF, TIFF  
PNG...

+ MANY MORE!

Get your FREE evaluation copy at [www.aspose.com](http://www.aspose.com)

.NET

Java

Cloud

# Language-Agnostic Code Generation with Roslyn

Alessandro Del Sole

The Roslyn code base provides powerful APIs you can leverage to perform rich code analysis over your source code. For instance, analyzers and code refactorings can walk through a piece of source code and replace one or more syntax nodes with new code you generate with the Roslyn APIs. A common way to perform code generation is via the `SyntaxFactory` class, which exposes factory methods to generate syntax nodes in a way that compilers can understand. The `SyntaxFactory` class is certainly very powerful because it allows generating any possible syntax element, but there are two different `SyntaxFactory` implementations: `Microsoft.CodeAnalysis.CSharp.SyntaxFactory` and `Microsoft.CodeAnalysis.VisualBasic.SyntaxFactory`. This has an important implication if you want to write an analyzer with a code fix that targets

both C# and Visual Basic—you have to write two different analyzers, one for C# and one for Visual Basic, using the two implementations of `SyntaxFactory`, each with a different approach due to the different way those languages handle some constructs. This likely means wasting time writing the analyzer twice, and maintaining them becomes more difficult. Fortunately, the Roslyn APIs also provide the `Microsoft.CodeAnalysis.Editor.SyntaxGenerator`, which allows for language-agnostic code generation. In other words, with `SyntaxGenerator` you can write your code-generation logic once and target both C# and Visual Basic. In this article I'll show you how to perform language-agnostic code generation with `SyntaxGenerator`, and I'll give you some hints about the Roslyn Workspaces APIs.

## This article discusses:

- Using Roslyn to generate syntax that targets both C# and Visual Basic
- The `SyntaxGenerator` class and its members
- Generating syntax nodes and compilation units
- The Workspaces APIs

## Technologies discussed:

Roslyn, C#, Visual Basic

## Code download available at:

[msdn.com/magazine/0616magcode](https://msdn.com/magazine/0616magcode)

## Starting with Code

Let's start with some source code that will be generated using `SyntaxGenerator`. Consider the simple `Person` class that implements the `ICloneable` interface in C# (**Figure 1**) and Visual Basic (**Figure 2**).

You'd probably argue that declaring auto-implemented properties would have the same effect and would keep code much cleaner in this particular case, but later you'll see why I'm using the expanded form.

This implementation of the `Person` class is very simple, but it contains a good number of syntax elements, making it helpful for understanding how to perform code generation with `SyntaxGenerator`. Let's generate this class with Roslyn.

Figure 1 A Simple Person Class in C#

```
public abstract class Person : ICloneable
{
    // Not using auto-props is intentional for demo purposes

    private string _lastName;
    public string LastName
    {
        get
        {
            return _lastName;
        }
        set
        {
            _lastName = value;
        }
    }

    private string _firstName;
    public string FirstName
    {
        get
        {
            return _firstName;
        }
        set
        {
            _firstName = value;
        }
    }

    public Person(string LastName, string FirstName)
    {
        _lastName = LastName;
        _firstName = FirstName;
    }

    public virtual object Clone()
    {
        return MemberwiseClone();
    }
}
```

Figure 2 A Simple Person Class in Visual Basic

```
Public MustInherit Class Person
Implements ICloneable

'Not using auto-props is intentional for demo purposes
Private _lastName As String
Private _firstName As String

Public Property LastName As String
Get
    Return _lastName
End Get
Set(value As String)
    _lastName = value
End Set
End Property

Public Property FirstName As String
Get
    Return _firstName
End Get
Set(value As String)
    _firstName = value
End Set
End Property

Public Sub New(LastName As String, FirstName As String)
    _lastName = LastName
    _firstName = FirstName
End Sub

Public Overridable Function Clone() As Object Implements ICloneable.Clone
    Return MemberwiseClone()
End Function
End Class
```

## Creating a Code Analysis Tool

The first thing to do is create a new project in Visual Studio 2015 with references to the Roslyn libraries. Because of the general purpose of this article, instead of creating an analyzer or refactoring, I'll choose another project template available in the .NET Compiler Platform SDK, the Stand-Alone Code Analysis Tool, available in the Extensibility node of the New Project dialog (see **Figure 3**).

This project template actually generates a console application and automatically adds the proper NuGet packages for the Roslyn APIs, targeting the language of your choice. Because the idea is to target both C# and Visual Basic, the first thing to do is add the NuGet packages for the second language. For instance, if you initially created a C# project, you'll need to download and install the following Visual Basic libraries from NuGet:

- Microsoft.CodeAnalysis.VisualBasic.dll
- Microsoft.CodeAnalysis.VisualBasic.Workspaces.dll
- Microsoft.CodeAnalysis.VisualBasic.Workspaces.Common.dll

You can just install the latter from NuGet, and this will automatically resolve dependencies for the other required libraries. Resolving dependencies is important anytime you plan to use the `SyntaxGenerator` class, no matter what project template you're using. Forgetting to do this will result in exceptions at run time.

Resolving dependencies is  
important anytime you want to  
use the `SyntaxGenerator` class,  
no matter what project template  
you're using.

## Meet `SyntaxGenerator` and the Workspaces APIs

The `SyntaxGenerator` class exposes a static method called `GetGenerator`, which returns an instance of `SyntaxGenerator`. You use the returned instance to perform code generation. `GetGenerator` has the following three overloads:

```
public static SyntaxGenerator GetGenerator(Document document)
public static SyntaxGenerator GetGenerator(Project project)
public static SyntaxGenerator GetGenerator(Workspace workspace, string language)
```

The first two overloads work against a `Document` and a `Project`, respectively. The `Document` class represents a code file in a project, while the `Project` class represents a Visual Studio project as a whole. These overloads automatically detect the language (C# or Visual Basic) the `Document` or `Project` target. `Document`, `Project`, and `Solution` (an additional class that represents a Visual Studio .sln solution) are part of a `Workspace`, which provides a managed way to interact with everything that makes up an MSBuild solution with projects, code files, metadata and objects. The `Workspaces` APIs expose several classes you can use to manage workspaces, such as the `MSBuildWorkspace` class, which allows working against an .sln solution, or the `AdhocWorkspace` class, which is instead very useful when you're not working against an existing MSBuild solution but want an in-memory workspace that represents one.



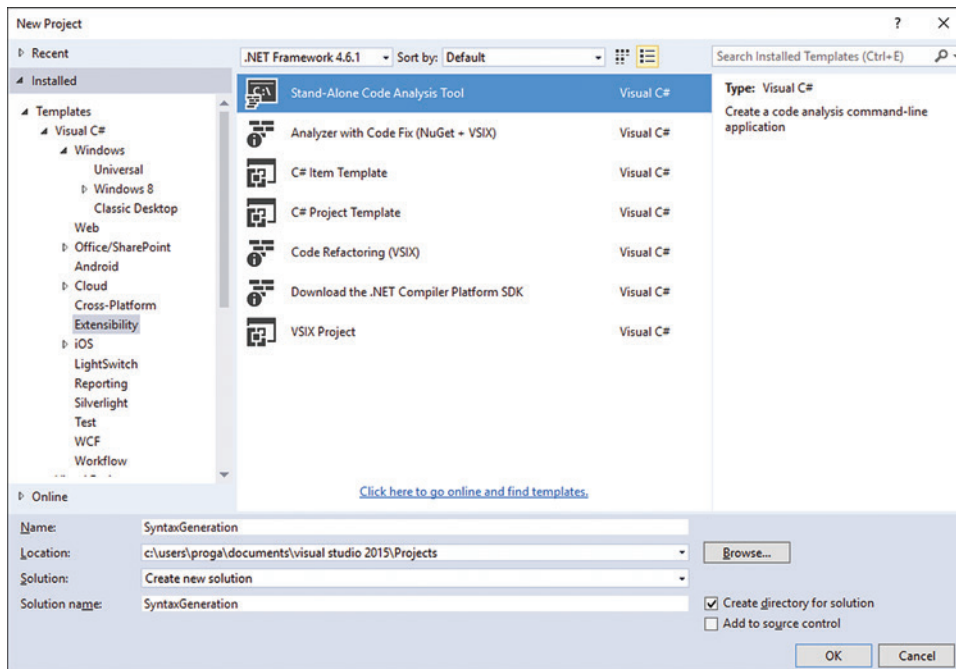


Figure 3 The Stand-Alone Code Analysis Tool Project Template

In the case of analyzers and code refactorings, you already have an MSBuild workspace that allows you to work against code files using instances of the Document, Project and Solution classes. In the current sample project, there's no workspace, so let's create one using the third overload of `SyntaxGenerator`. To get a new empty workspace, you can use the `AdhocWorkspace` class:

```
// Get a workspace
var workspace = new AdhocWorkspace();
```

Now you can get an instance of `SyntaxGenerator`, passing the workspace instance and the desired language as arguments:

```
// Get the SyntaxGenerator for the specified language
var generator = SyntaxGenerator.GetGenerator(workspace, LanguageNames.CSharp);
```

The `SyntaxGenerator` class exposes instance factory methods that generate proper syntax nodes in a way that's compliant with the grammar and semantics of both C# and Visual Basic.

The language name can be `CSharp` or `VisualBasic`, both constants from the `LanguageNames` class. Let's start with C#; later you'll see how to change the language name to `VisualBasic`. You have all the tools you need now and are ready to generate syntax nodes.

## Generating Syntax Nodes

The `SyntaxGenerator` class exposes instance factory methods that generate proper syntax nodes in a way that's compliant with the grammar and semantics of both C# and Visual Basic. For example, methods with names ending with the `Expression` suffix generate expressions; methods with names ending with the `Statement` suffix generate statements; methods with names ending with the `Declaration` suffix generate declarations. For each category, there are specialized methods that generate specific syntax nodes. For instance, `MethodDeclaration` generates a method block, `PropertyDeclaration` generates a property, `FieldDeclaration` generates a field and so on (and, as usual, `IntelliSense` is your best friend). The peculiarity

of these methods is that each returns `SyntaxNode`, instead of a specialized type that derives from `SyntaxNode`, as happens with the `SyntaxFactory` class. This provides great flexibility, especially when generating complex nodes.

Based on the sample `Person` class, the first thing to generate is a `using/Imports` directive for the `System` namespace, which exposes the `ICloneable` interface. This can be accomplished with the `NamespaceImportDeclaration` method as follows:

```
// Create using/Imports directives
var usingDirectives = generator.NamespaceImportDeclaration("System");
```

This method takes a string argument that represents the namespace you want to import. Let's go ahead and declare two fields, which is accomplished via the `FieldDeclaration` method:

```
// Generate two private fields
var lastNameField = generator.FieldDeclaration("_lastName",
    generator.TypeExpression(SpecialType.System_String),
    Accessibility.Private);
var firstNameField = generator.FieldDeclaration("_firstName",
    generator.TypeExpression(SpecialType.System_String),
    Accessibility.Private);
```

Figure 4 Generating Two Properties via the `PropertyDeclaration` Method

```
// Generate two properties with explicit get/set
var lastNameProperty = generator.PropertyDeclaration("LastName",
    generator.TypeExpression(SpecialType.System_String), Accessibility.Public,
    getAccessorStatements: new SyntaxNode[]
    { generator.ReturnStatement(generator.IdentifierName("_lastName")) },
    setAccessorStatements: new SyntaxNode[]
    { generator.AssignmentStatement(generator.IdentifierName("_lastName"),
        generator.IdentifierName("value")) });

var firstNameProperty = generator.PropertyDeclaration("FirstName",
    generator.TypeExpression(SpecialType.System_String),
    Accessibility.Public,
    getAccessorStatements: new SyntaxNode[]
    { generator.ReturnStatement(generator.IdentifierName("_firstName")) },
    setAccessorStatements: new SyntaxNode[]
    { generator.AssignmentStatement(generator.IdentifierName("_firstName"),
        generator.IdentifierName("value")) });
```



# 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](http://www.DynamicPDF.com)

**TRY OUR PDF SOLUTIONS FREE TODAY!**

[www.DynamicPDF.com/eval](http://www.DynamicPDF.com/eval) or call 800.631.5006 | +1 410.772.8620

**ceTe software**



FieldDeclaration takes the field name, the field type, and the accessibility level as arguments. To supply the proper type, you invoke the TypeExpression method, which takes a value from the SpecialType enumeration, in this case System\_String (don't forget to use IntelliSense to discover other values). The accessibility level is set with a value from the Accessibility enumeration. When invoking methods from SyntaxGenerator, it's very common to nest invocations to other methods from the same class, as in the case of TypeExpression. The next step is generating two properties, which is accomplished by invoking the PropertyDeclaration method, shown in **Figure 4**.

Assignments are represented by the AssignmentStatement method, which takes two arguments, the left and right sides of the assignment.

As you can see, generating a syntax node for a property is more complex. Here you still pass a string with the property name, then a TypeExpression for the property type, then the accessibility level. With a property you also typically need to provide the Get and Set accessors, especially for those situations in which you need to execute code other than for setting or returning the property value (such as raising the OnPropertyChanged event when implementing the INotifyPropertyChanged interface). Both the Get and Set accessors are represented by an array of SyntaxNode objects. In the Get, you typically return the property value, so here the code invokes the ReturnStatement method, which represents the return instruction plus the value or object it returns. In this case, the returned value is a field's identifier. A syntax node for an identifier is obtained by invoking the IdentifierName method, which takes an argument of type string, and still returns SyntaxNode. The Set accessors in contrast store the property value into a field via an assignment. Assignments are represented by the AssignmentStatement method, which takes two arguments, the left and right sides of the assignment. In the current case, the assignment is between two identifiers, so the code invokes IdentifierName twice, one for the left side of the assignment (the field name) and one for the right side (the property value). Because the property value is represented by the value identifier in both C# and Visual Basic, it can be hardcoded.

The next step is code generation for the Clone method, which is required by the ICloneable interface implementation. Generally speaking, a method consists of the declaration, which includes the signature and block delimiters, and of a number of statements, which make up the method body. In the current example, Clone must also implement the ICloneable.Clone method. For this reason, a convenient approach is dividing the code generation for

the method into three smaller syntax nodes. The first syntax node is the method body, which looks like the following:

```
// Generate the method body for the Clone method
var cloneMethodBody = generator.ReturnStatement(generator.
    InvocationExpression(generator.IdentifierName("MemberwiseClone")));
```

In this case, the Clone method returns the result of the invocation to the MemberwiseClone method it inherits from System.Object. For this reason, the method body is just an invocation to ReturnStatement, which you met previously. Here, the argument of the ReturnStatement is an invocation of the InvocationExpression method, which represents a method invocation and whose parameter is an identifier representing the name of the invoked method. Because the InvocationExpression argument is of type SyntaxNode, a convenient way to supply the identifier is using the IdentifierName method, passing the string representing the identifier of the method to invoke. If you had a method with a more complex method body, you'd need to generate an array of type SyntaxNode, with each node representing some code in the method body.

The next step is generating the Clone method declaration, which is accomplished like so:

```
// Generate the Clone method declaration
var cloneMethodDeclaration = generator.MethodDeclaration("Clone", null,
    null, null,
    Accessibility.Public,
    DeclarationModifiers.Virtual,
    new SyntaxNode[] { cloneMethodBody } );
```

You generate a method with the MethodDeclaration method. This takes a number of arguments, such as:

- the method name, of type String
- the method parameters, of type IEnumerable<SyntaxNode> (null in this case)
- the type parameters for generic methods, of type IEnumerable<SyntaxNode> (null in this case)
- the return type, of type SyntaxNode (null in this case)
- the accessibility level, with a value from the Accessibility enumeration
- the declaration modifiers, with one or more values from the DeclarationModifiers enumeration; in this case the modifier is virtual (Overridable in Visual Basic)
- the statements for the method body, of type SyntaxNode; in this case, the array contains one element, which is the return statement defined earlier

Because the property value is represented by the value identifier in both C# and Visual Basic, it can be hardcoded.

You'll see an example of how to add method parameters with the more specialized ConstructorDeclaration method shortly. The Clone method must implement its counterpart from the ICloneable interface, so this must be handled. What you need now is a syntax node that represents the interface name and that will also be

useful when the interface implementation is added to the Person class. This can be accomplished by invoking the IdentifierName method, which returns a proper name from the specified string:

```
// Generate a SyntaxNode for the interface's name you want to implement
var ICCloneableInterfaceType = generator.IdentifierName("ICCloneable");
```

If you wanted to import the fully qualified name, System.ICCloneable, you'd use DottedName instead of IdentifierName in order to generate a proper qualified name, but in the current example a NamespaceImportDeclaration for System was already added. At this point, you can put it all together. SyntaxGenerator has the AsPublicInterfaceImplementation and AsPrivateInterfaceImplementation methods that you use to tell the compiler that a method definition is implementing an interface, as in the following:

```
// Explicit ICloneable.Clone implementation
var cloneMethodWithInterfaceType = generator.
    AsPublicInterfaceImplementation(cloneMethodDeclaration,
    ICCloneableInterfaceType);
```

This is particularly important with Visual Basic, which explicitly requires the Implements clause. AsPublicInterfaceImplementation is the equivalent of implicit interface implementation in C#, whereas AsPrivateInterfaceImplementation is the equivalent of explicit interface implementation. Both work against methods, properties and indexers.

The next step is about generating the constructor, which is accomplished via the ConstructorDeclaration method. As with the Clone

method, the constructor's definition should be split into smaller pieces for easier understanding and cleaner code. As you'll recall from **Figure 1** and **Figure 2**, the constructor takes two parameters of type string, which are required for property initialization. So it's a good idea to generate the syntax node for both parameters first:

```
// Generate parameters for the class' constructor
var constructorParameters = new SyntaxNode[] {
    generator.ParameterDeclaration("LastName",
    generator.TypeExpression(SpecialType.System_String)),
    generator.ParameterDeclaration("FirstName",
    generator.TypeExpression(SpecialType.System_String)) };
```

As with the Clone method, the constructor's definition should be split into smaller pieces for better understanding and cleaner code.

Each parameter is generated with the ParameterDeclaration method, which takes a string representing the parameter name, and an expression representing the parameter type. Both parameters are of type String, so the code simply uses the TypeExpression method, as you already learned. The reason for packing both parameters into a SyntaxNode is that the ConstructorDeclaration wants an object of this type to represent parameters.

Now you need to construct the method body, which takes advantage of the AssignmentStatement method you saw previously, as follows:

```
// Generate the constructor's method body
var constructorBody = new SyntaxNode[] {
    generator.AssignmentStatement(generator.IdentifierName("_lastName"),
    generator.IdentifierName("LastName")),
    generator.AssignmentStatement(generator.IdentifierName("_firstName"),
    generator.IdentifierName("FirstName"))};
```

In this case there are two statements, both grouped into a SyntaxNode object. Finally, you can generate the constructor, putting together the parameters and the method body:

```
// Generate the class' constructor
var constructor = generator.ConstructorDeclaration("Person",
    constructorParameters, Accessibility.Public,
    statements:constructorBody);
```

ConstructorDeclaration is similar to MethodDeclaration, but is specifically designed to generate a .ctor method in C# and a Sub New method in Visual Basic.

## Generating a CompilationUnit

So far you've seen how to generate code for every member in the Person class. Now you need to put these members together and generate a proper SyntaxNode for the class. Class members must be supplied in the form of a SyntaxNode, and the following demonstrates how to put together all the members previously created:

```
// An array of SyntaxNode as the class members
var members = new SyntaxNode[] { lastNameField,
    firstNameField, lastNameProperty, firstNameProperty,
    cloneMethodWithInterfaceType, constructor };
```

Now you can finally generate the Person class, taking advantage of the ClassDeclaration method as follows:

```
using System;

namespace MyTypes
{
    public abstract class Person : ICloneable
    {
        private string _lastName;
        private string _firstName;
        public string LastName
        {
            get
            {
                return _lastName;
            }
            set
            {
                _lastName = (value);
            }
        }

        public string FirstName
        {
            get
            {
                return _firstName;
            }
            set
            {
                _firstName = (value);
            }
        }

        public virtual void Clone()
        {
            return MemberwiseClone();
        }

        public Person(string LastName, string FirstName)
        {
            _lastName = (LastName);
            _firstName = (FirstName);
        }
    }
}
```

Figure 5 The C# Roslyn-Generated Code for the Person Class



```
// Generate the class
var classDefinition = generator.ClassDeclaration(
    "Person", typeParameters: null,
    accessibility: Accessibility.Public,
    modifiers: DeclarationModifiers.Abstract,
    baseType: null,
    interfaceTypes: new SyntaxNode[] { ICloneableInterfaceType },
    members: members);
```

By using `SyntaxGenerator` instead of `SyntaxFactory`, you can target both C# and Visual Basic simultaneously.

As with other kinds of declarations, this method requires specifying the name, the generic type (null in this case), the accessibility level, the modifiers (Abstract in this case, or `MustInherit` in Visual Basic), base types (null in this case) and the implemented interfaces (in this case a `SyntaxNode` containing the interface name created previously as a syntax node). You might also want to encapsulate the class into a namespace. `SyntaxGenerator` includes the `NamespaceDeclaration` method, which accepts the namespace name and the `SyntaxNode` it contains. You use it like this:

```
// Declare a namespace
var namespaceDeclaration = generator.NamespaceDeclaration("MyTypes", classDefinition);
```

Compilers already know how to handle the generated syntax node for the complete namespace and nested members, and how

to perform code analysis over syntax, but sometimes you need to return this result in the form of a `CompilationUnit`, a type that represents a code file. This is typical with analyzers and code refactorings. Here's the code you write to return a `CompilationUnit`:

```
// Get a CompilationUnit (code file) for the generated code
var newNode = generator.CompilationUnit(usingDirectives, namespaceDeclaration).
    NormalizeWhitespace();
```

This method accepts one or more `SyntaxNode` instances as the argument.

## The Output in C# and Visual Basic

After all this work, you're ready to see the result. **Figure 5** shows the generated C# code for the `Person` class.

Now, simply change the language to Visual Basic in the line of code that creates a new `AdhocWorkspace`:

```
generator = SyntaxGenerator.GetGenerator(workspace, LanguageNames.VisualBasic);
```

If you re-run the code, you'll get a Visual Basic class definition, as shown in **Figure 6**.

The key point here is that, with `SyntaxGenerator`, you wrote code once and were able to generate both C# and Visual Basic code with which the Roslyn analysis APIs can work. When you're done, don't forget to invoke the `Dispose` method over the `AdhocWorkspace` instance, or simply enclose your code within a `using` statement. Because nobody is perfect and the generated code might contain errors, you can also check the `ContainsDiagnostics` property to see if any diagnostics exist in the code and get detailed information about code issues via the `GetDiagnostics` method.

## Language-Agnostic Analyzers and Refactorings

You can use the Roslyn APIs and the `SyntaxGenerator` class whenever you need to perform rich analysis over source code, but this approach is also very useful with analyzers and code refactorings. In fact, analyzers, code fixes, and refactorings have the `DiagnosticAnalyzer`, `ExportCodeFixProvider`, and `ExportCodeRefactoringProvider` attributes, respectively, each accepting the primary and secondary supported languages. By using `SyntaxGenerator` instead of `SyntaxFactory`, you can target both C# and Visual Basic simultaneously.

## Wrapping Up

The `SyntaxGenerator` class from the `Microsoft.CodeAnalysis` namespace provides a language-agnostic way of generating syntax nodes, targeting both C# and Visual Basic with one code base. With this powerful class you can generate any possible syntax element in a way that's compliant with both compilers, saving time and improving code maintainability. ■

**ALESSANDRO DEL SOLE** has been a Microsoft MVP since 2008. Awarded MVP of the Year five times, he has authored many books, eBooks, instructional videos and articles about .NET development with Visual Studio. Del Sole works as a solution developer expert for Brain-Sys (brain-sys.it), focusing on .NET development, training and consulting. You can follow him on Twitter: @progalex.

**THANKS** to the following Microsoft technical experts for reviewing this article: Anthony D. Green and Matt Warren

```
Imports System

Namespace MyTypes

    Public MustInherit Class Person
        Implements ICloneable

        Private _lastName As String

        Private _firstName As String

        Public Property LastName As String
            Get
                Return _lastName
            End Get

            Set(value As String)
                _lastName = value
            End Set
        End Property

        Public Property FirstName As String
            Get
                Return _firstName
            End Get

            Set(value As String)
                _firstName = value
            End Set
        End Property

        Public Overridable Sub Clone() Implements ICloneable.Clone
            Return MemberwiseClone()
        End Sub

        Public Sub New(LastName As String, FirstName As String)
            _lastName = LastName
            _firstName = FirstName
        End Sub
    End Class
End Namespace
```

Figure 6 The Visual Basic Roslyn-Generated Code for the `Person` Class



## DevExpress DXperience 15.2 | from \$1,439.99



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



## Help & Manual Professional | from \$586.04



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

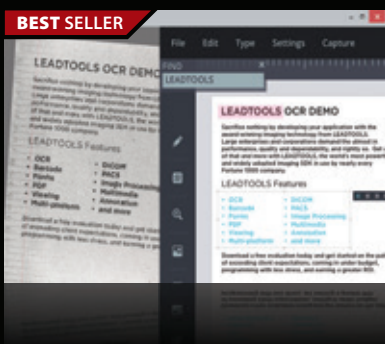


## Aspose.Total for .NET | from \$2,449.02



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



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



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

# Microsoft Azure Media Services and Power BI

Sagar Bhanudas Joshi

**Microsoft Azure Media Services** offers a rich platform for developers and independent software vendors to deliver video-on-demand and live-streaming experiences over the Web and native apps. To enrich consumer experiences and gain insights into context/application usage, it's important to weave a robust cross-platform solution in the back end for analytics and data visualization. As of this writing, the Azure Media Services platform doesn't offer analytics out of the box; hence, developers are always challenged by business demands usage data from an analytics standpoint.

This article focuses on helping developers build an analytics platform on top of Azure Media Services (and Player) to surface out usage trends. The solution space includes usage of an intermediate (Web API) service and database, with the visualization culminating with Power BI.

## This article discusses:

- Dynamic packaging and preparation for video consumption
- The Web API and the database
- Gaining insights through Power BI

## Technologies discussed:

Azure Media Services, Azure Web Apps, Azure SQL Database, Microsoft Power BI

## Code download available at:

[msdn.com/magazine/0616magcode](http://msdn.com/magazine/0616magcode)

## The Scenario

Most of the organizations creating/embedding media content need to delve on usage/analytics data in an effort to improvise end-user experience. To achieve this, developers need to record some of the key performance indicators (KPIs) related to their video/media consumption. Here are some of the most common and desirable KPIs:

- Which is the most watched video?
- How many people watched the video until completion?

## Figure 1 Code for Submitting an Encoding Job

```
static public IAsset EncodeToAdaptiveBitrateMP4s(IAsset asset,
    AssetCreationOptions options)
{
    IJob job = _context.Jobs.CreateWithSingleTask(
        "Media Encoder Standard",
        "H264 Multiple Bitrate 720p",
        asset,
        "Adaptive Bitrate MP4",
        options);
    Console.WriteLine("Submitting transcoding job...");

    job.Submit();
    job = job.StartExecutionProgressTask(
        j =>
        {
            Console.WriteLine("Job state: {0}", j.State);
            Console.WriteLine("Job progress: {0:0.##}%",
                j.GetOverallProgress());
        },
        CancellationToken.None).Result;
    Console.WriteLine("Transcoding job finished.");
    IAsset outputAsset = job.OutputMediaAssets[0];
    return outputAsset;
}
```

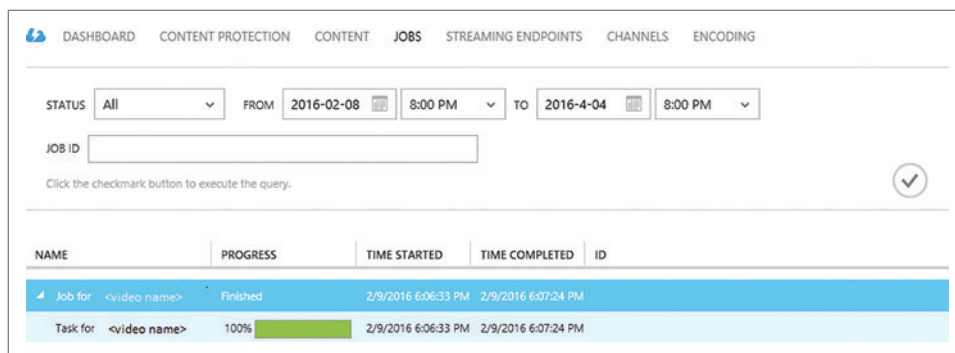


Figure 2 Azure Media Services Sample Showing Job Progress

- How many times was the video paused and at which position?
- Which is the most streamed format/bit rate?
- Platform information and other demographic details.

In order for you to decide the right mix of services and tools that will help deliver these KPIs, a significant amount of time and energy is spent to either create each component from the ground up or build on top of proven platform services.

Thankfully, the Azure Cloud Platform has a rich set of services that can be leveraged in the overall solution design to capture and address analytics requirements. The following components will be used for this scenario:

- Azure Media Services
- Azure Media Player
- Azure Web Apps
- Azure SQL Database
- Power BI

The best part of using Azure Media Services is that you can take your video/audio content and prepare it for consumption on various platforms.

The idea here is to capture raw data from Azure Media Player and feed it back to a middle-tier (Web API), which brokers the connection with the managed Azure SQL (reporting) database. You then connect Power BI to the data sources within the reporting database for surfacing out the trends of media usage/consumption through visualizations.

Further sections in this article detail the actual implementation of enabling this scenario.

## Dynamic Packaging and Preparation for Video Consumption

The best part of using Azure Media Services is that you can take your video/audio content and prepare it for consumption on various platforms. You can achieve this either by using the Azure Portal or by

performing the same steps from code if you need to automate the steps/solution. This section of the scenario discusses the following tasks:

- Identify the content (in this case, I'll choose a demo video content) to be consumed on demand
- Upload the video to Azure Media Services (backed by an Azure Storage account)
- Monitor the upload progress and submit the job for dynamic packaging

- Get the relevant URLs for consumption on various platforms

I'll use the Azure sample located at [bit.ly/22ly1ST](http://bit.ly/22ly1ST) to get started with the C# console app for achieving these tasks. Though this sample leverages NuGet packages and the C# program, you can use SDKs available in other languages, as well. The most important code here is to submit the video for encoding and get the URLs, as shown in Figure 1.

If you're new to Azure Media Services, here's a quick list of terminologies:

- Asset (or IAsset)—an entity representing a Media package with Azure Media Services. It may contain one or more content files.
- Job (or IJob)—an entity representing a unit of "encoding" work to be performed by the Azure Media Service. Think of it as converting a file from one format to another.
- Adaptive Bitrate—an encoding format that adapts to CPU/network capability of the target system and delivers content matching the criteria that best suits the device. You just have to create the Adaptive Bitrate files and Azure Media Services will identify the right bitrate to be streamed to the client device.

Now, putting the pieces together from the code in Figure 1, the function submits an "asset" to the Media Services Standard Encoder for conversion to the source format Adaptive Bitrate MP4 asset.

The sample also shows job progress, or you can track it through the Azure Portal as shown in Figure 2 in the "jobs" section. (I have blanked out the ID field values.)

Figure 3 The HTML Front-End Code

```
<!DOCTYPE html>
<html>
<head>
  <title>Welcome to the awesome world of Azure Media Services</title>
  <link href="http://amp.azure.net/libs/amp/1.6.3/skins/amp-default/
    azuremediaplayer.min.css"
    rel="stylesheet">
  <script src="http://amp.azure.net/libs/amp/1.6.3/azuremediaplayer.min.
    js"></script>
  <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.1.min.
    js"></script>
  <meta charset="utf-8" />
</head>
<body>
  <video id="azuremediaplayer" class="azuremediaplayer
    amp-default-skin amp-big-play-centered"
    tabindex="0"></video>
  <script src="scripts/Player/Player.js"></script>
</body>
</html>
```



Now that you've successfully created a media asset ready to be consumed across devices and platforms, you must be sure to let Azure Media Services know the platform of the target device so that it can stream out the right encoding and content format. For example, Windows devices generally support playing Silverlight Smooth Streaming format:

```
http://testendpoint-testaccount.streaming.mediaservices.windows.net/fecebb23-46f6-490d-8b70-203e86b0df58/assetvideo.ism/Manifest
```

where iOS devices support HLS video format and others:

```
http://testendpoint-testaccount.streaming.mediaservices.windows.net/fecebb23-46f6-490d-8b70-203e86b0df58/assetvideo.ism/Manifest(format=m3u8-aapl-v3)
```

Notice the trailing few letters of the URL, which are appended after "Manifest." These format notations help the Azure Media Services endpoint identify the content format to be streamed to

**Figure 4 Setting Video Source and Capturing Azure Media Player Attributes**

```
var myOptions = {
  "nativeControlsForTouch": true,
  controls: true,
  autoplay: true,
  width: "640",
  height: "400",
}
myPlayer = amp("azuremediaplayer", myOptions);
myPlayer.src([
{

// For feature detection, you can use libraries like Modernizr and then
// construct the URL

"src": "http://demoendpoint.streaming.mediaservices.windows.net/8frnf8nf-1jd8-19i8-009w-92073ffd3fsce/assetvideo.ism/Manifest",
"type": "application/vnd.ms-sstr+xml",

}
]);

// Events
myPlayer.addEventListener(amp.eventName.pause, _ampEventHandler);

// More events like

// Content load complete
// Media completed
// Video seek
// Page unload

function _ampEventHandler(eventDetails)
{
  var eventName = eventDetails.type;
  var pauseTime = eventDetails.presentationTimeInSec;
  var title = "Hello Azure Media Service ! ";
  var ExtraData = "None";
  var dateTime = new Date().toUTCString();

  var data = {
    'MediaRef': 9999,
    'EventTime': pauseTime,
    'MediaTitle': title,
    'TimeStamp': dateTime,
    'ExtraData': ExtraData
  }

  $(function () {
$.ajax({
  type: "POST",
  data: JSON.stringify(data),
  url: "http://demowebapi.azurewebsites.net/api/MediaAnalytics",
  contentType: "application/json"
});
});
})
```

the device. Now that you have the media ready to be streamed to the device, let's create an HTML page for consuming the video through Azure Media Player ([bit.ly/1SQ8Rwr](http://bit.ly/1SQ8Rwr)).

## Consuming Media and Sending Analytics Data to the Back End

Azure Media Player is a Web video player that complements the playback of Azure Media Services content on the client side. It works with an underlying browser platform to render the video content (primarily from Azure Media Services) with minimal configuration in JavaScript code. However, the goal is not only to play the media but also to derive analytical data of the usage and log it to the back end. Because the objective is to view the aggregate data toward the end of the solution for analysis, I'll focus only on a select number of parameters to be recorded; for example, an identifier of the video content, title of the video, pause time, timestamp and extra remarks/data.

Hence, the HTML code for player looks like that in **Figure 3**.

The HTML (design) code in **Figure 3** simply declares the video element along with the Azure Media Player-specific properties to denote the UI of the player. Notice the Player.js file, which will perform the task of detecting the platform specifying the video URL, create an object analytics data and send it asynchronously to the custom back end.

The simplest code in Player.js can look something like that in **Figure 4**.

**Figure 4** shows the simplest form of attributes that can be collected from the Azure Media Player for reference. Your solution media player can attach many more event handlers and capture richer metrics (see [bit.ly/1VA5osy](http://bit.ly/1VA5osy)). Although the code references a smooth streaming format, you can detect the platform features and quickly prepare the media URL for the relevant format of the content.

In this example, a handler is attached to the "paused" event of the Azure Media Player, which calls a back-end Web API through an AJAX call (jQuery) to avoid any blocking interference with the video playback. Hence, whenever the user pauses the video, the back-end call is made to push the data asynchronously to the analytics database, without any response being passed back to the HTML page.

Because JavaScript isn't a strongly typed language, you can create and append dynamic properties to the object and set its values. The "data" object in this example represents the instance of the attributes

**Figure 5 The Web API and the Database**

```
public class MediaAnalyticsController : ApiController
{

    async public void Post([FromBody]MediaWebAPI.Models.MediaData value)
    {

        using (SqlConnection connection =
            SQLConnHelper.CreateDatabaseConnection())
        {
            await connection.OpenAsync();
            int EntryID = SQLConnHelper.SQLWriteAnalyticsDataAsync(value, connection);
            connection.Close();
            connection.Dispose();
        }
    }
}
```



# GIVE YOUR

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

## Microsoft HQ

# CODE A VOICE

MICROSOFT HQ  
AUGUST 8-12, 2016  
REDMOND, WA



## DEVELOPMENT TOPICS:

- ALM / DevOps
- Cloud Computing
- Database & Analytics
- Mobile Client
- Software Practices
- Visual Studio / .NET Framework
- Web Client
- Web Server
- Windows Client

★ **MICROSOFT-LED SESSIONS:** With all of the announcements coming out of Build, we'll be finalizing the FULL Track of Microsoft-led sessions shortly. Be sure to check [vslive.com/redmond](http://vslive.com/redmond) for session updates!

## REGISTER BY JUNE 8 AND SAVE \$400!

USE PROMO CODE **VSLJUNTI**

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



Visual Studio **LIVE!**  
EXPERT SOLUTIONS FOR .NET DEVELOPERS

[VSLIVE.COM/REDMOND](http://VSLIVE.COM/REDMOND)

# JOIN US

## ON THE **2016** CAMPAIGN FOR CODE TRAIL!



EVENT PARTNER



PLATINUM SPONSOR



SUPPORTED BY



PRODUCED BY



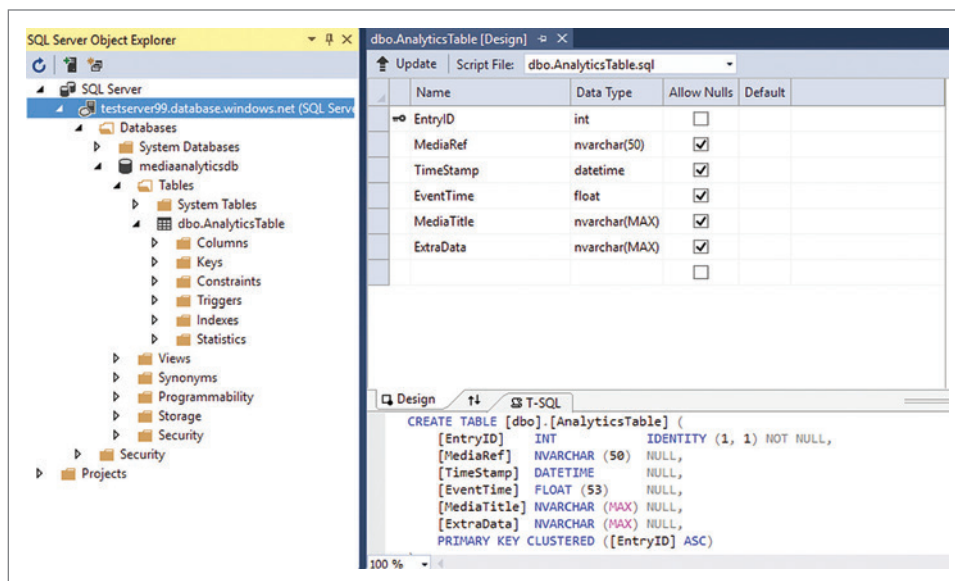


Figure 6 A Simplistic Table Design

captured from Azure Media Player and sent as a JSON string to the back end of one of the widely used formats for Web API communications.

The JSON string looks like this:

```

{
  "MediaRef":12345,
  "EventTime":15.5499976,
  "MediaTitle":"Video - 21",
  "TimeStamp":"Mon, 04 Apr 2016 16:03:36 GMT",
  "ExtraData":"None"
}

```

## The Web API and the Database

The choice of Web API and database can be across the spectrum because the reporting tool for this scenario is Power BI (powerbi.microsoft.com). However, just to keep this simple, an ASP.NET Web API project has been created to quickly set up the back-end service. This helps you easily set up communication from the front end without having to worry about writing a lot of code for connecting to database systems. Unlike the front-end AJAX call, it's important that you ensure connectivity and feedback from the database layer. Any issue with CRUD operations with DB might result in failure to capture client-side metrics. This approach also helps in testing the Web API-to-database connections without having to write complex test cases.

The code in Figure 5 shows how the back-end Web API received the "data" JSON object from the HTML front end and pushes it to the SQL Azure Database.

I'm abstracting much of the data access layer class code and the actual

hit to the database, as the options may vary here and this can be general code that can symbolize the use of multiple types of middleware plus database combinations. Upon successful entry to the database, you receive the unique ID for the row, EntryID. There are definitely other methods of receiving feedback from the database for successful operation and any of them will work here, as well.

To represent the analytics data in a structured format within the database, I've created a table closely matching the Model (in MVC) or the JSON object I received from the HTML front end, as shown in Figure 6.

Tip: Another quick way to create

Backend as a Service is to use Azure Mobile Apps, which lets you quickly provision Web API, as well as Azure SQL database, during service creation.

## Stitching the Workflow Together: Gaining Insights Through Power BI

All the phases previously were building blocks for setting the stage for demonstrating the end goal of the very topic discussed here: Analytics/BI using the Azure Media Services component.

With the combination of Azure Media Services and Azure Media Player, you were able to quickly deliver the video experience without having to worry much about the encoding and streaming capabilities. However, most organizations are also interested in gaining insights into the usage pattern of the media to surface out

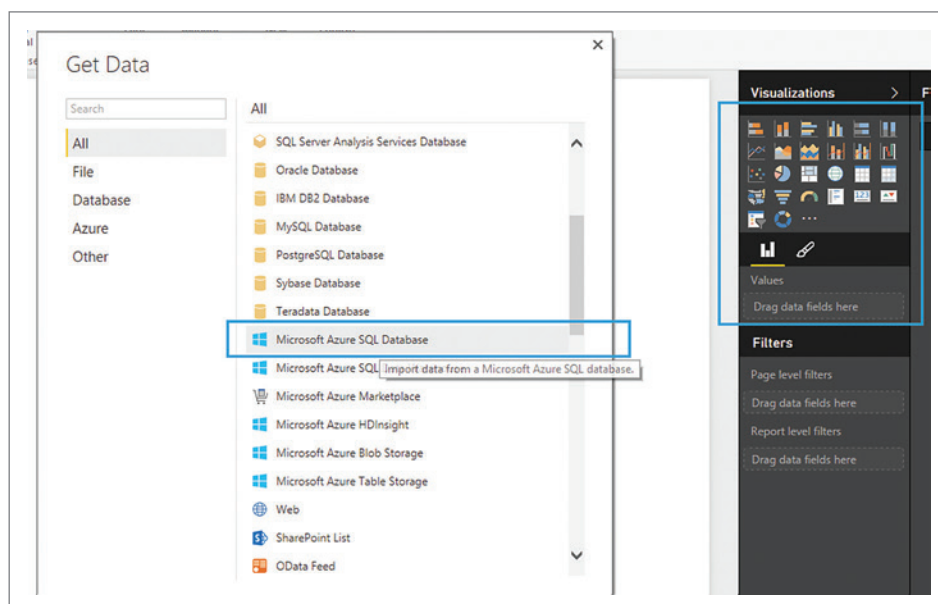


Figure 7 Power BI Getting Started



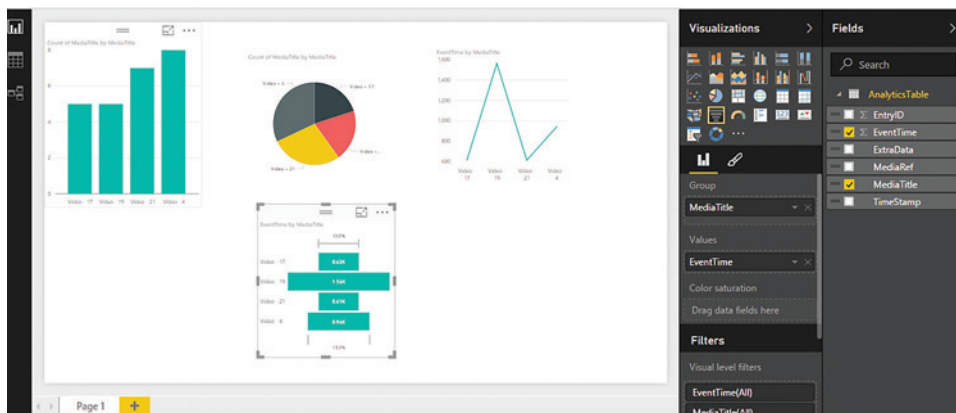


Figure 8 Creating Visualization Charts

consumption trends. Azure provides a great visualization platform to innovate and enable solution developers to create engaging experiences around its cloud-powered services. Power BI is a data visualization and analytics tool, which supports creation of interactive dashboards to monitor metrics/KPIs through easy-to-configure development environments.

To create the visualizations and surface out the trends in the media content usage, you'll use the Power BI desktop tool ([bit.ly/1S8XkL0](http://bit.ly/1S8XkL0)) to configure and create the reports, which will then be published as dashboards for monitoring the KPIs. In the Power BI desktop, click the Get Data button and select More from the dropdown. You should see the list of various supported database sources and, for this instance, you'd select Microsoft Azure SQL Database, as shown in Figure 7.

The Get Data wizard then connects to the database and you can specify additional options to import data into the designer. Power BI uses datasets to load data from data sources.

After the data is imported, you'll create a report using visualization charts/artifacts to reflect the KPIs. In a multi-table scenario, there can be relationships between tables for complex user objects that can be depicted in the model, as well. The datasets and visualizations can be paired through the top-right pane of the Power BI desktop tool, as shown in Figure 8.

ested parties. The tiles can be spread across different dashboards to form the views or KPIs that are of interest to sets of teams. For example, CXOs might be interested in viewing media completion rates, demographic details and most consumed categories, whereas IT and engineering teams can choose to monitor other KPIs such as failure rates, most streamed Bitrate, network analytics and so on.

Finally, the reports can be published over the Web at [PowerBI.com](http://PowerBI.com) for broad consumption and usage, as shown in Figure 9.

## Wrapping Up

This is just the tip of the iceberg regarding solutions and services Microsoft Azure can provide developers and solution providers. Along with video on demand, Azure Media Services can also deliver live streaming services for richer media experiences. There's ample documentation around use cases and scenarios for Azure Media Services, Web apps and databases that can help developers provision and start running applications in a matter of minutes. Continuous improvements are incorporated to Azure and the Power BI platform almost every month to enable relevant scenarios for the services.

Little customization can help you address specific needs and this article is a starting point to demonstrate how usage statistics can be captured and extended to provide media-consumption analytics. ■

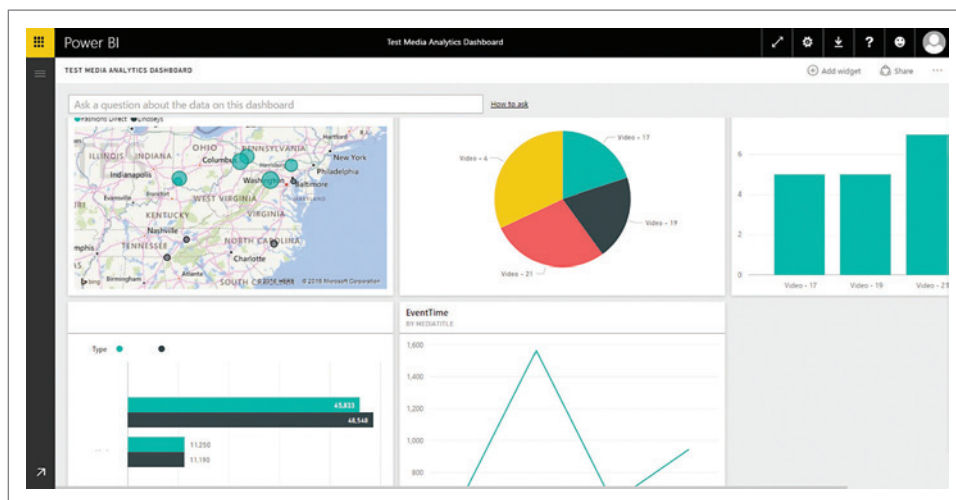


Figure 9 Power BI Dashboard Displaying Reports

**SAGAR BHANUDAS JOSHI** has worked with developers and ISVs on Universal Windows Platform and the Microsoft Azure platform for more than six years. His role includes working with ISVs and startups to help them architect, design and on-board solutions and applications to Microsoft Azure, Windows and the Office 365 platform. Joshi lives and works in Mumbai, India. Contact him on Twitter: @sagarjms handle.

**THANKS** to the following Microsoft technical expert for reviewing this article: Sandeep J. Alur

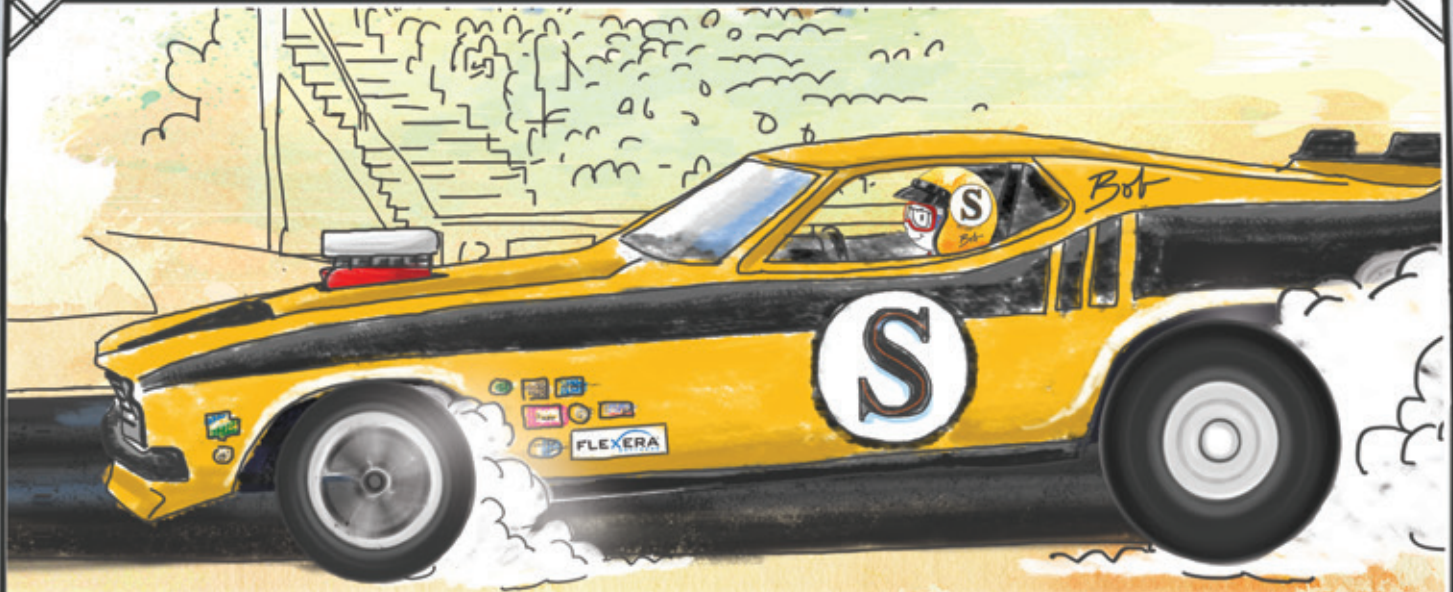




FLEXERA SOFTWARE®

InstallShield®

# GO FOR THE WIN



COMING  
SOON

## MODERN DESKTOP APPS: THE FUTURE OF DESKTOP INSTALLATIONS

### Take a Fresh Look at InstallShield

**Inside Track** – Simplify complex installs for web, cloud, virtual, PC and server

**Wow the Crowd** – Reliably deliver fast downloads and simple installations

**Pole Position** – Spend less time building installations and more time creating features that put you in the lead

**Team Win** – Distributed and agile teams can easily collaborate and save time



Download the White Paper:

**Building MSI Installer  
Updates and Patches**

[flexerasoftware.com/MSI](http://flexerasoftware.com/MSI)

**Watch the Video:**  
[flexerasoftware.com/GoForTheWin](http://flexerasoftware.com/GoForTheWin)

*Keep competitors in your rearview mirror:  
Test drive InstallShield today.*



**FLEXERA**  
SOFTWARE

**US/Canada: 1-800-809-5659**  
**International: 1-847-466-4000**

Ask to speak to an InstallShield Account Manager

© 2016 Flexera Software LLC. All other brand and product names mentioned herein may be the trademarks and registered trademarks of their respective owners.

# Using Azure App Services to Convert a Web Page to a PDF

Benjamin Perkins

**Converting** a Web page to a PDF is nothing new, but my goal—to place a link on my Web site that gave visitors a simple way to convert a specific page to a PDF document in real time—turned out to be somewhat complicated. There are numerous Web sites and open source binaries that let you do this, but I wasn't ever able to connect all the dots and get the output I wanted in the way that I wanted it.

The best, or at least my favorite, Web page-to-PDF converter is the open source program called wkhtmltopdf ([wkhtmltopdf.org](http://wkhtmltopdf.org)), which uses the command line, as shown in **Figure 1**.

However, running a program from a command line is a long way from real-time conversion with a button on a Web page.

I worked on different portions of this solution over the past months, but the execution of the wkhtmltopdf process stubbornly prevented me from achieving my goal. The question that remained unanswered was: “How can I get Microsoft Azure App Service

Web Apps to spawn this process to create the PDF?” App Service Web Apps runs within a sandbox and I knew from the start that I couldn't do that—there was zero possibility of having a request sent from a client machine starting and running a process on the server. Having worked on the IIS support team for many years, I knew that making this happen even on a standalone version of IIS would require configurations that would make security analysts lose sleep. Then I thought of WebJobs.

WebJobs are made for exactly this situation because they can run executables either continuously or when triggered from an external source; for example, manually from the Azure SDK or by using an Azure Scheduler, CRON or the Azure WebJob API ([bit.ly/1SD9gVJ](http://bit.ly/1SD9gVJ)). And, bang, there was the answer. I could call the wkhtmltopdf program from my App Service Web App using the WebJob API. The other components of the solution had already been worked out; I finally had the last piece of the puzzle, as **Figure 2** shows.

The example code contains an ASP.NET Web site with an index page that allows a user to enter a URL, send that Web page to get converted to a PDF and then download the PDF to a client device.

## This article discusses:

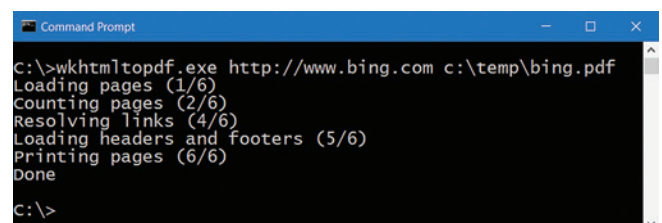
- Azure App Service Web Apps
- App Services Authentication and Authorization
- Azure Storage
- Azure WebJobs
- SignalR

## Technologies discussed:

Microsoft Azure, C#, SignalR,

## Code download available at:

[msdn.com/magazine/0616magcode](http://msdn.com/magazine/0616magcode)



```
C:\>wkhtmltopdf.exe http://www.bing.com c:\temp\bing.pdf
Loading pages (1/6)
Counting pages (2/6)
Resolving links (4/6)
Loading headers and footers (5/6)
Printing pages (6/6)
Done
C:\>
```

Figure 1 Running the wkhtmltopdf Converter from the Console

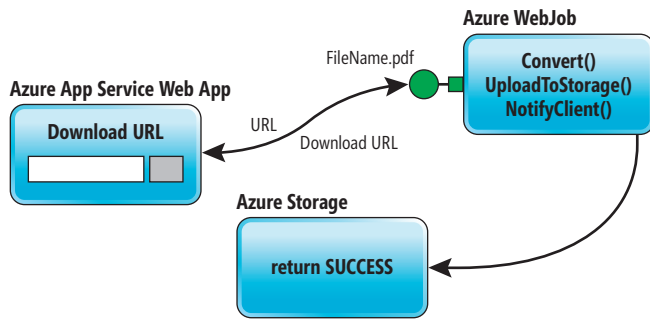


Figure 2 The Complete Solution

It takes very little effort to dynamically set this URL to the current page and have the button send the page to the WebJob API for conversion and download. The next few sections of this article discuss the technologies used to create the solution, and explain how you can build and utilize them.

## HTML-to-PDF Converter Overview

I've used numerous technologies to create the real-time HTML-to-PDF App Service Web App solution. The table in **Figure 3** presents a brief description of these technologies, and I describe them in more detail in the sections that follow.

Each section includes a functional and technical description of a technology, plus the details of coding and/or configuration requirements. I've ordered the different portions of the solution as I created them, but it could be done using a number of different sequences. The technical goal is to pass a URL to the App Service Web App and get back a PDF. Let's get started.

## Azure App Service Web App

Azure App Services lets you work with a variety of app types: Web, Mobile, Logic (preview) and API. All App Services function in the same way in the back end, with each having additional configurable capabilities on the front end. By back end I mean that App Services run in different service plans (Free, Shared, Basic, Standard and Premium) and instance sizes (F1-P4); see [bit.ly/1CVtRec](http://bit.ly/1CVtRec) for more details. The plans provide features such as deployment slots, disk-space limits, auto-scaling, maximum number of instances and so forth, and the instance sizes describe the number of dedicated CPUs, as well as the memory per App Service Plan (ASP), which is equivalent to a virtual machine (VM). And for the front end, the features for a given App Service provide specially designed capabilities for a particular App Service type to get your application deployed, configured and running in the shortest amount of time.

For the HTML-to-PDF converter, I'll use an S2 Azure App Service Web App because I don't need any of the features provided by the other App Service types.

To start, create the Web App within the Azure portal by selecting New | Web + Mobile | Web App, then provide the App name, Subscription, Resource Groups and

App Service Plan and press the Create button. Once you've created the app, you use this location to deploy the source code contained in the downloadable Visual Studio 2015 solution, `convertHTML-toPDF`. Deployment details are provided at the end of the article; you'll need to make some changes to get the code to work with your particular Web App and WebJob.

Web apps, Mobile apps and API apps include a federated identity-based feature for setting up authentication and authorization with Azure Active Directory and other identity providers like Facebook, Microsoft Live, Twitter and so on, as discussed in the next section.

## App Service Authentication and Authorization

I decided to configure the App Service Authentication / Authorization feature for my Web app because it fit nicely into the SignalR scheme, in which a display name or the identity of the client is desirable. SignalR creates a `ConnectionId` for each client, but it's friendlier and more personal to use the real name of a visitor when sending or posting messages. This can be done by capturing it from the callback of the Authentication feature and then displaying it using the SignalR code. As I implemented the Microsoft Account identity provider (IDP), the name of the authenticated visitor is returned in the `X-MS-CLIENT-PRINCIPAL-NAME` request header. The identity name is also accessible from the `System.Security.Principal.IPrincipal.Identity.Name` property.

Getting the Authentication / Authorization feature to work requires no code changes on the app back end and you can simply follow the instructions at [bit.ly/1MQZZdF](http://bit.ly/1MQZZdF). The implementation requires only that you enable App Service Authentication, accessible from the Settings blade for the given App Service, and configure one or more of the Authentication Providers, as shown in **Figure 4**.

The feature offers numerous choices for an "Action to take when request is not authorized." For example, in order to access the HTML-to-PDF Web app, you must have a Microsoft Account and be authenticated by the identity provider; no Web app code is executed before this IDP authentication takes place. In this case, pre-authentication is required because I selected "Log in with Microsoft Account" from the dropdown. All App Service resources require such authentication once an action is applied. You can configure the authentication feature so that visitors can access a login page or other endpoints of the Azure-hosted App Service, which is accomplished by selecting the Allow request (no action) item from the dropdown. However, it would then be up to the application code to restrict access to protected pages. This more granular approach is commonly achieved by checking the `Context.User.Identity.IsAuthenticated` Boolean before executing the code within the page.

Figure 3 Technologies Used in the Solution

Technology	Brief description
Azure App Service Web App (S2 Plan)	Front end that hosts SignalR code
App Service Authentication and Authorization	Confirms client identity
Azure Storage	Stores the PDF document
Azure WebJob	Converts HTML to PDF, uploads PDF to Azure Storage
Azure WebJob API	An interface for triggering a WebJob
ASP.NET SignalR	Manages response from server back to client



The last component of the no-code, real-time HTML-to-PDF conversion solution is the creation and configuration of the Azure Storage account and container.

## Azure Storage

The Azure Storage container is the location where the PDF file is stored for download. If the storage container is made public, anyone can access the files hosted in the container by referencing the filename using a URL such as `https://{storage-account}.blob.core.windows.net/{container-name}/{filename.pdf}`. Inserting, updating or removing files from the container requires an access key when performed by code. Doing so via the Azure Management Portal or from within Visual Studio can be restricted using role-based access control (RBAC) or simply by disallowing user access to the Azure subscription.

To create the storage account, select **New | Data + Storage** and the storage account. The Name attribute becomes the storage account where the container is created, and the first part of the URL: `https://{storage-account}.blob.core.windows.net`. The Deployment model attribute lets you choose either Resource manager or Classic. Unless you have existing applications deployed into a classic virtual network (VNET), it's recommended you use Resource manager for all new development activity. The Azure Resource Manager (ARM) is a more declarative approach that uses templates and scripts. In contrast, interfacing with the Classic model, commonly referred to as Azure Service Manager (ASM), is generally performed using code and libraries.

When deciding whether to choose Standard or Premium Performance, you'll want to consider cost and throughput. Standard is the most cost-effective and is optimal for applications that store infrequently accessed bulk data. Premium storage is backed by solid-state drives (SSD) that offer optimal performance for virtual machines with intensive I/O requirements.

The Replication attribute has numerous options—Local, Zone, Global and Read-Access Global—each providing a greater level of redundancy and accessibility. I used the default settings for the HTML-to-PDF solution, and selected the same Subscription, Resource group and Location as for the Web app created previously.

Finally, after successfully creating the storage account, select Blobs services from the Storage account General blade, and then add the container.

The Access Type on the New container blade can be either Private (an access key is required for all operations), Blob (allows public read access) or Container (allows public read and list access).

That's it, that's all the Azure configuration required for this solution. Let's jump into some C# code now to see how to get this real-time HTML-to-PDF conversion to work.

## Azure WebJob

The Azure WebJob feature supports running a script or executable file in a continuous, triggered or scheduled manner ([bit.ly/10g9P95](http://bit.ly/10g9P95)). Don't confuse this with a Windows Service; think of it instead as a task or batch job that needs to run at certain times or when a certain event happens. In this case, using the real-time HTML-to-PDF conversion tool triggers the WebJob using the API. Alternatively, WebJobs can be started manually via Visual Studio or by using the Azure Scheduler Job Collections capability.

The Azure App Service platform determines whether the WebJob is triggered or continuous according to the path in which the WebJob is stored. If the WebJob is to be triggered, it should be deployed into the `d:\home\site\wwwroot\app_data\jobs\triggered\{job name}` directory; if it's to be continuous, simply replace the *triggered* directory path with *continuous*. To deploy the WebJob, add the `app_data\jobs\triggered\{job name}` directory to a Web site project in Visual Studio, add the script or executable to it, similar to what's described at [bit.ly/1Uczf8L](http://bit.ly/1Uczf8L), and publish it to the Azure App Service platform.

The WebJob I created performs two tasks, converting the page at a given Web address to a PDF file and uploading that PDF file to an Azure Storage container. I could have called `wkhtmltopdf.exe` directly using the WebJob API, but I would have had to make a second API call to then upload the file to storage and that would've involved a lot of complexity in managing the file and sending the result back to the client. Therefore, I created a console application called `convertToPdf` (which you can see in the source) that performs these

two tasks, one after the other, and returns the location of the PDF file to the client that made the request.

To start `wkhtmltopdf.exe` and pass it the two required parameters—the Web address and PDF filename—I used `System.Diagnostics.ProcessStartInfo`, as shown in **Figure 5**.

The code creates an instance of the `ProcessStartInfo` class and sets the `FileName`, `Arguments` and other properties of the class. The method then starts the process identified by the `FileName` property, waits for it to complete and exits the process. By default, when the WebJob is uploaded to the Azure App Service environment, it's copied by the platform to a temporary local

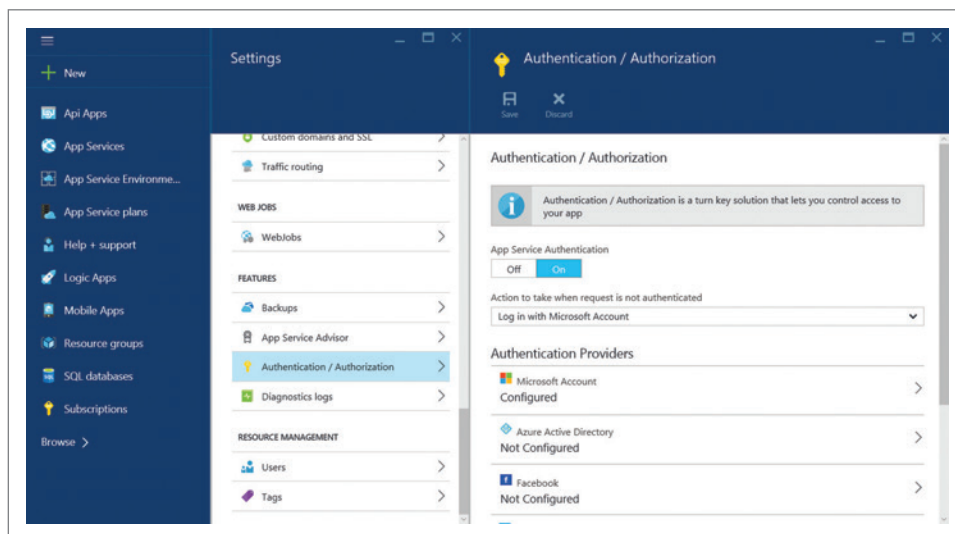


Figure 4 The App Service Authentication / Authorization Feature



# Get the Complete UI Control Toolkit.

Deliver Awesome Windows, Web, and Mobile Apps



**Newly Added Controls for Xamarin:** Start building cross-platform iOS and Android apps with C# and Visual Studio.

Develop modern, streamlined, futureproof .NET applications with ComponentOne Studio's powerful, lightweight Microsoft Visual Studio controls. Deliver quality and high-performing desktop, web, and mobile apps, including native mobile apps in C#.

**Explore the flexibility of our modular references and API across all controls and platforms**



## Grids & Data Management

Get every spreadsheet function you need with our datagrid control - without any heaving lifting.



## Data Visualization

Present large data sets out-of-the-box with our beautiful, flexible charts and gauges.



## Reporting & Documentation

Migrate reports from another platform or generate your own with a full collection of controls.



## Scheduling

Offer instant Outlook-like functionality in any application.

WinForms | WPF | UWP | ASP.NET MVC | Xamarin | ActiveX | LightSwitch | Silverlight | ASP.NET Web Forms

Download your free trial at  
[www.ComponentOne.com](http://www.ComponentOne.com)



directory—`D:\local\temp\jobs\triggered\{job name}\****\`, where `****` is a dynamically generated directory name. This is also where the PDF file is physically stored prior to being uploaded to the Azure Storage container. Because the file is local only, it's not persisted or accessible to any other instance of the Azure App Service Web App. If you're running on multiple instances, you might not see it in the local directory, but the Azure Storage container is globally accessible.

Once the PDF file is created, it needs to be uploaded to the Azure Storage container. You'll find an excellent tutorial that describes in detail how to do this at [bit.ly/10AXIQ0](http://bit.ly/10AXIQ0). In summary, the capability to create, read, update and delete content in a container is controlled by two NuGet packages, the Microsoft Azure Configuration Manager library for .NET and the Microsoft Azure Storage Client library for .NET. Both packages are referenced from the `convertToPdfWebJob` console application. To install them, right-click the console application project and then Manage NuGet Packages. Then search for and install the libraries.

I used `CloudConfigurationManager.GetSetting`, which is part of the Microsoft Azure Configuration Manager library, to retrieve the storage connection string values for making the connection to the Azure Storage container. The values are the `AccountName`, which is the Azure Storage Account name (in this case, `converthtmltopdf`), not the container name, and the `AccountKey`, which

is retrieved from the Storage Account blade by clicking on Settings | Access keys. **Figure 6** shows how to upload the PDF file to the Azure Storage container created previously.

This configuration information is used as input for the `CloudStorageAccount` class, which is part of the Microsoft Azure Storage Client library. As an alternative to the `CloudConfigurationManager` to retrieve the `StorageConnectionString` from the `App.config` file, you can use `System.Configuration.ConfigurationManager.AppSettings["StorageConnectionString"]`.

I use an instance of the `CloudStorageAccount` class to create a `CloudBlobClient`, then use the `blobClient` to get a reference to the Azure Storage container with the `GetContainerReference` method. Then, using the `GetBlockBlobReference` method of the `CloudBlobContainer` class, I create a `CloudBlockBlob` containing the name of the file being uploaded. Both of the executable files, as previously noted, are located in the `D:\local\temp\jobs\triggered\convertToPdf\****\` directory—the same place where the PDF file is stored and referenced. This is why no path to the filename is required, because the file is created in the same temporary directory as the executables. Last, I pass an instance of `System.IO.FileStream` using the `System.IO.File.OpenRead` method, and upload it to the container using the `UploadFromStream` method of the `CloudBlockBlob` class.

Once the code is complete and compiles, add both `wkhtmltopdf.exe` and the `convertToPdf.exe` to the `\app_data\jobs\triggered\convertToPdf` directory of the Visual Studio solution that will be published to the Azure App Service Web App. You can also publish just the WebJob files using an FTP tool, transferring the code directly to the Web site.

Now that the `convertToPdfWebJob` that creates and stores the PDF is complete, let's look at how to call the WebJob from C# code using the `HttpClient`. After that, all that remains is creating a SignalR-based Azure App Service Web App front end to allow a visitor to send a URL to the WebJob and get back the URL to the PDF for download.

**Figure 5 Starting wkhtmltopdf.exe**

```
static void Main(string[] args)
{
    var URL = args[0];
    var filename = args[1];
    try
    {
        using (var p = new System.Diagnostics.Process())
        {
            var startInfo = new System.Diagnostics.ProcessStartInfo
            {
                FileName = "wkhtmltopdf.exe",
                Arguments = URL + " " + filename,
                UseShellExecute = false
            };
            p.StartInfo = startInfo;
            p.Start();
            p.WaitForExit();
            p.Close();
        }
    }
    catch (Exception ex) { WriteLine($"Something Happened: {ex.Message}"); }
}
```

**Figure 6 Uploading the PDF to an Azure Storage Container**

```
static void Main(string[] args)
{
    try
    {
        CloudStorageAccount storageAccount =
            CloudStorageAccount.Parse(
                CloudConfigurationManager.GetSetting("StorageConnectionString"));
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container =
            blobClient.GetContainerReference("pdf");
        CloudBlockBlob blockBlob = container.GetBlockBlobReference(filename);
        using (var fileStream = System.IO.File.OpenRead(filename))
        {
            blockBlob.UploadFromStream(fileStream);
        }
    }
    catch (StorageException ex) { WriteLine($"StorageException: {ex.Message}"); }
    catch (Exception ex) { WriteLine($"Exception: {ex.Message}"); }
}
```

## Azure WebJob API

I wrote an article about the Azure WebJob API ([bit.ly/1SD9gVJ](http://bit.ly/1SD9gVJ)) in which I discussed how to call the API that triggers the WebJob. In essence, the WebJob API is a Web interface that executes a script or executable using the arguments passed in the URL.

Prior to creating the SignalR Hub that triggers the WebJob API, I created a simple console application consumer, shown in **Figure 7**, that calls the WebJob API. It's included in the downloadable solution and is called `convertToPDF-consumer`. This console application simplified the coding, troubleshooting and testing as it removed the SignalR feature from the scenario.

Use the `HttpClient` method of `System.Net.Http.HttpClient` class to make the request. Then use the Source Control Management (SCM)-based Azure App Service Web App URL as the Base-Address property for the request. As you might know, each Azure App Service Web App comes with an SCM URL (aka the KUDU console) that's accessible using `https://{appname}.scm.azurewebsites.net` and is the URL used for calling the WebJob API. Appending `/basicAuth` to the end of the URL allows the calling client to authenticate using a basic challenge-and-response handshake. The `userName` and `password` are the Publish Profile credentials, which are downloadable from the Azure Management Portal by navigating to the

Azure App Service Web App and selecting Get publish profile. Within the downloaded \*.PublishSettings file you'll find the userName and userPWD to use in the code. For simplicity, I hardcoded the userName and password into the application, but for the real world these should be placed in a safe location and retrieved from code, so they can be changed if desired by selecting the Reset publish profile in the Azure Management Portal. You don't want to have to deploy updated code every time something changes.

Basic authentication requires associating an ASCII-encoded Base64 string of the userName and password to the Basic header in this format: Basic userName:password. Once the header value is created using the ASCIIEncoding method of the System.Text.ASCIIEncoding class, together with the ToBase64String method of the System.Convert class, add it to a new instance of the System.Net.Http.Headers.AuthenticationHeaderValue class along with the Basic header name. Use the instance of the System.Net.Http.HttpClient class created in the using statement to add the AuthenticationHeaderValue to the DefaultRequestHeaders.Authorization property of the System.Net.Http.Headers.HttpRequestHeaders class.

For the filename I used eight characters of a GUID using the Substring method of the String class, removing the dashes from the GUID. The GUID was created using the NewGuid method of the System.Guid class, by passing an "N" parameter to the ToString method of the Guid class. Finally, I asynchronously posted to the WebJob API using the PostAsync method of the System.Net.Http.HttpClient class, passing the URL and filename as arguments of the WebJob and awaiting its completion. When the process successfully completes, the URL to the Azure Storage container with the concatenated filename is displayed to the console, otherwise, a notification is sent that the creation of the PDF failed.

**Figure 7 The Simple Console Application Consumer**

```
static async Task<string> ConvertToPDFWebJobAPIAsync(string Url)
{
    try
    {
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(
                "https://converthtmltopdf.scm.azurewebsites.net/");
            client.DefaultRequestHeaders.Accept.Clear();
            var userName = "your userName";
            var password = "your userPWD";
            var encoding = new ASCIIEncoding();
            var authHeader =
                new AuthenticationHeaderValue("Basic",
                    Convert.ToBase64String(
                        encoding.GetBytes(string.Format("{0}:{1}", userName, password))));
            client.DefaultRequestHeaders.Authorization = authHeader;
            var content = new System.Net.Http.StringContent("");
            string filename = Guid.NewGuid().ToString("N").Substring(0, 8) + ".pdf";
            HttpResponseMessage response =
                await client.PostAsync(
                    $"api/triggeredwebjobs/convertToPDF/run?arguments={Uri.EscapeDataString(filename)}, {content}");
            if (!response.IsSuccessStatusCode)
            {
                return $"Conversion for {Url} {filename} failed: " +
                    DateTime.Now.ToString();
            }
            return $"({response.StatusCode.ToString()}) your PDF can be downloaded from here:";
        }
    }
    catch (Exception ex) { return ex.Message; }
}
```

To see the status of the WebJob, go to the Azure Management Portal, navigate to the Azure App Service Web App running the WebJob, and select Settings | WebJobs. The WebJobs blade contains Name, Type, Status, and a very useful link to the WebJob execution logs. Click on the link to access a WebJob-specific KUDU console to see recent job runs, their status and a link to the actual log output of the WebJob, as shown in **Figure 8**. For example, if the WebJob is a console application, when you use the System.Console.WriteLine method to write the state of the execution to the console output window, this information is also written to the WebJob log and is viewable via the link from the Azure Management Portal.

Once this part was working as expected, all that remained was just a simple copy and paste into the SignalR solution, discussed next.

## ASP.NET SignalR

ASP.NET SignalR is an open source library for ASP.NET developers to ease the sending of real-time notifications to browser-based, mobile or .NET client applications. The server to client remote procedure call (RPC) makes use of an API that calls JavaScript functions on the client from server-side .NET code. Prior to the existence of this technology, a common approach for achieving a similar solution was using an ASP.NET UpdatePanel control that would frequently refresh itself by making a request to the server to check if there was any change in the state of the data. This was much more a PULL approach, where the client triggered the request to the server instead of the server PUSHing the real-time data to the client as soon as it became available.

The client-side JavaScript code instantiates a Hub proxy, exposes the methods the server can trigger and identifies the server-side method to call (Send) when a click event occurs:

```
var pdf = $.connection.pdfHub;
pdf.client.broadcastMessage = function (userId, message) {};
pdf.client.individualMessage = function (userId, message) {};
$('#sendmessage').click(function () {
    pdf.server.send($('#displayname').val(), $('#message').val());
});
```

The name of the Hub proxy in the client-side JavaScript is the name of the Hub created to run on the server side; in this example, the Hub is named PDFHub, and it inherits from the Microsoft.AspNet.SignalR.Hub class. The two methods exposed by the client are broadcastMessage and individualMessage; each has a function with parameters that match the pattern of the server-side Send method, userId and message. The Send method is called on the server when the send button is clicked by a visitor on the Web app. The ConvertToPDFWebJobAsync method is a cut and paste of the console application created in the previous section that calls the Azure WebJob API to convert the provided Web page into a PDF file and load it into an Azure Storage container. Last, the server-side Send method uses an instance of the Microsoft.AspNet.SignalR.Hub.Clients property, which implements the IHubCallerConnectionContext interface. The Clients property is linked to the two client-side methods and provides the information sent from the server to the appropriate clients (see **Figure 9**).

You might be wondering why I chose SignalR to consume the Azure WebJob API instead of just a simple ASP.NET Web Form or ASP.NET MVC Web application. It's true, there are numerous ways to consume an API. For example, the downloadable code for this



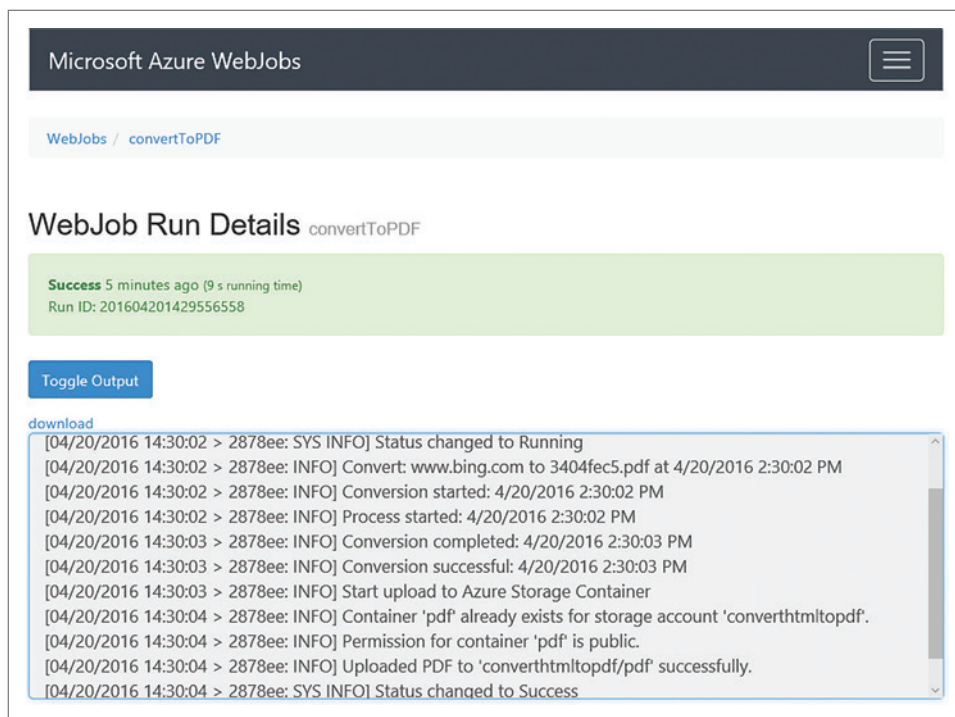


Figure 8 Azure WebJob Output Log

solution contains a console application that consumes the Azure WebJob API, so why do I use SignalR?

To answer that question, notice in **Figure 9** that when the server has a message for the connected clients, two client-side methods are invoked. First, the `broadcastMessage` method notifies all the connected clients that a specific person converted a given URL to a PDF, but it doesn't provide the link to the Azure Storage container and PDF file for download. The second client-side method is `individualMessage`, which sends the status of the HTML to PDF conversion and the link to the Azure Storage container with the concatenated PDF filename. The reason for using SignalR is to give the consuming clients a sense of social interaction by providing all the connected clients information about what's happening on the Azure App Service Web app.

Recall that previously I mentioned the `System.Security.Principal.IPrincipal.Identity.Name` and noted how it made the Web app much more friendly because it could render a visitor's name to

Figure 9 The PDFHub Class

```
public class PDFHub : Hub
{
    public void Send(string userId, string message)
    {
        string name = Context.User.Identity.Name;
        string convertMessage = "no message yet";
        Task.Run(async () =>
        {
            convertMessage = await ConvertToPDFWebJobAPIAsync(message);
        }).Wait();
        Clients.All.broadcastMessage(userId, "just converted: " + message +
            " to a pdf");
        Clients.Client(Context.ConnectionId).individualMessage(
            name, convertMessage);
    }
}
```

the client instead of, for example, a unique but generic `connectionId`. The `Context.User.Identity.Name` property is used to set the name of the visitor, validated by their Microsoft Account, which adds to the social friendliness of the client.

Now all that needs to happen is to deploy the code (client code, server code and Azure WebJob) to the Azure App Service Web App platform using Visual Studio or an FTP application and test it out. Detailed instruction on how to deploy to an Azure App Service Web App can be found at [bit.ly/1nXnhmB](http://bit.ly/1nXnhmB).

## Software as a Service

While writing this article I started thinking about Software as a Service (SaaS) and whether this real-time HTML-to-PDF converter is a SaaS solution or simply an API-accessible app running in the cloud. I decided that the exposure

of the Azure WebJob API, by the name itself, makes it an API and not SaaS. For my solution, the WebJob API is exposed through a URL and protected by Basic authentication. The API is available for other consumers to build on top of it or to add functionality to their applications, which is the definition of an API. However, as soon as there's a consumer for the API, additional features are added around the consumed API that can be used by multiple online users, so it matches the definition of SaaS. Therefore, the Azure WebJob API alone is simply an API, while my ASP.NET SignalR client running on the Azure App Service Web App platform is a SaaS solution. Sure it's not OneDrive, Office 365, CRM Dynamics Online or Hotmail, but if you need to convert a Web site to a PDF really quick, you know where to come.

## Wrapping It up

This article explored three Azure features: an Azure App Service Web App; Azure Service Authentication and Authorization; and an Azure Storage account and container. These features are the platform that support the Azure WebJob, expose the Azure WebJob API and host the ASP.NET SignalR browser-based consumer. I discussed each of the features and the steps needed to configure them. I also described the code for the Azure WebJob, the code for calling the Azure WebJob API, and the ASP.NET SignalR client. ■

**BENJAMIN PERKINS** is an escalation engineer at Microsoft and author of four books on C#, IIS, NHibernate and Microsoft Azure. He recently coauthored *Beginning C# 6 Programming with Visual Studio 2015* (John Wiley & Sons). Reach him at [benperk@microsoft.com](mailto:benperk@microsoft.com).

**THANKS** to the following Microsoft technical expert for reviewing this article:  
Richard Marr



# JOIN US on the CAMPAIGN TRAIL in 2016!

	<p><b>JUNE 13 - 16</b> HYATT CAMBRIDGE, MA <b><a href="http://vslive.com/boston">vslive.com/boston</a></b> See pages 60 – 61 for more info</p>	
	<p><b>AUGUST 8 - 12</b> MICROSOFT HQ, REDMOND, WA <b><a href="http://vslive.com/redmond">vslive.com/redmond</a></b> See pages 70 – 73 for more info</p>	
	<p><b>SEPTEMBER 26 - 29</b> HYATT ORANGE COUNTY, CA – A DISNEYLAND® GOOD NEIGHBOR HOTEL <b><a href="http://vslive.com/anaheim">vslive.com/anaheim</a></b> See pages 44 – 45 for more info</p>	
	<p><b>OCTOBER 3 - 6</b> RENAISSANCE, WASHINGTON, D.C. <b><a href="http://vslive.com/dc">vslive.com/dc</a></b> See pages 46 – 47 for more info</p>	
	<p><b>PART OF LIVE! 360</b></p> <p><b>DECEMBER 5 - 9</b> LOEWS ROYAL PACIFIC ORLANDO, FL <b><a href="http://vslive.com/orlando">vslive.com/orlando</a></b> See pages 78 – 79 for more info</p>	

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vsive – @VSLive



facebook.com – Search “VSLive”



linkedin.com – Join the  
“Visual Studio Live” group!

**VSLIVE.COM**



CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

# CODE

## Anaheim

### FOR A BETTER TOMORROW



SEPT. 26-29, 2016

HYATT REGENCY  
DISNEYLAND®  
GOOD NEIGHBOR HOTEL

## Visual Studio **LIVE!**

EXPERT SOLUTIONS FOR .NET DEVELOPERS



**VISUAL STUDIO LIVE!** (VSLive!™) is blazing new trails on the Campaign for Code: For the first time ever, we are headed to Anaheim, CA, to bring our unique brand of unbiased developer training to Southern California. With our host hotel situated just down the street from Disneyland®, developers, software architects, engineers, designers and more can code by day, and visit the Magic Kingdom by night. From Sept. 26-29, explore how the code you write today will create a better tomorrow—not only for your company, but your career, as well!

SUPPORTED BY



msdn  
magazine

Visual Studio  
MAGAZINE



PRODUCED BY

I105MEDIA<sup>®</sup>  
YOUR GROWTH, OUR BUSINESS.

# ANAHEIM • SEPT. 26-29, 2016

HYATT REGENCY, A DISNEYLAND® GOOD NEIGHBOR HOTEL



## CONNECT WITH VISUAL STUDIO LIVE!



[twitter.com/vslive](https://twitter.com/vslive) – @VSLive



[facebook.com](https://facebook.com) – Search “VSLive”



[linkedin.com](https://linkedin.com) – Join the  
“Visual Studio Live” group!

### DEVELOPMENT TOPICS INCLUDE:

- Windows Client
- Visual Studio/.NET
- Windows Client
- Mobile Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Database and Analytics

## California code with us: register to join us today!

### REGISTER NOW AND SAVE \$300!



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

USE PROMO CODE VSLAN2

## VSLIVE.COM/ANAHEIM



VOTE "YES" FOR BETTER

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

CODE  
Washington, D.C.



Visual Studio **LIVE!**  
EXPERT SOLUTIONS FOR .NET DEVELOPERS



**VISUAL STUDIO LIVE!** (VSLive!™) is on a Campaign for Code in 2016, in support of developer education. It's only fitting that we return with our unique brand of practical, unbiased, Developer training to the nation's capital this year. From Oct. 3-6, we're offering four days of sessions, workshops and networking events to developers, software architects, engineers and designers—all designed to help you vote "yes" for better code and write winning applications across all platforms.

SUPPORTED BY



Visual Studio  
MAGAZINE



PRODUCED BY





# WASHINGTON, D.C. • OCT. 3-6, 2016

RENAISSANCE, WASHINGTON, D.C.



## CONNECT WITH VISUAL STUDIO LIVE!



[twitter.com/vslive](https://twitter.com/vslive) – @VSLive



[facebook.com](https://facebook.com) – Search “VSLive”



[linkedin.com](https://linkedin.com) – Join the  
“Visual Studio Live” group!

### DEVELOPMENT TOPICS INCLUDE:

- Visual Studio/.NET
- Windows Client
- Mobile Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Database and Analytics

## Do your developer duty: register to join us today!

**REGISTER  
NOW AND  
SAVE \$300!**



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

USE PROMO CODE VSLDC2

**VSLIVE.COM/DC**

# Speed Up Your Mobile Development Using an MBaaS Platform

Paras Wadehra

A very large percentage of mobile app development cost comes from back-end integration. Moreover, most apps have a core set of features that vary little across them. So instead of building the same features and components over and over again from scratch for every application, what if you could focus on what makes a great mobile app—the UX—and consume (instead of build) the other critical app features? That's exactly the theory behind Mobile Backend as a Service (MBaaS), which provides those important but common app features as a service for you to consume, allowing you to save massive amounts of time (and therefore money) on app development while focusing on delivering a great experience to your users.

Not only do you not have to reinvent the wheel for every app, the MBaaS platform also allows you to use a “de-coupled” development approach for the app front end and back end. This means building the front end and the back end at the same time is now possible, and connecting the two just takes “flipping the switch” when both sides are done.

## This article discusses:

- Authentication
- Data access
- Adding and editing offline data
- Push notifications
- File storage and access

## Technologies discussed:

Mobile Backend as a Service (MBaaS)

I'm going to walk you through the major steps of a sample enterprise app's development process using both “do it yourself” (DIY) and MBaaS approaches, and then compare the two and see in what kind of scenario one makes sense over the other. The high-level requirements of this use case are as follows:

1. The app needs to authenticate its users against Active Directory Federation Services (AD FS).
2. The app should connect to and display data from a SharePoint instance, but needs to filter the data rather than display the full record set from SharePoint.
3. The app should allow the user to browse the data while offline.
4. The app should allow the user to add new records while offline and automatically synchronize these records with the server when the app is online.
5. The app should receive a push notification confirming the new or updated record was successfully saved in SharePoint.
6. The app should allow the user to take a picture and upload it to the server attached to the record being created.

Now imagine you've just been handed these requirements and asked to build a mobile app. Where do you start? Let's look at the choices.

One option is to get all the back-end services like AD FS and SharePoint up and running, along with a server to host your files—pictures in this case. You'd also need to build services on top of AD FS and SharePoint to let you connect and talk to them from the mobile app. Then you'd be able to build the front-end mobile app to connect into these back-end services.

You could also start by building the front-end mobile app, having it talk to mock data and authentication sources first, then updating

the mobile app to talk to the actual data and authentication sources once the back-end services are up and running. However, this requires you to create mock data and authentication systems or services you can use to build the mobile app.

With either of these DIY options, your front-end app will need to talk the same language as the back end and has the potential to break if the back end changes, requiring you to make expensive and time-consuming fixes to the app.

In contrast, using an MBaaS platform means the front-end mobile app and the back-end services can be created at the same time by two different teams—meaning faster time to market—and the two can be connected to each other using the MBaaS platform itself. One of the several benefits of this approach is that you don't need to worry about creating mock services to mimic data and authentication sources.

Using an MBaaS platform means  
the front-end mobile app and  
the back-end services can be  
created at the same time by two  
different teams.

An MBaaS platform provides you with SDKs for both native and hybrid platforms. You simply integrate the SDK for the platform of your choice into your development environment and code against it to make it easy to consume back-end services. One of the most important features of MBaaS your apps will consume is integrating with back-end data and identity sources. The MBaaS platform also provides an abstraction layer, thereby hiding the complexities of the back end from the front-end mobile app.

Let's take a more detailed look at what each of these options entails. I'll examine each of the required features of the app and focus first on the DIY way of implementing that feature, then compare that to the MBaaS way of doing things. In the end, I'll do an overall comparison of the two approaches.

## Authentication

With the DIY approach, you first need to build up the connector to AD FS and then use that connector to authenticate the user. The connector must be consumable from a mobile app, and allow the passing of a username and password from the app back to Active Directory for authentication. Once Active Directory successfully authenticates the user, it returns an authentication token that will need to be parsed, encrypted and stored for use in future calls. In addition, if an authentication provider supports refresh tokens, code will need to be written to automatically refresh the authentication token when it expires.

With MBaaS, in contrast, you make an authentication call using the client SDK to authenticate the user, something like:

```
MBaaS.Login(redirectURI);
```

Depending on the type of authentication, the provider might present you with its own login screen (OAuth style) or an MBaaS provider might present its own version of the login screen. All MBaaS providers offer at least some no-code connectors into enterprise identity sources, including, but not limited to, AD FS. You simply supply a few configuration parameters in the MBaaS cloud portal and configure MBaaS to talk to your instance of AD FS. The types of configuration parameters you provide include provider login URI, logout URI, certificate text and name ID format URI along with redirect URIs and time-to-live (TTL) for the authentication token. You should be able to get all these values from the AD FS administrator in your enterprise. Just by setting up these configuration parameters you create a connection to your AD FS instance.

The complexity of implementing the whole authentication process has now been simplified to that one line of code. All the back-end complexities, like the authentication handshake, retrieving the authentication token, encrypting and storing it, and so forth are now taken care of by the MBaaS platform and the corresponding SDK.

## Data Access

DIY data fetching seems straightforward—until you sit down and try it. It's not too bad for data sources that expose a consumable Web service on top of the data by default. It might even be easy to connect directly to the data source from a Web-based app. However, mobile platforms typically do not have the same connectors to enterprise data sources available as Web apps. This means a custom connector needs to be written, most probably as a Web service that talks to the data source and exposes its data using normal HTTP verbs like GET, PUT, POST and DELETE. Moreover, this connector will need to be hosted somewhere. In most cases, your app will be accessed from outside your network, which means the Web service you create must be able to talk to both the outside world, as well as the inside-the-firewall enterprise world, and that means a hole probably needs to be poked in the firewall. After the custom connector has been created, you can connect to it from the mobile app and perform all the data operations accordingly. But what if you have to let the user fetch only certain record sets based on their queries? This requires building querying capabilities in the Web service, which can get complicated, especially if you need to allow the users to make complex queries.

Most MBaaS systems provide out-of-the-box connectors to several enterprise data sources, which means you simply configure them instead of creating them. For the sample use case, you'd configure the connector for SharePoint by providing configuration parameters like the host URL and username and password to connect to your instance of SharePoint. Consuming the data from SharePoint within the mobile app is as easy as writing this one line of code:

```
MBaaS.data.get(NameofDataCollection [,QueryParams] [,Options]);
```

This would normally return an array of JSON objects from your data source. One of the benefits of using an out-of-the-box connector from an MBaaS provider is that such connectors provide a way to discover all the objects in your data store so you can simply discover all SharePoint lists and select the ones you want to provide access to from the mobile app. You can also filter and orchestrate the fields being returned by SharePoint to just the few



that are needed in the mobile app so that large amounts of data sets aren't sent to the mobile device when only a small percentage might be actually useful to the app. Again, filters can be applied without having to write any custom code—a huge value proposition for mobile developers. Without an MBaaS providing such a feature set, the entire burden of filtering and orchestrating the data comes down either to the mobile app (which means greater bandwidth and battery consumption on the device—never a good experience for the mobile user) or to using a server-side script that you'll need to write, host somewhere and manage (which means ongoing work and maintenance for you). The QueryParams input to the method in the previous line of code allows you to pass in query-based parameters to allow searching and filtering of records based on the need or input of the user.

## Using a client SDK from an MBaaS provider should make it much easier to enable offline data consumption within your app.

An MBaaS platform generally also provides data and file stores built into the platform itself, so apps that don't have their own data source can use the built-in MBaaS data store to store and consume data.

### Offline Data

Just making the data available for offline consumption is actually pretty easy with the DIY approach. All you need to do is store the data on the local device, either in local storage or in a data store like SQLite, and read it from there when the user tries to access that data within your app.

However, it's not so easy to enable offline editing of existing data or adding new data and then synchronizing it with the server while taking care of conflicts that might happen along the way. As anyone who has ever tried implementing conflict resolution code will tell you, it's not the most fun thing to do. You need to check which

entities were modified locally and which were modified on the server at the same time, then figure out what attributes of each entity were modified, and if the same attributes of the same entity were modified on both the client and the server, decide which change takes precedence and save that as the value of record.

Using a client SDK from an MBaaS provider should make it much easier to enable offline data consumption within your app. Using the data consumption example from the previous section, you'd simply pass certain option values to the call, as follows:

```
MBaaS.data.get(NameofDataCollection [,QueryParams] , Data.Offline, Data.EncryptFull);
```

In this code, I enable offline data storage on the device, as well as encryption of the data. How the SDK stores the data for offline consumption varies from provider to provider, but the implementation of that shouldn't affect how you interact with the MBaaS SDK. The SDK will take care of all the complexities around offline implementation for you. Even better, enabling encryption should be just as easy and simply involves setting up an option. This makes the offline data store secure, a key requirement for most enterprise apps. The MBaaS SDK automatically decrypts data for user interaction or display on the screen and encrypts user-entered data before storing it on the device. The SDK generally takes care of enabling offline editing and the addition of new data, as well. For this to work as expected, though, the back-end systems must have LastUpdatedTime implemented for each record. Similarly, setting up further options for conflict resolution determines whether the changes on the client or the server win when a record is updated simultaneously on both sides.

### Push Notifications

Setting up push notifications for a DIY project can be a challenge. As shown in **Figure 1**, multiple steps must take place for a successful push notification channel to be set up and a push notification to be sent. (See [bit.ly/UWPPush](http://bit.ly/UWPPush) for more information.)

1. The Universal Windows Platform (UWP) app requests a push notification channel from the OS.
2. A new notification channel is created by Windows Notification Service (WNS) and returned to the calling device in the form of a Uniform Resource Identifier (URI).
3. The notification channel URI is returned by Windows to your app.
4. Your app sends the URI to your own cloud service, where it's stored so you can access the URI when you send notifications.
5. When your cloud service has a notification to send, it informs WNS using the URI registered earlier. This is done by issuing an HTTP POST request, including the notification payload, over Secure Sockets Layer (SSL).
6. WNS receives the request and routes the notification to the appropriate device.

In addition, you need to request a channel each time the app launches because channel URIs can expire and there's no guarantee a previous channel URI will still be valid. If the returned channel URI is different from the URI you've been using, you'll need to update the URI in your cloud service. You also need to map the channel URI to the device ID for the user so you can update the appropriate channel URI on your server.

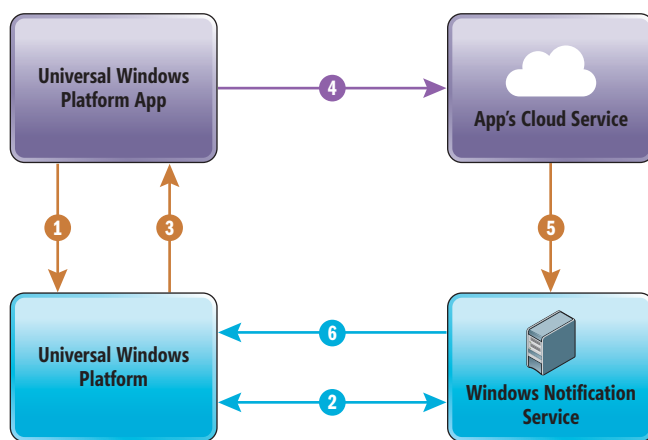


Figure 1 Flow for Setting Up and Sending Push Notifications



DOMAINS | WEBSITES | CLOUD SERVERS | WEB HOSTING

# SAFEST OF THE SAFE!



SSL CERTIFICATE  
HIGHEST SECURITY  
EXCLUSIVE TO 1&1!

## HOSTING? ONLY THE SAFEST!

1&1 provides the highest standard of protection available! Show your online visitors that their security is your top priority with:

- ✓ SSL Certificate included
- ✓ Geo-redundancy
- ✓ Certified data centers
- ✓ DDoS protection



☎ 1 (877) 461-2631



1and1.com

Visit [www.1and1.com](http://www.1and1.com) for full promotional offer details. 1&1 and the 1&1 logo are trademarks of 1&1 Internet, all other trademarks are the property of their respective owners. ©2016 1&1 Internet. All rights reserved.

	DIY	MBaaS
Request Push Notification Channel	X	✓
Set up server for managing push registration	X	✓
Upload notification channel URI to server	X	✓
Set up server-side script to determine who gets push	X	✓
Create script to send push	X	✓

Figure 2 Push Notification Complexities Taken Care of by MBaaS

Any good MBaaS platform provides ways to simplify push notification setup. Remember the steps you just saw for sending a successful push notification to your UWP app? Now take a look at **Figure 2**, which shows what you *don't* have to do if you use an MBaaS platform.

As you can see, MBaaS hides all the complexities of setting up, managing and sending push notifications to users. All this is handled with a single call made using the client-side SDK:

```
MBaaS.registerForPush();
```

and writing server-side business logic within the MBaaS platform that determines when to send a push notification and invokes the predefined method that sends out the push to the appropriate users.

## File Server

Following the DIY path means you need to host and manage your own file server to allow users to take a picture from their device and upload it to the server, and also manage user access to those files so one user can't access another user's files. In addition, you need to manage the uptime of the server, as well as scalability, performance and security updates.

With an MBaaS platform, you probably won't have to implement anything to be able to host files in the cloud. One of the features of MBaaS platforms is access to a file store out of the box. You get a fully CDN-backed file store to host all the common file types—PDFs, images, videos, office documents and the like. As with the other features, all you need to do is make a simple call to a method in the MBaaS SDK to work with files stored in the cloud. The Files API lets you upload, download, and stream files to and from the File Store, using calls like:

```
MBaaS.File.upload(fileName);
MBaaS.File.download(fileName or fileId);
```

This eliminates the headaches of managing a huge file server (depending on the number and size of the files being stored), including expanding and shrinking storage based on needs at any given point, and managing both uptime and the latency of file access.

## Other MBaaS Features

I've described the feature set of the sample app at the beginning of this article, but I'd also like to highlight a couple of other MBaaS platform features that might ease your app development:

Server-side business logic enables you to run heavy processing on the server rather than consuming power on the client device. It also lets you filter the data being sent from the data source to send only what's needed on the mobile device saving bandwidth for the

user. Moreover, you can intercept any of the calls going through, to and from the MBaaS platform.

Server-side caching enables sub-second UX scenarios. Traditional enterprise data sources often take several seconds to respond to a query, but mobile users expect a much faster response and if loading an app takes too long, they'll simply uninstall the app and go to a competitor. With the server-side caching feature, the data from the enterprise data source can be cached within the MBaaS platform for a quick response to the requests from the mobile app, and the cache can be silently updated from the original data source in the background.

## Wrapping Up

Both DIY and MBaaS approaches have a place in app development. The benefits of using an MBaaS platform are clear, but in some cases just going with a DIY approach may make sense. Such instances include one-off app development projects that already have all the necessary services in place to connect to data and identity sources. For scenarios where the service connections into such data and identity sources haven't been built yet and will need to be consumed from multiple apps, an MBaaS platform can really speed up mobile development by providing the most complex features of back-end integration out of the box. With the help of an MBaaS platform you can focus on your app's UX rather than worrying about integrating data and identity sources, encryption, offline implementation, conflict resolution, push notification setup, and other basic but critical components of mobile apps.

In the end, it's the abstraction an MBaaS platform provides that is the benefit, allowing you to change your back-end data and identity sources as often as you want. You can also change your server-side business logic without ever needing to rewrite a single line of code in your app. The mobile SDKs the MBaaS platform provide work with this abstraction to keep your app running and save you from updating, testing (including unit, integration and regression testing), and deploying the app and waiting for store approval again!

One drawback is that using an MBaaS platform might make it harder to change platforms, as the implementations of the connectors are abstracted from the user. In a DIY approach, you own the code for all connectors so it's easier to further customize it for future needs.

MBaaS does offer significant cost savings when building mobile apps, both in terms of time and effort and also from the perspective of ongoing maintenance, security, and upkeep of the services and servers involved. ■

---

**PARAS WADEHRA** is an experienced software architect who solves problems while making applications that work across the Web, desktop and mobile platforms. He has experience developing for all major mobile platforms including iOS, Android and Windows. He writes code in C#, Node.js, JavaScript, SQL and Swift to name just a few. Wadehra is currently a Microsoft MVP in Windows Development. He speaks at various industry events and conferences and can be reached on Twitter: @ParasWadehra and on LinkedIn.

---

**THANKS** to the following technical experts for reviewing this article: Hermit Dave (Associated Press Limited, UK), Kurt Monnier (Kinvey, Inc.) and Arun Nagarajan (Uber)

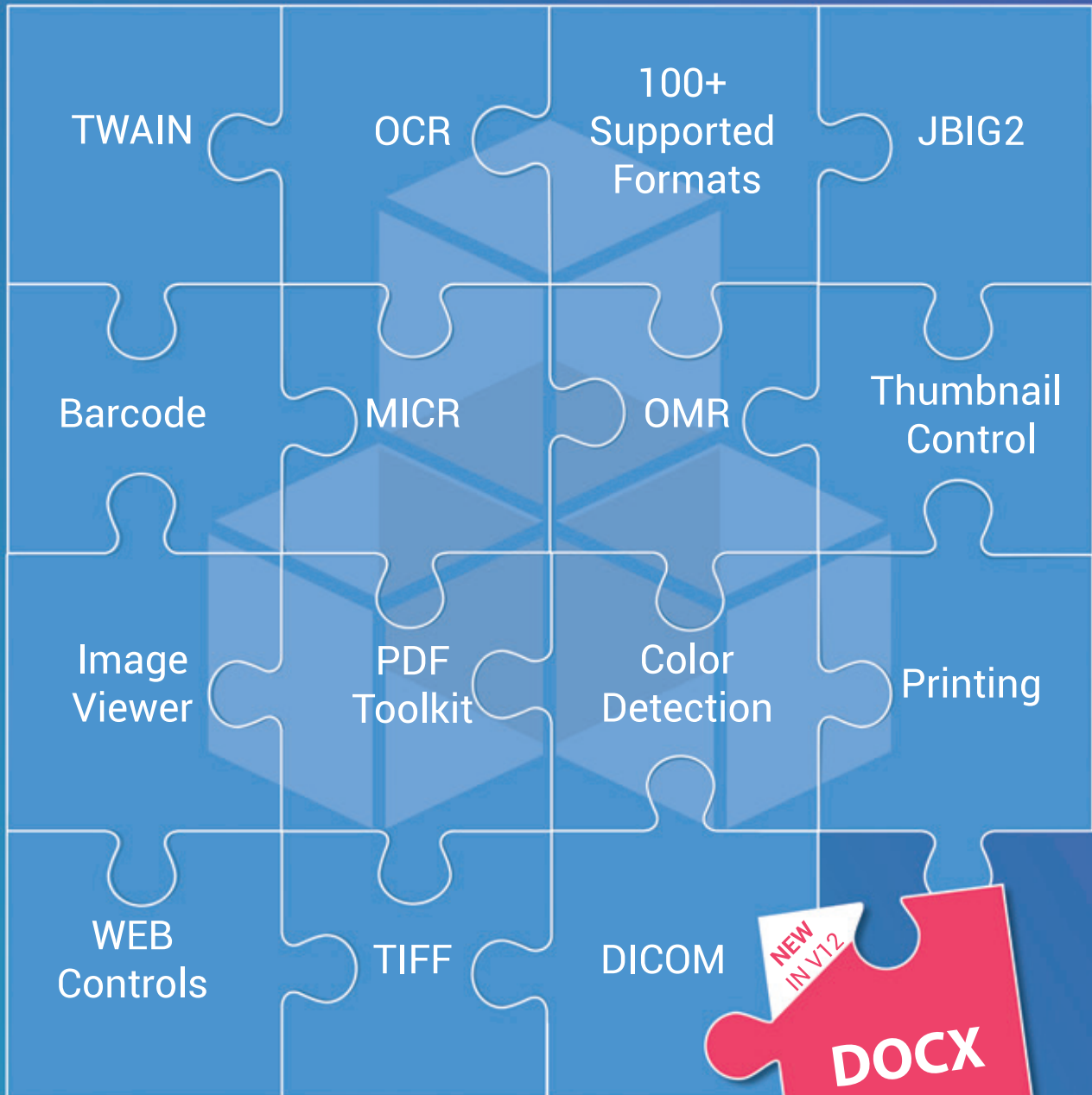
# GdPicture.NET



12

**100% ROYALTY FREE**

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



Try **GdPicture.NET V12** for **Free** for 30 days

**[www.gdpicture.com](http://www.gdpicture.com)**



# Introduction to Prediction Markets

Suppose you want to predict the outcome of an upcoming championship football game between the Xrays and the Yanks. You find a group of 20 football experts and give each of them \$500 in tokens. The experts are allowed to buy and sell shares of each of the two teams, in a way that's somewhat similar to how the stock market works.

When an expert buys shares in one team, say the Xrays, the price of a share of that team increases and the price of a share of the other team decreases. Over time, the experts will buy and sell shares of the two teams until prices stabilize, and then you'll be able to infer the probability of each team winning.

You halt trading the day before the championship game. After the game is played and the winner is determined, you pay experts who have shares in the winning team according to the last price of the team when trading closed. Because the experts know they'll be paid, they have incentive to give their true opinions during trading.

What I've just described is called a prediction market. In this article, I'll describe the math behind prediction markets and show you how to implement the key functions in code. It's unlikely you'll ever have to create a prediction market in your day-to-day job, but I think you'll find the ideas very interesting. Additionally, some of the programming techniques presented in this article can be used in more common software development scenarios.

This article assumes you have at least beginner-level coding skill, but doesn't assume you know anything about prediction markets. I present a complete demo program, and you can also get the source code from the download that accompanies this article. The demo uses C#, but you should have no trouble refactoring the code to another language if you wish.

Note that this is an informal introduction to prediction markets, intended primarily for software developers. I take some liberties with terminology and definitions in order to keep the main ideas as clear as possible.

## An Example

Prediction markets are perhaps best explained with a concrete example. Take a look at the demo program in **Figure 1**. After some preliminary messages, the demo output starts with:

```
Setting liquidity parameter = 100.0
```

```
Initial number of shares owned of teams [0] and [1] are:  
0 0
```

```
Initial inferred probabilities of winning are:  
0.5000 0.5000
```

Code download available at [msdn.com/magazine/0616magcode](http://msdn.com/magazine/0616magcode).

The liquidity parameter will be explained in detail shortly, but for now it's enough to know that liquidity controls how much market prices react to buying and selling. Larger values of liquidity produce smaller changes in prices.

Initially, no shares are owned by the experts. Because the number of shares owned for each team is the same (zero), it's reasonable that the initial inferred probability a team will win is 0.50.

The next part of the demo output is:

```
Current costs for one share of each team are:  
$0.5012 $0.5012
```

At any point in time, a share of each team has a certain price. Experts need to know this price because they're playing for real money. Because the initial probabilities of winning are equal, it's reasonable that the prices for a share of each team are also the same.

The final probabilities are the goal of the prediction market.

The next part of the demo output is:

```
Update: expert [01] buys 20 shares of team [0]
```

```
Cost of transaction to expert was: $10.50
```

Expert #1 believes that team 0, the Xrays, will win and buys 20 shares of team 0. The cost to the expert is \$10.50. Notice that the price for 20 shares (\$10.50) is not the same as 20 times the price of a single share ( $20 * \$0.5012 = \$10.02$ ). As each share is purchased, the price for an additional share of the team increases. The next part of the demo output is:

```
New number of shares owned of teams [0] and [1] are:  
20 0
```

```
New inferred probabilities of winning are:  
0.5498 0.4502
```

The demo displays the updated number of shares outstanding on each team,  $(x, y) = (20, 0)$  and computes and displays updated inferred probabilities of each team winning (0.55, 0.45). Because experts have bought more shares of team 0 than team 1, the inferred probability of team 0 winning must be greater than that of team 1. The calculation of the probabilities will be explained shortly.

Next, the demo displays:

```
Current costs for one share of each team are:  
$0.5511 $0.4514
```

```
Update: expert [02] buys 20 shares of team [1]  
Cost of transaction to expert was: $9.50
```

The new cost per share for each team is calculated and displayed. Notice that the price of a share of team 0 (\$0.55) is now quite a bit



more expensive than that of team 1 (\$0.45). This gives experts an incentive to buy shares of team 1 if they think the price is a good value relative to the likelihood of team 1 winning. In this case, the demo simulates expert #2 buying 20 shares of team 1 for a cost of \$9.50. Next:

New number of shares owned of teams [0] and [1] are:  
20 20

New inferred probabilities of winning are:  
0.5000 0.5000

There are now 20 shares outstanding for each team, so the inferred probabilities of each team winning revert to 0.50 and 0.50.

The next part of the demo output is:

Current costs for one share of each team are:  
\$0.5012 \$0.5012

Update: expert [03] buys 60 shares of team [0]  
Cost of transaction to expert was: \$34.43

New number of shares owned of teams [0] and [1] are:  
80 20

New inferred probabilities of winning are:  
0.6457 0.3543

Expert #3 believes strongly that team 0 will win, so he buys 60 shares of team 0 for a cost of \$34.43. This transaction changes the number of outstanding shares to (80, 20) and causes the new inferred probabilities of winning to move strongly toward team 0 (0.65, 0.35).

Next, expert #1 sees that the value of his shares in team 0 have risen greatly to approximately \$0.6468 per share:

Current costs for one share of each team are:  
\$0.6468 \$0.3555

Update: expert [01] sells 10 shares of team [0]  
Cost of transaction to expert was: -\$6.34

New number of shares owned of teams [0] and [1] are:  
70 20

New inferred probabilities of winning are:  
0.6225 0.3775

Expert #1 feels that team 0 is now somewhat overpriced relative to its chances of winning and sells 10 of his 20 shares, getting \$6.34 (indicated by the negative sign). The new inferred probabilities adjust back to a bit more equal, but team 0 is still predicted to win with probability 0.63.

The demo ends by closing trading. The final probabilities are the goal of the prediction market. After the game between the Xrays and the Yanks is played, experts would be paid for shares they hold in the winning team, based on the final share price of the winning team. The payments encourage the experts to give their true opinions.

## The Four Key Prediction Market Equations

A basic prediction market uses four math equations, as shown in **Figure 2**. Bear with me; the equations aren't nearly as complicated as they might first appear. There are several math models that can be used to define a prediction market. The model presented in this article is based on what's called the Logarithmic Market Scoring Rule (LMSR).

Figure 1 A Prediction Market Demo

Equation 1 is the cost function associated with a set of outstanding shares (x, y). The equation, which isn't at all obvious, comes from economics theory. From a developer's point of view, you can think of the equation as a helper function. It accepts x, which is the number of shares held of option 0, and y, which is the number of shares held of option 1, and returns a value. Variable b in all four equations is the liquidity parameter. Suppose x = 20 and y = 10. If b = 100.0, then  $C(x,y) = 100.0 * \ln(\exp(20/100) + \exp(10/100)) = 100.0 * \ln(1.22 + 1.11) = 100.0 * 0.8444 = \$84.44$ . The return value is used in equation 2.

Equation 2 is the cost of a transaction to a buyer. Suppose a current set of outstanding shares is (20, 10) and an expert buys 30 shares of option 0. The cost of that transaction to the expert is computed using

equation 2 as  $C(20+30, 10) - C(20, 10) = C(50, 10) - C(20, 10) = 101.30 - 84.44 = \$16.86$ . If an expert sells shares, the cost of the transaction will be a negative value indicating the expert is paid.

Equation 3 is technically the marginal price of option 0 based on a set of outstanding shares ( $x, y$ ). But a marginal price can be loosely interpreted as the probability that an option will win. Equation 4 is the marginal price (probability) of option 1. If you look at the two equations closely, you'll notice they must sum to 1.0, as is required for a set of probabilities.

Implementing the four key prediction market equations is straightforward. The demo program implements the cost, equation 1, as:

```
static double Cost(int[] outstanding, double liq)
{
    double sum = 0.0;
    for (int i = 0; i < 2; ++i)
        sum += Math.Exp(outstanding[i] / liq);
    return liq * Math.Log(sum);
}
```

The Cost method is virtually an exact translation of equation 1. Notice method Cost assumes there are just two options. For simplicity, no error checking is performed.

Equation 2 is also rather simple to implement:

```
static double CostOfTrans(int[] outstanding, int idx, int nShares, double liq)
{
    int[] after = new int[2];
    Array.Copy(outstanding, after, 2);
    after[idx] += nShares;
    return Cost(after, liq) - Cost(outstanding, liq);
}
```

The array named after holds the new number of outstanding shares after a transaction, and the method then just calls the Cost helper method twice. With a method to calculate the cost of a transaction in hand, it's easy to write a method that calculates the cost of buying a single share of each of the two options:

```
static double[] CostForOneShare(int[] outstanding, double liq)
{
    double[] result = new double[2];
    result[0] = CostOfTrans(outstanding, 0, 1, liq);
    result[1] = CostOfTrans(outstanding, 1, 1, liq);
    return result;
}
```

The cost of a single share can be used by experts to get an approximation of how much it would cost to buy  $n$  shares of an option.

Method Probabilities returns the two marginal prices (inferred probabilities) of each option winning in an array:

```
static double[] Probabilities(int[] outstanding, double liq)
{
    double[] result = new double[2];
    double denom = 0.0;
    for (int i = 0; i < 2; ++i)
        denom += Math.Exp(outstanding[i] / liq);
    for (int i = 0; i < 2; ++i)
        result[i] = Math.Exp(outstanding[i] / liq) / denom;
    return result;
}
```

If you compare the code for method Probabilities with equations 3 and 4, you'll see that, again, the code follows directly from the math definition.

$$(1) \quad C(x, y) = b * \ln(e^{x/b} + e^{y/b})$$

$$(2) \quad C_{trans} = C_{after} - C_{before}$$

$$(3) \quad p_x(x, y) = \frac{e^{x/b}}{e^{x/b} + e^{y/b}}$$

$$(4) \quad p_y(x, y) = \frac{e^{y/b}}{e^{x/b} + e^{y/b}}$$

Figure 2 The Four Key Prediction Market Equations

## The Demo Program

To create the demo program, I launched Visual Studio and selected the C# console application program template. I named the project PredictionMarket. The demo has no significant Microsoft .NET Framework dependencies, so any version of Visual Studio will work.

After the template code loaded, in the Solution Explorer window I renamed file Program.cs to the more descriptive PredictionMarketProgram.cs and allowed Visual Studio to automatically rename class Program for me. At the top of the source code, I deleted all using statements that referenced unneeded .NET namespaces, leaving just the reference to the top-level System namespace.

The complete demo code, with a few minor edits and some WriteLine statements deleted to save space, is presented in Figure 3. All the program control logic is in the Main method. All the prediction market functionality is in four static methods, and there are two ShowVector helper display methods.

After displaying some preliminary messages, program execution in method Main begins with:

```
double liq = 100.0;
int[] outstanding = new int[] { 0, 0 };
ShowVector(outstanding);
```

Variable liq is the liquidity parameter. A value of 100.0 is typical, but if you experiment by adjusting the value, you'll see how it affects the change in share prices after a transaction. Larger liquidity values produce smaller changes. The array named outstanding holds the total number of shares owned by all experts, on each of the two teams. Notice that the liquidity parameter has to be passed to the four static market prediction methods. An alternative design is to encapsulate the methods into a C# class and define liquidity as a member field.

Next, the number of outstanding shares is used to determine the inferred probabilities of each team winning:

```
double[] probs = Probabilities(outstanding, liq);
Console.WriteLine("Initial probabilities of winning:");
ShowVector(probs, 4, " ");
```

Next, the demo displays the costs of buying a single share of each of the two teams:

```
double[] costPerShare = CostForOneShare(outstanding, liq);
Console.WriteLine("Current costs for one share are: ");
ShowVector(costPerShare, 4, " $");
```

In a realistic prediction market, this information would be useful to the market experts to help them assess whether the share price of a team is too high or too low relative to the expert's perception that the team will win.

The demo program simulates one of the experts buying some shares, like so:

```
Console.WriteLine("Update: expert [0] buys 20 shares of team [0]");
```

```
double costTrans = CostOfTrans(outstanding, 0, 20, liq);
Console.WriteLine("Cost of transaction to expert was: $" +
    costTrans.ToString("F2"));
```

In a real prediction market, the system would have to maintain quite a bit of information about experts' account balances and the number of shares owned.

**Ultimate Data Visualization  
Controls for WPF and Windows Forms...**

# LightningChart v.7

**New! DirectX 11 rendering - Faster WPF Chart - Bindable WPF Chart - Smith Chart**



## LIGHTNING-FAST CHARTING COMPONENTS FOR SCIENCE, ENGINEERING AND TRADING

- Superior rendering performance
- Outstanding configurability
- DirectX 9 and 11 rendering engines
- WARP rendering for virtual machines
- Optimized for real-time data
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Hundreds of examples

**New!  
FREE Gauges**



**4-in-1: All editions included**

Prefer performance or binding features

WinForms

WPF

WPF properties binding

WPF properties + data binding

Performance



**Download a free 30-day trial**  
**[www.LightningChart.com](http://www.LightningChart.com)**





Figure 3 Prediction Market Demo

```
using System;
namespace PredictionMarket
{
    class PredictionMarketProgram
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Begin prediction market demo ");
            Console.WriteLine("Goal is to predict winner of Xrays");
            Console.WriteLine("vs. Yanks using expert opinions");

            double liq = 100.0;
            Console.WriteLine("Setting liquidity parameter = " +
                liq.ToString("F1"));

            int[] outstanding = new int[] { 0, 0 };
            Console.WriteLine("Initial number of shares owned are:");
            ShowVector(outstanding);

            double[] probs = Probabilities(outstanding, liq);
            Console.WriteLine("Initial probabilities of winning:");
            ShowVector(probs, 4, " ");

            Console.WriteLine("=====");

            double[] costPerShare = CostForOneShare(outstanding, liq);
            Console.WriteLine("Current costs for one share are: ");
            ShowVector(costPerShare, 4, " $");

            Console.WriteLine("Update: expert [01] buys 20 shares " +
                "of team [0]");
            double costTrans = CostOfTrans(outstanding, 0, 20, liq);
            Console.WriteLine("Cost of transaction to expert was: $" +
                costTrans.ToString("F2"));

            outstanding = new int[] { 20, 0 };
            Console.WriteLine("New number of shares owned are: ");
            ShowVector(outstanding);

            probs = Probabilities(outstanding, liq);
            Console.WriteLine("New inferred probs of winning:");
            ShowVector(probs, 4, " ");

            Console.WriteLine("=====");

            costPerShare = CostForOneShare(outstanding, liq);
            Console.WriteLine("Current costs for one share are:");
            ShowVector(costPerShare, 4, " $");

            Console.WriteLine("Update: expert [02] buys 20 shares " +
                "of team [1]");
            costTrans = CostOfTrans(outstanding, 1, 20, liq);
            Console.WriteLine("Cost of transaction to expert was: $" +
                costTrans.ToString("F2"));

            outstanding = new int[] { 20, 20 };
            Console.WriteLine("New number of shares owned are:");
            ShowVector(outstanding);

            probs = Probabilities(outstanding, liq);
            Console.WriteLine("New inferred probs of winning:");
            ShowVector(probs, 4, " ");

            Console.WriteLine("=====");

            costPerShare = CostForOneShare(outstanding, liq);
            Console.WriteLine("Current costs for one share are:");
            ShowVector(costPerShare, 4, " $");

            Console.WriteLine("Update: expert [03] buys 60 shares " +
                "of team [0]");
            costTrans = CostOfTrans(outstanding, 0, 60, liq);
            Console.WriteLine("Cost of transaction to expert was: $" +
                costTrans.ToString("F2"));

            outstanding = new int[] { 80, 20 };
            Console.WriteLine("New number of shares owned are:");
            ShowVector(outstanding);

            probs = Probabilities(outstanding, liq);
            Console.WriteLine("New inferred probs of winning:");
            ShowVector(probs, 4, " ");

            Console.WriteLine("=====");

            costPerShare = CostForOneShare(outstanding, liq);
            Console.WriteLine("Current costs for one share are: ");
            ShowVector(costPerShare, 4, " $");

            Console.WriteLine("Update: expert [01] sells 10 shares " +
                "of team [0]");
            costTrans = CostOfTrans(outstanding, 0, -10, liq);
            Console.WriteLine("Cost of transaction to expert was: $" +
                costTrans.ToString("F2"));

            outstanding = new int[] { 70, 20 };
            Console.WriteLine("New number of shares owned are:");
            ShowVector(outstanding);

            probs = Probabilities(outstanding, liq);
            Console.WriteLine("New inferred probs of winning:");
            ShowVector(probs, 4, " ");

            Console.WriteLine("=====");

            Console.WriteLine("Update: Market Closed");
            Console.WriteLine("\nEnd prediction market demo \n");
            Console.ReadLine();
        }
    } // Main()

    static double[] Probabilities(int[] outstanding,
        double liq)
    {
        double[] result = new double[2];
        double denom = 0.0;
        for (int i = 0; i < 2; ++i)
            denom += Math.Exp(outstanding[i] / liq);
        for (int i = 0; i < 2; ++i)
            result[i] = Math.Exp(outstanding[i] / liq) / denom;
        return result;
    }

    static double Cost(int[] outstanding, double liq)
    {
        double sum = 0.0;
        for (int i = 0; i < 2; ++i)
            sum += Math.Exp(outstanding[i] / liq);
        return liq * Math.Log(sum);
    }

    static double CostOfTrans(int[] outstanding, int idx,
        int nShares, double liq)
    {
        int[] after = new int[2];
        Array.Copy(outstanding, after, 2);
        after[idx] += nShares;
        return Cost(after, liq) - Cost(outstanding, liq);
    }

    static double[] CostForOneShare(int[] outstanding,
        double liq)
    {
        double[] result = new double[2];
        result[0] = CostOfTrans(outstanding, 0, 1, liq);
        result[1] = CostOfTrans(outstanding, 1, 1, liq);
        return result;
    }

    static void ShowVector(double[] vector, int dec, string pre)
    {
        for (int i = 0; i < vector.Length; ++i)
            Console.Write(pre + vector[i].ToString("F" + dec) + " ");
        Console.WriteLine("\n");
    }

    static void ShowVector(int[] vector)
    {
        for (int i = 0; i < vector.Length; ++i)
            Console.Write(vector[i] + " ");
        Console.WriteLine("\n");
    }
} // Program class
} // ns
```



Next, the number of outstanding shares is updated, like so:

```
outstanding = new int[] { 20, 0 };
Console.WriteLine("New number of shares owned on teams [0] " +
    "and [1] are: ");
ShowVector(outstanding);
```

If you refer back to the math equations in **Figure 2**, you'll notice that the number of outstanding shares for each team/option, (x, y), is needed by all equations.

After the number of outstanding shares has been updated, that information is used to estimate the revised probabilities of each team or option winning:

```
probs = Probabilities(outstanding, liq);
Console.WriteLine("New inferred probabilities of
    winning are: ");
ShowVector(probs, 4, " ");
```

Recall that these values are really marginal prices, but it's useful to think of them as probabilities. Ultimately, the purpose of a prediction market is to produce the likelihood that each team or option will win, so the final set of probabilities after the market stabilizes is what you're after.

The demo program concludes by repeating the following five operations three more times:

- Show current cost for one share of each team
- Perform a buy or sell transaction
- Show the cost of the transaction
- Update the total number of shares outstanding
- Update the probability of each team winning

Notice that the demo program begins with the probabilities of both teams being equal. This isn't realistic in many real prediction-market scenarios. It's possible to initialize a prediction market with unequal probabilities by solving for x and y in equations 3 and 4.

## Wrapping Up

The information in this article is based on the 2002 research paper, "Logarithmic Market Scoring Rules for Modular Combinatorial Information Aggregation," by Robin Hanson. You can find a PDF version of the paper in several places on the Internet by using any search tool.

Prediction markets aren't just an abstract theoretical idea. In the past few years, several companies have been created that actually implement prediction markets for real money.

An area of active research is in what are called combinatorial prediction markets. Instead of picking just one of two options to win, experts can buy shares in combination events such as team A will beat team B

and Team J will beat team K. Combinatorial prediction markets are much more complex than simple markets. ■

**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](mailto:jammc@microsoft.com).

**THANKS** to the following Microsoft technical experts who reviewed this article: Pallavi Choudhury, Gaz Iqbal, Umesh Madan and Tien Suwandy

# msdn magazine

Where you need us most.



Visual Studio **LIVE!**  
EXPERT SOLUTIONS FOR .NET DEVELOPERS

**LIVE!**  
360  
TECH EVENTS WITH PERSPECTIVE

[MSDN.microsoft.com](http://MSDN.microsoft.com)

# BETTER

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

# CODE

Boston

# FOR ALL

CAMBRIDGE, MA  
JUNE 13-16, 2016  
THE HYATT REGENCY



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.

Life, liberty, and the  
pursuit of better code:  
register to join us today!

**SESSIONS ARE FILLING UP QUICKLY—  
REGISTER TODAY!**

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

USE PROMO CODE VSLB02



PLATINUM SPONSOR

Microsoft Virtual Academy  
**LEARNING HAPPENS HERE**

SILVER SPONSOR  
 **Atalasoft**  
from Lexmark

SUPPORTED BY

 Visual Studio

 msdn  
magazine

Visual Studio  
MAGAZINE

PRODUCED BY


**1105MEDIA**  
YOUR GROWTH. OUR BUSINESS.



# BOSTON AGENDA AT-A-GLANCE

ALM / DevOps	Cloud Computing	Database and Analytics	Mobile Client	Software Practices	UX / Design	Visual Studio / .NET Framework	Web Client	Web Server
Visual Studio Live! Pre-Conference Workshops: Monday, June 13, 2016 <small>(Separate entry fee required)</small>								
7:30 AM	9:00 AM	Pre-Conference Workshop Registration • Coffee and Morning Pastries						
9:00 AM	6:00 PM	M01 Workshop: SQL Server for Developers - Andrew Brust and Leonard Label		M02 Workshop: DevOps for Your Mobile Apps and Services - Brian Randall		M03 Workshop: Native Mobile App Development for iOS, Android and Windows Using C# - Marcel de Vries and Roy Cornelissen		
6:45 PM	9:00 PM	Dine-A-Round						
Visual Studio Live! Day 1: Tuesday, June 14, 2016								
8:00 AM	9:00 AM	KEYNOTE: What's New in ASP.NET and VS 2015, Scott Hunter, Principal Program Manager Lead, ASP.NET Team, Microsoft						
9:15 AM	10:30 AM	T01 Technical Debt—Fight It with Science and Rigor - Brian Randall		T02 What's New in SQL Server 2016 - Leonard Label		T03 Getting Started with Aurelia - Brian Noyes		T04 Developer Productivity in Visual Studio 2015 - Robert Green
10:45 AM	12:00 PM	T05 Automated X-Browser Testing of Your Web Apps with Visual Studio CodedUI - Marcel de Vries		T06 Database Lifecycle Management and the SQL Server Database - Brian Randall		T07 Angular 2 101 - Deborah Kurata		⚠ T08 This session is sequestered, details will be released soon
12:00 PM	1:30 PM	Lunch • Visit Exhibitors						
1:30 PM	2:45 PM	T09 ASP.NET Core 1.0 in All Its Glory - Adam Tuliper		T10 Predicting the Future Using Azure Machine Learning - Eric D. Boyd		T11 TypeScript for C# Developers - Chris Klug		T12 Testing at the New DevOps World -Deniz Ercoskun
3:00 PM	4:15 PM	T13 Hack Proofing Your Modern Web Applications - Adam Tuliper		T14 No Schema, No Problem!—Introduction to Azure DocumentDB - Leonard Label		T15 Angular 2 Forms and Validation - Deborah Kurata		T16 Windows 10—The Universal Application: One App To Rule Them All? - Laurent Bugnion
4:15 PM	5:30 PM	Welcome Reception						
Visual Studio Live! Day 2: Wednesday, June 15, 2016								
8:00 AM	9:15 AM	W01 AngularJS & ASP.NET MVC Playing Nice - Miguel Castro		W02 Cloud Enable .NET Client LOB Applications - Robert Green		W03 Creating Great Windows Universal User Experiences - Danny Warren		W04 Building Cross-Platform C# Apps with a Shared UI Using Xamarin.Forms - Nick Landry
9:30 AM	10:45 AM	W05 Did a Dictionary and a Func Just Become the New Black in ASP.NET Development? - Chris Klug		W06 Azure Mobile Apps: APIs in the Cloud for Your Mobile Needs - Danny Warren		W07 User Experience Case Studies—The Good and The Bad - Billy Hollis		W08 Build Cross-Platform Mobile Apps with Ionic, Angular, and Cordova - Brian Noyes
11:00 AM	12:00 PM	General Session: More Personal Computing through Emerging Experiences, Tim Huckaby, Founder / Chairman - InterKnowledge & Actus Interactive Software						
12:00 PM	1:30 PM	Birds-of-a-Feather Lunch • Visit Exhibitors						
1:30 PM	2:45 PM	W09 Richer MVC Sites with Knockout JS - Miguel Castro		W10 Breaking Down Walls with Modern Identity - Eric D. Boyd		W11 Creating Dynamic Pages Using MVVM and Knockout.JS - Christopher Harrison		W12 Mobile App Development with Xamarin and F# - Rachel Reese
3:00 PM	4:15 PM	W13 Securing Client JavaScript Apps - Brian Noyes		W14 Exploring Microservices in a Microsoft Landscape - Marcel de Vries		W15 Learning to Live Without Data Grids in Windows 10 - Billy Hollis		W16 Conquer the Network—Making Your C# Mobile App More Resilient to Network Hiccups - Roy Cornelissen
4:30 PM	5:45 PM	W17 Get Good at DevOps: Feature Flag Deployments with ASP.NET, WebAPI, & JavaScript - Benjamin Day		W18 Patterns and Practices for Real-World Event-Driven Microservices - Rachel Reese		W19 Take Your Site From Ugh to OOH with Bootstrap - Philip Japikse		W20 Strike Up a Conversation with Cortana on Windows 10 - Walt Ritscher
6:15 PM	8:30 PM	VSLive!'s Boston By Land and Sea Tour						
Visual Studio Live! Day 3: Thursday, June 16, 2016								
8:00 AM	9:15 AM	TH01 Windows Presentation Foundation (WPF) 4.6 - Laurent Bugnion		TH02 Open Source Software for Microsoft Developers - Rockford Lhotka		TH03 Real World Scrum with Team Foundation Server 2015 & Visual Studio Team Services - Benjamin Day		TH04 Windows for Makers: Raspberry Pi, Arduino & IoT - Nick Landry
9:30 AM	10:45 AM	TH05 Power BI 2.0: Analytics in the Cloud and in Excel - Andrew Brust		TH06 Dependencies Demystified - Jason Bock		TH07 Automate Your Builds with Visual Studio Team Services or Team Foundation Server - Tiago Pascoal		TH08 Automated UI Testing for iOS and Android Mobile Apps - James Montemagno
11:00 AM	12:15 PM	TH09 Big Data and Hadoop with Azure HDInsight - Andrew Brust		TH10 Improving Performance in .NET Applications - Jason Bock		TH11 Cross Platform Continuous Delivery with Team Build and Release Management - Tiago Pascoal		⚠ TH12 This session is sequestered, details will be released soon
12:15 PM	1:30 PM	Lunch						
1:30 PM	2:45 PM	TH13 Top 10 Entity Framework Features Every Developer Should Know - Philip Japikse		TH14 Architecting For Failure: How to Build Cloud Applications - Michael Stiefel		TH15 JavaScript Patterns for the C# Developer - Ben Hoelting		TH16 Developing with Xamarin & Amazon AWS to Scale Native Cross-Platform Mobile Apps - James Montemagno
3:00 PM	4:15 PM	TH17 Pretty, Yet Powerful. How Data Visualization Transforms the Way We Comprehend Information - Walt Ritscher		TH18 Architects? We Don't Need No Stinkin' Architects! - Michael Stiefel		TH19 Unit Testing & Test-Driven Development (TDD) for Mere Mortals - Benjamin Day		TH20 Advanced Mobile App Development for the Web Developer - Ben Hoelting

Speakers and sessions subject to change

 **DETAILS COMING SOON!** These sessions have been sequestered by our conference chairs. Be sure to check [vslive.com/boston](http://vslive.com/boston) for session updates!

CONNECT WITH VISUAL STUDIO LIVE!

 [twitter.com/vslive](https://twitter.com/vslive) – @VSLive
  [facebook.com](https://facebook.com/vslive) – Search “VSLive”
  [linkedin.com](https://linkedin.com/vslive) – Join the “Visual Studio Live” group!

**VSLIVE.COM/BOSTON**





## How to be MEAN: Passport

Welcome back, MEANers.

I've been doing a ton of server-side work and it's getting close to the time that I start moving over to the client end of things. Before I do that, though, there's one more thing that absolutely needs discussing before I can make the transition entirely. Specifically, I need to be able to support users. Most applications (if not all of them, by this point) require some kind of user authentication mechanism to establish a user's identity, typically so that you can restrict the data that you show them or the options that you allow them to do within the system.

While it's always tempting to "roll your own," within the Node community, that's just so 2010! The right answer to any of these kinds of dilemmas is always "go to npm," and in this case, the widespread hands-down winner around authentication systems is a Node.js library called Passport.

### Passport

By this point, it should be straightforward to figure out what the first steps are for using the Passport library; find out the npm package name and "npm install." The npm package name can be discovered either by searching the online npm registry, or by visiting the Passport homepage. However, visiting [PassportJS.org](http://PassportJS.org) first yields two interesting tidbits: one, that contrary to every other Node.js package homepage ever written, the "npm install" command isn't present right there on the front page; two, that Passport apparently has this concept of "strategies," and it's important.

The reason for this is simple: When you say, "It's time to authenticate the user's credentials," that's actually a vague statement. Not only is there a variety of different credentials that might be used to authenticate, there's a thousand or more different credential stores (a la servers) against which a user might authenticate. Passport wants to be the solution to any sort of authentication against any kind of credential store—Facebook, LinkedIn, Google or your own local database—and uses a variety of different kinds of credentials, from username/password through JSON Web Tokens to HTTP Bearer headers and just about anything else that you might dream up.

This means, then, that Passport isn't just one package; there's a core passport package and then there are strategies (307 of them, in fact, at the time of this writing) for how Passport is to do the actual work of authenticating. The choice of strategy (or strategies—I'll get to that in a moment) defines the actual package required, which in turn defines what to install. (But, truth-in-advertising time here, Passport does in fact define a core package "passport" that will be used by the other strategies involved, so you can get a jump on

things by doing an "npm install --save passport" before I dive into the strategy details).

### Hello, Local

Far and away the most common strategy (particularly for systems that are being built against internal user databases/credential stores) is the "local" strategy. This is the classic, "Client sends a username and a password, and you compare it against ... well, whatever you store usernames and passwords in." It's arguably also not nearly as secure as some of the other strategies, but it's a good place to start.

Right now, in the code I've been working with, there's been no authentication whatsoever. So, let's keep things simple by just hardcoding a fixed username/password in place. Once you see how Passport works, it's relatively easy to see where the code for a database lookup would go to do the comparison, so I'm going to leave that out.

Having decided that I want to use the passport-local strategy, I begin with "npm install --save passport-local" to get the necessary passport bits in place. (Remember, the "--save" argument puts it into the package manifest file so that it'll get automatically tracked as a formal dependency.)

Once installed, I need to do three things: one, configure Passport to use the given strategy; two, establish the HTTP URL route to which the user will be sending the authentication request; three, set up the Express middleware to require authentication before allowing the user to actually access the HTTP URL in question.

### Configuration

I'll start by getting Passport loaded up in the application first. Assuming that passport and passport-local have already been installed, I need to load them into the app.js script via the usual require magic:

```
var express = require('express'),
    bodyParser = require('body-parser'),
    // ...
    passport = require('passport'),
    LocalStrategy = require('passport-local').Strategy;
```

Notice that the LocalStrategy is set slightly different; like with MongoClient before, you actually assign LocalStrategy the result of accessing the field "Strategy" out of the object that's returned from the require call. This isn't common in Node.js, but it's not so rare as to be unique. LocalStrategy in this case is going to serve as a kind of class to be instantiated (or as close to it as JavaScript can generally get).

I also need to tell the Express environment that Passport is on the job:

```
var app = express();
app.use(bodyParser.json());
app.use(passport.initialize());
```

The initialize call is fairly self-explanatory; it will prep Passport to prepare to receive incoming requests. Often, there'll be a similar call to `passport.session` to set up per-user sessions, similar to what you see in ASP.NET, but for an HTTP API like what I'm building here, that's less often necessary or desirable (I'll talk about that in a bit).

## Challenge

Next, I need to establish the callback that Passport will invoke when it receives an authentication request. This callback will do the work of looking up the user and validating the password passed in. (Or, in a more real-world scenario, looking up the user and validating that the salted password hash is the same as the salted password hash currently stored in the database—but that's really more outside the scope of Passport itself.) That's done by calling `passport.use` and passing in an instance of the Strategy to use, with the callback embedded within it, as shown in **Figure 1**.

Passport doesn't particularly care what the URL pattern is that does the actual authentication.

Several things are going on here. First, by the time the callback is invoked, Passport has already done the work of parsing the incoming request and extracting the username and password to pass in to this callback. For the LocalStrategy, Passport assumes that those values are passed in via parameters named `username` and `password`, respectively. (This is configurable in the LocalStrategy construction call, if those aren't acceptable.)

Second, the actual mechanism of verification is entirely outside of Passport's jurisdiction; it assumes the strategies will do the verification, and in this case, the "local" strategy defers that entirely to the application code. In this example, you just check against a hardcoded value, but in more conventional cases this would be a Mongo lookup for a user whose username matched what was passed in and then a check against the password.

Third, in keeping with the usual Node.js middleware style, success or failure is signaled by use of the `done` function, with the parameters passed indicating whether success or failure took place. Success

Figure 1 Establishing the Callback

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    debug("Authenticating ", username, ", ", password);
    if ((username === "sa") && (password === "nopassword")) {
      var user = {
        username: "ted",
        firstName: "Ted",
        lastName: "Neward",
        id: 1
      };
      return done(null, user);
    }
    else {
      return done(null, false, { message: "DENIED" });
    }
  }
));
```

means the second parameter is a user object that will be placed within the Express request object that's passed further on down the pipeline; failure will cause Passport to ask Express to return a 401 (Not Authorized) response, and can optionally include the failure message, usually used for "flash" messages against the UI. (If flash messages aren't being used, the message is effectively thrown away.)

## Consequences

Now, all that remains is to configure the route by which the authentication will take place:

```
app.post('/login',
  passport.authenticate('local', { session: false }),
  function(req, res) {
    debug("user ", req.user.firstName, " authenticated against the system");
    res.redirect("/persons");
  });
```

Passport doesn't particularly care what the URL pattern is that does the actual authentication; `/login` is just a convention, but `/signin` or `/user/auth` or any of a half-dozen other varieties would be entirely reasonable. The key is that the first step when resolving this route is to call the passport `authenticate` function, passing in which strategy to use (local), whether to use per-user session cookies (which, as already noted, is not particularly appropriate for an API), and the actual function to invoke if the authentication succeeds. Here, that function simply logs a message to debug and then redirects the user to the list of Persons stored in the database.

Now, I can test this by passing in either form-POSTed content or by sending in JSON content; because this is an API, it's probably better and easier to send in a JSON packet:

```
{ "username": "sa", "password": "nopassword" }
```

If the username and password match, success and a 302 redirect to `/persons` is returned; if not, then a 401 response is handed back. It works!

## Redirecting Traffic

In fact, it's a common pattern (when building a traditional server-side Web app using Express) that a successful authentication will take the user to a given route, whereas a failure should take the user to a new page, and for this reason, Passport allows for a simpler approach to `authenticate`'s callbacks:

```
app.post('/login',
  passport.authenticate('local', { successRedirect: '/',
    failureRedirect: '/login',
    failureFlash: true }
  ));
```

Here, on success, Passport will automatically redirect to the `/` URL, and on failure, back to the `/login` URL, and (in this case) with a flash message indicating that the user failed to sign in successfully.

In the case of an API, though, it's more common to hand back a JSON representation of the user object to the client for display and editing. Bear in mind, however, that nothing security-related or sensitive should ever be sent back as part of this—no passwords, in particular. The browsers are all extremely helpful in providing client-side debugging utilities, and as a result, any attacker could very easily reach into that user object held in the browser's memory and start editing away to their heart's content. That could be bad. (These JSON objects can also be tampered with "in flight," prompting most Node.js-based API systems to run over HTTPS, rather than HTTP. Fortunately, most of the time, configuring Express to run over HTTPS

instead of HTTP is more an exercise in cloud configuration than any programmatic change.) As a result, passwords should never leave the server, and “roles” (for a role-based authorization system) should always be checked from the database, not from the user object the request passed in.

As written, however, right now the API client will need to pass authentication credentials each time the “/login” route is hit, and the credentials aren’t checked on any of the other routes. While I certainly could put authentication checks on every route (and should, come to think of it), I probably don’t want to have to pass the credentials as part of every method call.

## Alternatives

Passport has this idea covered “in spades,” as they say.

First, you can always go back to turning sessions on; when sessions are on, Passport will create a unique identifier and hand it back as part of the HTTP response as a cookie. Clients are then required to hand that cookie back as part of each subsequent request. The main requirement at that point on the server end is that the Passport library needs to know how to transform a user object into an identifier, and back again; they call this serializing and deserializing a user, and it requires setting up method callbacks for each of these two Passport endpoints:

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});
```

The `serializeUser` function is designed to provide a unique identifier for the user to Passport (so I grab it out of the `user.id` field) and the `deserializeUser` function does the reverse (so I use the `id` passed in as the primary key in a database lookup for the user object as a whole).

You can turn on sessions for most, if not all, Passport strategies, but in general it works when the server is generating HTML to be interpreted directly by the browser. APIs tend not to work with cookies nearly as much, particularly because APIs are often hit by native mobile app clients as much as, or more often than, a browser-based client.

A second approach uses a different Passport strategy that relies on a “known secret” to both client and server. This can then be passed in a variety of ways. In some cases, the system maintains a known set of issued “API keys,” and you must provide that key as part of each request. This is quite common with a number of third-party REST services, but it bears a serious weakness in that if an attacker can obtain the key, the attacker can masquerade as the client until the client resets the key. Passport provides a strategy for this; use “`npm install --save passport-localapikey`.” It behaves much the same way as the Local strategy, except now the strategy authentication method will look up the API key in the database, rather than the username and password.

A similar approach makes use of JSON Web Tokens (JWTs), which are more secure, but require a much longer space to explain than what I have here; “`npm install --save passport-jwt`” brings it into the

project. JWTs are a packed set of a variety of different data elements, one of which can be a shared secret (à la API key or password), but can be verified against particular issuers, audience and more.

Or, perhaps, the goal is to not store any sort of credentials at all, but rely on third-party systems (like Facebook, Google, Twitter, LinkedIn or any of several hundred other popular sites) to do the authenticating. Passport has it covered here, as well, with specific strategies for each of these sites individually, as well as generalized OAuth 2.0 (and OpenID, for those sites that use that) strategies.

I think the point is becoming clear: If you can imagine an authentication system, Passport has a strategy already defined for it. Just “`npm install`,” set up the configuration, put the `authorize` call in the Express routes and off you go.

By the way, it seems important to point out that there are services across the Internet that will provide a single point of access control for all of these authentication issues. These “Authentication-as-a-Service” services are becoming more popular as the number of sites that people use on a regular basis proliferate and become more and more of an administrative headache. One of my favorites, Auth0 (which actually has a few ex-Microsoft folks in the technical side of the company), is a sponsor for the Passport project, and its icons and logos appear discreetly scattered throughout the Passport site. I would strongly encourage checking it out if the project doesn’t already have a pre-determined authentication strategy in place (such as a legacy system or integrating against Facebook or Dropbox, or what have you).

## Wrapping Up

Passport is arguably the most successful authentication projects ever developed, across any language or platform. It manages to provide the necessary authentication “hooks” while leaving open the actual means of authentication when you want to control that, yet slipping in and doing all that heavy lifting when you don’t. The strategy approach means it’s infinitely extensible, and can accommodate any sort of new authentication scheme that might emerge, even 20 years into the future. (Don’t laugh—all this JavaScript will, in fact, still be running 20 years from now. You watch.)

But Passport is defined almost as much by what it doesn’t do as what it does; it completely punts on any idea of role-based authorization and it doesn’t try to address any kind of encryption or cryptography. Passport is all about credentials checking, which of course by this point makes the name a lot clearer—just as when I travel to Europe, I need to show my passport to prove that I am an American citizen, Passport requires that users display their credentials so that they can prove they are citizens in good standing within the system.

Once again, I find myself out of space and time, so for now... happy coding! ■

---

**TED NEWARD** is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP and has authored and coauthored a dozen books. Reach him at [ted@tedneward.com](mailto:ted@tedneward.com) if you’re interested in having him come work with your team, or read his blog at [blogs.tedneward.com](http://blogs.tedneward.com).

---

**THANKS** to the following technical expert for reviewing this article:  
Shawn Wildermuth





Whitepaper Available  
Four Ways to Optimize ASP.NET Performance

# 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



**FREE Download**

[sales@alachisoft.com](mailto:sales@alachisoft.com)

US: +1 (925) 236 3830

[www.alachisoft.com](http://www.alachisoft.com)



# Dependency Injection with .NET Core

In my last two articles, “Logging with .NET Core” ([msdn.com/magazine/mt694089](https://msdn.com/magazine/mt694089)), and, “Configuration in .NET Core” ([msdn.com/magazine/mt632279](https://msdn.com/magazine/mt632279)), I demonstrated how .NET Core functionality can be leveraged from both an ASP.NET Core project (project.json) and the more common .NET 4.6 C# project (\*.csproj). In other words, taking advantage of the new framework isn’t limited to those who are writing ASP.NET Core projects. In this column I’m going to continue to delve into .NET Core, with a focus on .NET Core dependency injection (DI) capabilities and how they enable an inversion of control (IoC) pattern. As before, leveraging .NET Core functionality is possible from both “traditional” CSProj files and the emerging project.json type projects. For the sample code, this time I’ll be using XUnit from a project.json project.

## Why Dependency Injection?

With .NET, instantiating an object is trivial with a call to the constructor via the new operator (that is, new MyService or whatever the object type is you wish to instantiate). Unfortunately, an invocation like this forces a tightly coupled connection (a hardcoded reference) of the client (or application) code to the object instantiated, along with a reference to its assembly/NuGet package. For common .NET types this isn’t a problem. However, for types offering a “service,” such as logging, configuration, payment, notification, or even DI, the dependency may be unwanted if you want to switch the implementation of the service you use. For example, in one scenario a client might use NLog for logging, while in another they might choose Log4Net or Serilog. And, the client using NLog will prefer not to dirty up their project with Serilog, so a reference to both logging services would be undesirable.

To solve the problem of hardcoding a reference to the service implementation, DI provides a level of indirection such that rather than instantiating the service directly with the new operator, the client (or application) will instead ask a service collection or “factory” for the instance. Furthermore, rather than asking the service collection for a specific type (thus creating a tightly coupled reference), you ask for an interface (such as ILoggerFactory) with the expectation that the service provider (in this case, NLog, Log4Net or Serilog) will implement the interface.

The result is that while the client will directly reference the abstract assembly (Logging.Abstractions), defining the service interface, no references to the direct implementation will be needed.

We call the pattern of decoupling the actual instance returned to the client *Inversion of Control*. This is because rather than the client determining what is instantiated, as it does when explicitly invoking the constructor with the new operator, DI determines what will be returned. DI registers an association between the type requested by the client (generally an interface) and the type that will be returned. Furthermore, DI generally determines the lifetime of the type returned, specifically, whether there will be a single instance shared between all requests for the type, a new instance for every request, or something in between.

One especially common need for DI is in unit tests. Consider a shopping cart service that, in turn, depends on a payment service. Imagine writing the shopping cart service that leverages the payment service and trying to unit test the shopping cart service without actually invoking a real payment service. What you want to invoke instead is a mock payment service. To achieve this with DI, your code would request an instance of the payment service interface from the DI framework rather than calling, for example, new PaymentService. Then, all that’s needed is for the unit test to “configure” the DI framework to return a mock payment service.

In contrast, the production host could configure the shopping cart to use one of the (possibly many) payment service options. And, perhaps most important, the references would be only to the payment abstraction, rather than to each specific implementation.

Providing an instance of the “service” rather than having the client directly instantiating it is the fundamental principle of DI. And, in fact, some DI frameworks allow a decoupling of the host from referencing the implementation by supporting a binding mechanism that’s based on configuration and reflection, rather than a compile-time binding. This decoupling is known as the service locator pattern.

## .NET Core

### Microsoft.Extensions.DependencyInjection

To leverage the .NET Core DI framework, all you need is a reference to the Microsoft.Extensions.DependencyInjection.Abstractions NuGet package. This provides access to the IServiceCollection interface, which exposes a System.IServiceProvider from which you can call GetService<TService>. The type parameter, TService, identifies the type of the service to retrieve (generally an interface), thus the application code obtains an instance:

```
ILoggerFactory loggingFactor = serviceProvider.GetService<ILoggerFactory>();
```

Code download available at [GitHub.com/IntelliTect/Articles](https://github.com/IntelliTect/Articles).

# Document Technology for Everybody



XtremeDocumentStudio  
.NET



StarDocs API  
Server



XtremeDocumentStudio  
Java



XtremeDocumentStudio  
Delphi



Create



View



Print



Edit



Convert



Digitize

## Introducing The New Interactive Multi-Format HTML5 Document Viewer



- Responsive HTML5 control, automatically adjusts for Mobile, Desktop and Pad/Tablets.
- 100% Independent. No browser plug-ins required. No ActiveX. No Office Automation.
- Single viewer for PDF, Office documents, text files and Images.
- Text Search, On-the-fly OCR on images, PDF form-filling, and more...
- Full-fledged client-side JavaScript to configure and perform all operations.
- TypeScript Support.
- Simple, straightforward licensing.



XtremeDocumentStudio  
.NET



StarDocs API  
Server



XtremeDocumentStudio  
Java



XtremeDocumentStudio  
Delphi



There are equivalent non-generic `GetService` methods that have `Type` as a parameter (rather than a generic parameter). The generic methods allow for assignment directly to a variable of a particular type, whereas the non-generic versions require an explicit cast because the return type is `Object`. Furthermore, there are generic constraints when adding the service type so that a cast can be avoided entirely when using the type parameter.

**Figure 1 Registering and Requesting an Object from Dependency Injection**

```
public class Host
{
    public static void Main()
    {
        IServiceCollection serviceCollection = new ServiceCollection();

        ConfigureServices(serviceCollection);
        Application application = new Application(serviceCollection);

        // Run
        // ...
    }

    static private void ConfigureServices(IServiceCollection serviceCollection)
    {
        ILoggerFactory loggerFactory = new Logging.LoggerFactory();

        serviceCollection.AddInstance<ILoggerFactory>(loggerFactory);
    }
}

public class Application
{
    public IServiceProvider Services { get; set; }
    public ILogger Logger { get; set; }

    public Application(IServiceCollection serviceCollection)
    {
        ConfigureServices(serviceCollection);
        Services = serviceCollection.BuildServiceProvider();
        Logger = Services.GetRequiredService<ILoggerFactory>()
            .CreateLogger<Application>();
        Logger.LogInformation("Application created successfully.");
    }

    public void MakePayment(PaymentDetails paymentDetails)
    {
        Logger.LogInformation(
            $"Begin making a payment { paymentDetails }");
        IPaymentService paymentService =
            Services.GetRequiredService<IPaymentService>();

        // ...
    }

    private void ConfigureServices(IServiceCollection serviceCollection)
    {
        serviceCollection.AddSingleton<IPaymentService, PaymentService>();
    }
}

public class PaymentService: IPaymentService
{
    public ILogger Logger { get; }

    public PaymentService(ILoggerFactory loggerFactory)
    {
        Logger = loggerFactory?.CreateLogger<PaymentService>();
        if(Logger == null)
        {
            throw new ArgumentNullException(nameof(loggerFactory));
        }

        Logger.LogInformation("PaymentService created");
    }
}
```

If no type is registered with the collection service when calling `GetService`, it will return null. This is useful when coupled with the null propagation operator to add optional behaviors to the app. The similar `GetRequiredService` method throws an exception when the service type isn't registered.

As you can see, the code is trivially simple. However, what's missing is how to obtain an instance of the service provider on which to invoke `GetService`. The solution is simply to first instantiate `ServiceCollection`'s default constructor, then register the type you want the service to provide. An example is shown in **Figure 1**, in which you can assume each class (`Host`, `Application` and `PaymentService`) is implemented in separate assemblies. Furthermore, while the `Host` assembly knows which loggers to use, there's no reference to loggers in `Application` or `PaymentService`. Similarly, the `Host` assembly has no reference to the `PaymentServices` assembly. Interfaces are also implemented in separate "abstraction" assemblies. For example, the `ILogger` interface is defined in `Microsoft.Extensions.Logging.Abstractions` assembly.

You can think of the `ServiceCollection` type conceptually as a name-value pair, where the name is the type of an object (generally an interface) you'll later want to retrieve and the value is either the type that implements the interface or the algorithm (delegate) for retrieving that type. The call to `AddInstance`, in the `Host.ConfigureServices` method in **Figure 1**, therefore, registers that any request for the `ILoggerFactory` type return the same `LoggerFactory` instance created in the `ConfigureServices` method. As a result, both `Application` and `PaymentService` are able to retrieve the `ILoggerFactory` without any knowledge (or even an assembly/NuGet reference) to what loggers are implemented and configured. Similarly, the application provides a `MakePayment` method without any knowledge as to which payment service is being used.

Note that `ServiceCollection` doesn't provide `GetService` or `GetRequiredService` methods directly. Rather, those methods are available from the `IServiceProvider` that's returned from the `ServiceCollection.BuildServiceProvider` method. Furthermore, the only services available from the provider are those added before the call to `BuildServiceProvider`.

`Microsoft.Framework.DependencyInjection.Abstractions` also includes a static helper class called `ActivatorUtilities` that provides a few useful methods for dealing with constructor parameters that aren't registered with the `IServiceProvider`, a custom `ObjectFactory` delegate, or in situations where you want to create a default instance in the event that a call to `GetService` returns null (see [bit.ly/1Wlt4Ka#ActivatorUtilities](http://bit.ly/1Wlt4Ka#ActivatorUtilities)).

## Service Lifetime

In **Figure 1** I invoke the `IServiceCollection.AddInstance<TService>(TService implementationInstance)` extension method. `Instance` is one of four different `TService` lifetime options available with .NET Core DI. It establishes that not only will the call to `GetService` return an object of type `TService`, but also that the specific `implementationInstance` registered with `AddInstance` is what will be returned. In other words, registering with `AddInstance` saves the specific `implementationInstance` instance so it can be returned with every call to `GetService` (or `GetRequiredService`) with the `AddInstance` method's `TService` type parameter.

In contrast, the `IServiceCollection.AddSingleton<TService>` extension method has no parameter for an instance and instead relies on the `TService` having a means of instantiation via the constructor. While a default constructor works, `Microsoft.Extensions.DependencyInjection` also supports non-default constructors whose parameters are also registered. For example, you can call:

```
IPaymentService paymentService = Services.GetRequiredService<IPaymentService>()
// and DI will take care of retrieving the ILoggerFactory concrete
// instance and leveraging it when instantiating the PaymentService
// class that requires an ILoggerFactory in its constructor.
```

If there's no such means available in the `TService` type, you can instead leverage the overload of the `AddSingleton` extension method, which takes a delegate of type `Func<IServiceProvider, TService>` implementationFactory—a factory method for instantiating `TService`. Whether you provide the factory method or not, the service collection implementation ensures that it will only ever create one instance of the `TService` type, thus ensuring that there's a singleton instance. Following the first call to `GetService` that triggers the `TService` instantiation, the same instance will always be returned for the lifetime of the service collection.

`IServiceCollection` also includes the `AddTransient(Type serviceType, Type implementationType)` and `AddTransient(Type serviceType, Func<IServiceProvider, TService> implementationFactory)` extension methods. These are similar to `AddSingleton` except they return a new instance every time they're invoked, ensuring you always have a new instance of the `TService` type.

Last, there are several `AddScoped` type extension methods. These methods are designed to return the same instance within a given context and to create a new instance whenever the context—known as the scope—changes. The behavior of ASP.NET Core conceptually maps to the scoped lifetime. Essentially, a new instance is created for each `HttpContext` instance, and whenever `GetService` is called within the same `HttpContext`, the identical `TService` instance is returned.

In summary, there are four lifetime options for the objects returned from the service collection implementation: Instance, Singleton, Transient and Scoped. The last three are defined in the `ServiceLifetime` enum ([bit.ly/1SFtaG6](http://bit.ly/1SFtaG6)). Instance, however, is missing, because it's a special case of Scoped in which the context doesn't change.

Earlier I referred to the `ServiceCollection` as conceptually like a name-value pair with the `TService` type serving as the lookup. The actual implementation of the `ServiceCollection` type is done in the `ServiceDescriptor` class (see [bit.ly/1SFoDgu](http://bit.ly/1SFoDgu)). This class provides a container for the information required to instantiate the `TService`, namely the `ServiceType` (`TService`), the `ImplementationType` or `ImplementationFactory` delegate along with the `ServiceLifetime`. In addition to the `ServiceDescriptor` constructors, there are a host of static factory methods on `ServiceDescriptor` that help with instantiating the `ServiceDescriptor` itself.

Regardless of which lifetime you register your `TService` with, the `TService` itself must be a reference type, not a value type. Whenever you use a type parameter for `TService` (rather than passing `Type` as a parameter) the compiler will verify this with a generic class constraint. One thing, however, that's not verified is using a `TService`

of type object. You'll want to be sure to avoid this, along with any other non-unique interfaces (such as `Comparable`, perhaps). The reason is that if you register something of type object, no matter what `TService` you specify in the `GetService` invocation, the object registered as a `TService` type will always be returned.

## Dependency Injection for the DI Implementation

ASP.NET leverages DI to such an extent that, in fact, you can DI within the DI framework itself. In other words, you're not limited to using the `ServiceCollection` implementation of the DI mechanism found in `Microsoft.Extensions.DependencyInjection`. Rather, as long as you have classes that implement `IServiceCollection` (defined in `Microsoft.Extensions.DependencyInjection.Abstractions`; see [bit.ly/1SKdm1z](http://bit.ly/1SKdm1z)) or `IServiceProvider` (defined within the System namespace of .NET Core lib framework) you can substitute your own DI framework or leverage one of the other well-established DI frameworks including Ninject ([ninject.org](http://ninject.org), with a shout out to @IanFDavis for his work maintaining this over the years) and Autofac ([autofac.org](http://autofac.org)).

## Wrapping Up

As with .NET Core Logging and Configuration, the .NET Core DI mechanism provides a relatively simple implementation of its functionality. While you're unlikely to find the more advanced DI functionality of some of the other frameworks, the .NET Core version is lightweight and a great way to get started. Furthermore (and, again, like Logging and Configuration), the .NET Core implementation can be replaced with a more mature implementation. Thus, you might consider leveraging the .NET Core DI framework as a “wrapper” through which you can plug in other DI frameworks as the need arises in the future. In this way, you don't have to define your own “custom” DI wrapper, but can leverage .NET Core's as a standard one for which any client/application can plug in a custom implementation.

One thing to note about ASP.NET Core is that it leverages DI throughout. This is undoubtedly a great practice if you need it and it's especially important when trying to substitute mock implementations of a library in your unit tests. The drawback is that rather than a simple call to a constructor with the new operator, the complexity of DI registration and `GetService` calls is needed. I can't help but wonder if perhaps the C# language could simplify this, but, based on the current C# 7.0 design, that isn't happening any time soon. ■

---

**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](http://itl.tc/EssentialCSharp)). Contact him on Facebook at [facebook.com/Mark.Michaelis](https://facebook.com/Mark.Michaelis), on his blog at [IntelliTect.com/Mark](http://IntelliTect.com/Mark), on Twitter: @markmichaelis or via e-mail at [mark@IntelliTect.com](mailto:mark@IntelliTect.com).

---

**THANKS** to the following IntelliTect technical experts for reviewing this article: Kelly Adams, Kevin Bost, Ian Davis and Phil Spokas



# GIVE YOUR

CAMPAIGN FOR CODE 2016 ★ VISUAL STUDIO LIVE!

## Microsoft HQ

# CODE A VOICE



## Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS



**VISUAL STUDIO LIVE!** is on a Campaign for Code in 2016, and we're taking a swing through our home state, rallying @ **Microsoft Headquarters** in beautiful Redmond, WA. From August 8 – 12, developers, software architects, engineers, designers and more will convene for five days of unbiased and cutting-edge education on the Microsoft Platform. Plus, you'll be at the heart of it all: eating lunch with the Blue Badges, rubbing elbows with Microsoft insiders, exploring the campus, all while expanding your development skills and the ability to create better apps!

EVENT PARTNER



PLATINUM SPONSOR



SUPPORTED BY



PRODUCED BY





# MICROSOFT HQ • AUGUST 8-12, 2016

MICROSOFT HEADQUARTERS • REDMOND, WA



TURN THE  
PAGE FOR  
MORE EVENT  
DETAILS.

## CONNECT WITH VISUAL STUDIO LIVE!



[twitter.com/vslive](https://twitter.com/vslive) – @VSLive



[facebook.com](https://facebook.com) – Search "VSLive"



[linkedin.com](https://linkedin.com) – Join the  
"Visual Studio Live" group!

## TOPICS INCLUDE:

- ALM / DevOps
- Cloud Computing
- Database & Analytics
- Mobile Client
- Software Practices
- Visual Studio / .NET Framework
- Web Client
- Web Server
- Windows Client
- Microsoft-led Sessions

## Have your code heard: register to join us today.

**REGISTER BY  
JUNE 8 AND  
SAVE \$400!**



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

USE PROMO CODE VSLRED4

**VSLIVE.COM/REDMOND**



**Visual Studio Live!** has partnered with the Hyatt Regency Bellevue for conference attendees at a special reduced rate.



Explore the Microsoft Campus during **Visual Studio Live!** Redmond 2016!



**REGISTER BY JUNE 8 AND SAVE \$400!**



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

USE PROMO CODE **VSLRED4**

**VSLIVE.COM/REDMOND**

ALM / DevOps		Cloud Computing	Database & Analytics	Mobile Client
START TIME	END TIME			
8:00 AM	12:00 PM	M01 Workshop: DevOps in a Day - Brian Randell		
12:00 PM	2:00 PM			
2:00 PM	5:30 PM	M01 Workshop Continues		
7:00 PM	9:00 PM			
START TIME	END TIME			
8:30 AM	9:30 AM			
9:45 AM	11:00 AM	T01 Windows Presentation Foundation (WPF) 4.6 - Laurent Bugnion	T02 Angular 2 101 - Deborah Kurata	
11:15 AM	12:30 PM	T06 Windows 10—The Universal Application: One App To Rule Them All? - Laurent Bugnion	T07 TypeScript for C# Developers - Chris Klug	
12:30 PM	2:00 PM			
2:00 PM	3:15 PM	T11 Strike Up a Conversation with Cortana on Windows 10 - Walt Ritscher	T12 Angular 2 Forms and Validation - Deborah Kurata	
3:15 PM	3:45 PM			
3:45 PM	5:00 PM	T16 Building Truly Universal Applications - Laurent Bugnion	T17 Debugging the Web with Fiddler - Ido Flatow	
5:00 PM	6:30 PM			
START TIME	END TIME			
8:00 AM	9:15 AM	W01 Real World Scrum with Team Foundation Server 2015 & Visual Studio Team Services - Benjamin Day	W02 Richer MVC Sites with Knockout JS - Miguel Castro	
9:30 AM	10:45 AM	W06 Team Foundation Server 2015: Must-Have Tools and Widgets - Richard Hundhausen	W07 Get Good at DevOps: Feature Flag Deployments with ASP.NET, WebAPI, & JavaScript - Benjamin Day	
11:00 AM	12:00 PM			
12:00 PM	1:30 PM			
1:30 PM	2:45 PM	W11 Stop the Waste and Get Out of (Technical) Debt - Richard Hundhausen	W12 Getting Started with Aurelia - Brian Noyes	
2:45 PM	3:15 PM			
3:15 PM	4:30 PM	W16 Real World VSTS Usage for the Enterprise - Jim Szubryt	W17 AngularJS & ASP.NET MVC Playing Nice - Miguel Castro	
6:45 PM	9:00 PM			
START TIME	END TIME			
8:00 AM	9:15 AM	TH01 Enterprise Reporting from VSTS - Jim Szubryt	TH02 Build Real-Time Websites and Apps with SignalR - Rachel Appel	
9:30 AM	10:45 AM	TH06 App Services Overview—Web, Mobile, API and Logic Oh My... - Mike Benkovich	TH07 Breaking Down Walls with Modern Identity - Eric D. Boyd	
11:00 AM	12:15 PM	TH11 Real-World Azure DevOps - Brian Randell	TH12 What You Need To Know About ASP.NET Core and MVC 6 - Rachel Appel	
12:15 PM	2:15 PM			
2:15 PM	3:30 PM	TH16 Cloud Oriented Programming - Vishwas Lele	TH17 Take Your Site From Ugh to OOH with Bootstrap - Philip Japikse	
3:45 PM	5:00 PM	TH21 Migrating Cloud Service Applications to Service Fabric - Vishwas Lele	TH22 Test Drive Automated GUI Testing with WebDriver - Rachel Appel	
START TIME	END TIME			
8:00 AM	5:00 PM	F01 Workshop: Data-Centric Single Page Apps with Angular 2, Breeze, and Web API - Brian Noyes		

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!

# REDMOND AGENDA AT-A-GLANCE

★ **DETAILS COMING SOON!** With all of the announcements coming out of Build, we'll be finalizing the Redmond Keynotes and the FULL Track of Microsoft-led sessions shortly. Be sure to check [vslive.com/redmond](http://vslive.com/redmond) for session updates!

Software Practices	Visual Studio / .NET Framework	Web Client	Web Server	Windows Client	Microsoft-led Sessions
Visual Studio Live! Pre-Conference Workshops: Monday, August 8, 2016 <i>(Separate entry fee required)</i>					
M02 Workshop: Developer Dive into SQL Server 2016 - Leonard Lobel			M03 Workshop: Distributed Cross-Platform Application Architecture - Rockford Lhotka & Jason Bock		
Lunch @ The Mixer • Visit the Microsoft Company Store & Visitor Center					
M02 Workshop Continues			M03 Workshop Continues		
Dine-A-Round Dinner					
Visual Studio Live! Day 1: Tuesday, August 9, 2016					
★ KEYNOTE: To Be Announced, <i>Amanda Silver, Director of Program Management, Visual Studio, Microsoft</i>					
T03 Go Mobile with C#, Visual Studio, and Xamarin - James Montemagno		T04 What's New in SQL Server 2016 - Leonard Lobel		★ T05 This Microsoft session is sequestered, details will be released soon	
T08 Using Visual Studio Tools for Apache Cordova to Create Multi-Platform Applications - Kevin Ford		T09 No Schema, No Problem! – Introduction to Azure DocumentDB - Leonard Lobel		★ T10 This Microsoft session is sequestered, details will be released soon	
Lunch • Visit Exhibitors					
T13 Create Great Looking Android Applications Using Material Design - Kevin Ford		T14 Dependencies Demystified - Jason Bock		★ T15 This Microsoft session is sequestered, details will be released soon	
Sponsored Break • Visit Exhibitors					
T18 Building Cross-Platform C# Apps with a Shared UI Using Xamarin.Forms - Nick Landry		T19 Improving Performance in .NET Applications - Jason Bock		★ T20 This Microsoft session is sequestered, details will be released soon	
Microsoft Ask the Experts & Exhibitor Reception • Attend Exhibitor Demos					
Visual Studio Live! Day 2: Wednesday, August 10, 2016					
W03 Windows for Makers: Raspberry Pi, Arduino & IoT - Nick Landry		W04 Applying S.O.L.I.D. Principles in .NET/C# - Chris Klug		W05 ASP.NET Core: Reimagining Web Application Development in .NET - Ido Flatow	
W08 Build Cross-Platform Mobile Apps with Ionic, Angular, and Cordova - Brian Noyes		W09 Open Source Software for Microsoft Developers - Rockford Lhotka		★ W10 This Microsoft session is sequestered, details will be released soon	
★ GENERAL SESSION: To Be Announced					
Birds-of-a-Feather Lunch • Visit Exhibitors					
W13 Top 10 Entity Framework Features Every Developer Should Know - Philip Japikse		W14 Creating Dynamic Pages Using MVVM and Knockout.JS - Christopher Harrison		★ W15 This Microsoft session is sequestered, details will be released soon	
Sponsored Break • Exhibitor Raffle @ 2:55 pm (Must be present to win)					
W18 Predicting the Future Using Azure Machine Learning - Eric D. Boyd		★ W19 This session is sequestered, details will be released soon		★ W20 This Microsoft session is sequestered, details will be released soon	
Set Sail! VSLive's Seattle Sunset Cruise on Lake Washington					
Visual Studio Live! Day 3: Thursday, August 11, 2016					
TH03 Write Better C# - James Montemagno		TH04 Pretty, Yet Powerful. How Data Visualization Transforms the Way we Comprehend Information - Walt Ritscher		TH05 Docker and Azure - Steve Lasker	
TH08 Debugging Your Way through .NET with Visual Studio 2015 - Ido Flatow		TH09 Windows 10 Design Guideline Essentials - Billy Hollis		★ TH10 This Microsoft session is sequestered, details will be released soon	
TH13 Build Distributed Business Apps Using CSLA .NET - Rockford Lhotka		TH14 Learning to Live Without Data Grids in Windows 10 - Billy Hollis		★ TH15 This Microsoft session is sequestered, details will be released soon	
Lunch @ The Mixer • Visit the Microsoft Company Store & Visitor Center					
TH18 PowerShell for Developers - Brian Randell		TH19 Busy JavaScript Developer'sGuide to ECMAScript 6 - Ted Neward		★ TH20 This Microsoft session is sequestered, details will be released soon	
TH23 Tour d'Azure 2016 Edition...New Features and Capabilities that Dev's Need to Know - Mike Benkovich		TH24 Busy Developers Guide to Node.js - Ted Neward		★ TH25 This Microsoft session is sequestered, details will be released soon	
Visual Studio Live! Post-Conference Workshops: Friday, August 12, 2016 <i>(Separate entry fee required)</i>					
	F02 Workshop: Building Business Apps on the Universal Windows Platform - Billy Hollis				



**VSLIVE.COM/REDMOND**





## Playing with Audio in the UWP

The Universal Windows Platform (UWP) has rich APIs for recording audio and video. However, the feature set doesn't stop at recording. With just a few lines of code, developers can apply special effects to audio in real time. Effects such as reverb and echo are built into the API and are quite easy to implement. In this article, I'll explore some of the basics of audio recording and applying special effects. I'll create a UWP app that can record audio, save it, and apply various filters and special effects.

### Setting up the Project to Record Audio

Recording audio requires the app to have permission to access the microphone and that requires modifying the app's manifest file. In Solution Explorer, double-click on the Package.appxmanifest file. It'll always be the in-root of the project.

Once the app manifest file editor window opens, click on the Capabilities tab. In the Capabilities list box, check the Microphone capability. This will allow your app access to the end user's microphone. Without this, your app will throw an exception when you try to access the microphone.

### Recording Audio

Before you start adding special effects to audio, you first want to be able to record audio. This is fairly straightforward. First, add a class to your project to encapsulate all the audio recording code. You'll call this class `AudioRecorder`. It'll have public methods to start and stop recording, as well as to play the audio clip you just recorded. To do this, you'll need to add some members to your class. The first of these will be `MediaCapture`, which provides capabilities for capturing audio, video and images from a capture device, such as a microphone or webcam:

```
private MediaCapture _mediaCapture;
```

You'll also want to add an `InMemoryRandomAccessStream` to capture the input from the microphone into memory:

```
private InMemoryRandomAccessStream _memoryBuffer;
```

In order to keep track of the state of your recording, you'll add a publicly accessible property `Boolean` to your class:

```
public bool IsRecording { get; set; }
```

Recording the audio requires you to check if you're already recording and if you are, the code will throw an exception. Otherwise, you'll need to initialize your memory stream, delete the previous recording file and start recording.

Because the `MediaCapture` class provides multiple functions, you'll have to specify that you want to capture audio. You'll create an instance of `MediaCaptureInitializationSettings` to do just that. The code then creates an instance of a `MediaCapture` object and passes the `MediaCaptureInitializationSettings` to the `InitializeAsync` method, as shown in **Figure 1**. Finally, you'll tell the `MediaCapture` object to start recording, passing along parameters that it records in MP3 format and where to store the data.

Stopping the recording requires far fewer lines of code:

```
public async void StopRecording()
{
    await _mediaCapture.StopRecordAsync();
    IsRecording = false;
    SaveAudioToFile();
}
```

The `StopRecording` method does three things: it tells the `MediaCapture` object to stop recording, sets the recording state to false and saves the audio stream data to an MP3 file on disk.

### Saving Audio Data to Disk

Once the captured audio data is in the `InMemoryRandomAccessStream`, you want to save the contents onto disk, as shown in **Figure 2**. Saving audio data from an in-memory stream requires you to copy the contents over to another stream and then push that data onto disk. Using the utilities in the `Windows.ApplicationModel.Package` namespace, you're able to get the path to your app's install directory. (During development, this will be in the `\bin\x86\Debug` directory of the project.) This is where you want the file to

Figure 1 Creating an Instance of a `MediaCapture` Object

```
public async void Record()
{
    if (IsRecording)
    {
        throw new InvalidOperationException("Recording already in progress!");
    }

    await Initialize();
    await DeleteExistingFile();

    MediaCaptureInitializationSettings settings =
        new MediaCaptureInitializationSettings
    {
        StreamingCaptureMode = StreamingCaptureMode.Audio
    };

    _mediaCapture = new MediaCapture();
    await _mediaCapture.InitializeAsync(settings);
    await _mediaCapture.StartRecordToStreamAsync(
        MediaEncodingProfile.CreateMp3(AudioEncodingQuality.Auto), _memoryBuffer);
    IsRecording = true;
}
```

Code download available at [bit.ly/1ObYYIb](http://bit.ly/1ObYYIb).



# Spreadsheets Made Easy.



## Fastest Calculations

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



## 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.



Windows  
Forms



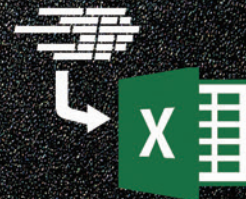
Silverlight



WPF

## Powerful Controls

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



## 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](http://SpreadsheetGear.com)



# SpreadsheetGear

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



Figure 2 Saving Audio Data to Disk

```
private async void SaveAudioToFile()
{
    IRandomAccessStream audioStream = _memoryBuffer.CloneStream();
    StorageFolder storageFolder = Package.Current.InstalledLocation;

    StorageFile storageFile = await storageFolder.CreateFileAsync(
        DEFAULT_AUDIO_FILENAME, CreationCollisionOption.GenerateUniqueName);
    this._fileName = storageFile.Name;

    using (IRandomAccessStream fileStream =
        await storageFile.OpenAsync(FileAccessMode.ReadWrite))
    {
        await RandomAccessStream.CopyAndCloseAsync(
            audioStream.GetInputStreamAt(0), fileStream.GetOutputStreamAt(0));
        await audioStream.FlushAsync();
        audioStream.Dispose();
    }
}
```

be recorded. You could easily modify the code to save elsewhere or have the user pick where to save the file.

## Playing Audio

Now that you have your audio data inside an in-memory buffer and on disk, you have two choices to play from: memory and disk.

The code for playing the audio from memory is quite simple. You create a new instance of the `MediaElement` control, set its source to the in-memory buffer, pass it a MIME type and then call the `Play` method.

```
public void Play()
{
    MediaElement playbackMediaElement = new MediaElement();
    playbackMediaElement.SetSource(_memoryBuffer, "MP3");
    playbackMediaElement.Play();
}
```

Playing from disk requires a little extra code, as opening files is an asynchronous task. In order to have the UI thread communicate with a task running on another thread, you'll need to use the `CoreDispatcher`. The `CoreDispatcher` sends messages between the thread a given piece of code is running on and the UI thread. With it, code can get the UI context from another thread. For an excellent description of `CoreDispatcher`, read David Crook's blog post on the subject at [bit.ly/1SbJ6up](http://bit.ly/1SbJ6up).

Aside from the extra steps to handle the asynchronous code, the method resembles the previous one that uses the in-memory buffer:

```
public async Task PlayFromDisk(CoreDispatcher dispatcher)
{
    await dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
    {
        MediaElement playbackMediaElement = new MediaElement();
        StorageFolder storageFolder = Package.Current.InstalledLocation;
        StorageFile storageFile = await storageFolder.GetFileAsync(this._fileName);
        IRandomAccessStream stream = await storageFile.OpenAsync(FileAccessMode.Read);

        playbackMediaElement.SetSource(stream, storageFile.FileType);
        playbackMediaElement.Play();
    });
}
```

## Building the UI

With the `AudioRecorder` class complete, the only thing left to do is build out the interface for the app. The interface for this project is quite simple, as all you need is a button to

record and a button to play back the recorded audio, as shown in **Figure 3**. Accordingly, the XAML is simple: a `TextBlock` and a stack panel with two buttons:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="43"/>
        <RowDefinition Height="**"/>
    </Grid.RowDefinitions>
    <TextBlock FontSize="24">Audio in UWP</TextBlock>
    <StackPanel HorizontalAlignment="Center" Grid.Row="1" >
        <Button Name="btnRecord" Click="btnRecord_Click">Record</Button>
        <Button Name="btnPlay" Click="btnPlay_Click">Play</Button>
    </StackPanel>
</Grid>
```

In the codebehind class, you create a member variable of `AudioRecorder`. This will be the object your app uses to record and play back audio:

```
AudioRecorder _audioRecorder;
```

You'll instantiate the `AudioRecorder` class in the constructor of your app's `MainPage`:

```
public MainPage()
{
    this.InitializeComponent();
    this._audioRecorder = new AudioRecorder();
}
```

The `btnRecord` button actually toggles the starting and stopping of audio recording. In order to keep the user informed of the current state of the `AudioRecorder`, the `btnRecord_Click` method changes the content of the `btnRecord` button, as well as starts and stops recording.

You have two options for the event handler for the `btnPlay` button: to play from the in-memory buffer or play from a file stored on disk.

To play from the in-memory buffer, the code is straightforward:

```
private void btnPlay_Click(object sender, RoutedEventArgs e)
{
    this._audioRecorder.Play();
}
```

As I mentioned previously, playing the file from disk happens asynchronously. This means that the task will run on a different thread than the UI thread. The OS scheduler will determine what thread the task will execute on at run time. Passing the `Dispatcher` object to the `PlayFromDisk` method allows the thread to get access to the UI context of the UI thread:

```
private async void btnPlay_Click(object sender, RoutedEventArgs e)
{
    await this._audioRecorder.PlayFromDisk(Dispatcher);
}
```

## Applying Special Effects

Now that you have your app recording and playing back audio, the time has come to explore some of the lesser-known features in the UWP: real-time audio special effects. Included within the APIs in the `Windows.Media.Audio` namespace are a number of special effects that can add an additional touch to apps.

For this project, you'll place all the special effects code into its own class. However, before you create the new class, you'll make one last modification to the `AudioRecorder` class. I'll add the following method:

```
public async Task<StorageFile>
    GetStorageFile(CoreDispatcher dispatcher)
{
    StorageFolder storageFolder =
        Package.Current.InstalledLocation;
    StorageFile storageFile =
        await storageFolder.GetFileAsync(this._fileName);
    return storageFile;
}
```

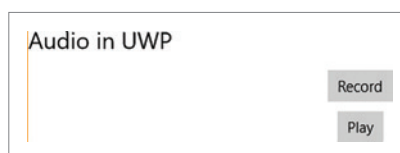


Figure 3 AudioRecorder UI



The `GetStorageFile` method returns a `StorageFile` object to the saved audio file. This is how my special effects class will access the audio data.

## Introducing the AudioGraph

The `AudioGraph` class is central to advanced audio scenarios in the UWP. An `AudioGraph` can route audio data from input source nodes to output source nodes through various mixing nodes. The full extent and power `AudioGraph` lies beyond the scope of this article, but it's something I plan to dive more deeply into in future articles. For now, the important point is that every node in an audio graph can have multiple audio effects applied to them. For more information on `AudioGraph`, be sure to read the article on the Windows Dev Center at [bit.ly/1VCIBfD](http://bit.ly/1VCIBfD).

First, you'll want to add a class called `AudioEffects` to your project and add the following members:

```
private AudioGraph _audioGraph;
private AudioFileInputNode _fileInputNode;
private AudioDeviceOutputNode _deviceOutputNode;
```

In order to create an instance of the `AudioGraph` class, you need to create an `AudioGraphSettings` object, which contains the configuration settings for the `AudioGraph`. You then call the `AudioGraph.CreateAsync` method passing these configuration settings. The `CreateAsync` method returns a `CreateAudioGraphResult` object. This class provides access to the created audio graph and a status value whether the audio graph creation failed or succeeded.

You also need to create an output node to play the audio. To do so, call the `CreateDeviceOutputNodeAsync` method on the `AudioGraph` class and set the member variable to the `DeviceOutputNode` property of the `CreateAudioDeviceOutputNodeResult`. The code to initialize the `AudioGraph` and the `AudioDeviceOutputNode` all resides in the `InitializeAudioGraph` method here:

```
public async Task InitializeAudioGraph()
{
    AudioGraphSettings settings = new AudioGraphSettings(AudioRenderCategory.Media);
    CreateAudioGraphResult result = await AudioGraph.CreateAsync(settings);
    this._audioGraph = result.Graph;
    CreateAudioDeviceOutputNodeResult outputDeviceNodeResult =
        await this._audioGraph.CreateDeviceOutputNodeAsync();
    _deviceOutputNode = outputDeviceNodeResult.DeviceOutputNode;
}
```

Playing audio from an `AudioGraph` object is easy; simply call the `Play` method. Because the `AudioGraph` is a private member of your `AudioEffects` class, you'll need to wrap a public method around it to make it accessible:

```
public void Play()
{
    this._audioGraph.Start();
}
```

Now that you have the output device node created on the `AudioGraph`, you need to create an input node from the audio file stored on disk. You'll also need to add an outgoing connection to the `FileInputNode`. In this case, you want the outgoing node to be your audio output device. That's exactly what you do in the `LoadFileIntoGraph` method:

```
public async Task LoadFileIntoGraph(StorageFile audioFile)
{
    CreateAudioFileInputNodeResult audioFileInputResult =
        await this._audioGraph.CreateFileInputNodeAsync(audioFile);

    _fileInputNode = audioFileInputResult.FileInputNode;
    _fileInputNode.AddOutgoingConnection(_deviceOutputNode);

    CreateAndAddEchoEffect();
}
```

You'll also notice a reference to the `CreateAndAddEchoEffect` method, which I'll discuss next.

## Adding the Audio Effect

There are four built-in audio effects in the audio graph API: echo, reverb, equalizer and limiter. In this case, you want to add an echo to the recorded sound. Adding this effect is as easy as creating an `EchoEffectDefinition` object and setting the properties of the effect. Once created, you need to add the effect definition to a node. In this case, you want to add the effect to the `_fileInputNode`, which contains the audio data recorded and saved onto disk:

```
private void CreateAndAddEchoEffect()
{
    EchoEffectDefinition echoEffectDefinition = new
    EchoEffectDefinition(this._audioGraph);

    echoEffectDefinition.Delay = 100.0f;
    echoEffectDefinition.WetDryMix = 0.7f;
    echoEffectDefinition.Feedback = 0.5f;

    _fileInputNode.EffectDefinitions.Add(echoEffectDefinition);
}
```

## Playing audio from an AudioGraph object is easy.

## Putting It All Together

Now that you have the `AudioEffect` class completed, you can use it from the UI. First, you'll add a button to your app's main page:

```
<Button Content="Play with Special Effect" Click="btnSpecialEffectPlay_Click" />
```

And inside the click event handler, you get the file where the audio data is stored, create an instance of the `AudioEffects` class and pass it to the audio data file. Once that's all done, all you need to do to play the sound is call the `Play` method:

```
private async void btnSpecialEffectPlay_Click(object sender, RoutedEventArgs e)
{
    var storageFile = await this._audioRecorder.GetStorageFile(Dispatcher);
    AudioEffects effects = new AudioEffects();
    await effects.InitializeAudioGraph();
    await effects.LoadFileIntoGraph(storageFile);
    effects.Play();
}
```

You run the app and click **Record** to record a small clip. To hear it as it was recorded, click the **Play** button. To hear the same audio with an echo added to it, click **Play with Special Effect**.

## Wrapping Up

The UWP not only has rich support for capturing audio, but it also has some superb features to apply special effects to media in real time. Included with the platform are several effects that can be applied to audio. Among these are echo, reverb, equalizer and limiter. These effects can be applied individually or in any number of combinations. The only limit is your imagination. ■

---

**FRANK LA VIGNE** is a technology evangelist on the Microsoft Technology and Civic Engagement team, where he helps users leverage technology in order to create a better community. He blogs regularly at [FranksWorld.com](http://FranksWorld.com) and has a YouTube channel called *Frank's World TV* ([youtube.com/FranksWorldTV](http://youtube.com/FranksWorldTV)).

---

**THANKS** to the following technical experts for reviewing this article: *Drew Batchelor* and *Jose Luis Manners*



# Orlando 2016

ROYAL PACIFIC RESORT AT UNIVERSAL  
DECEMBER 5-9



## *The Ultimate Education Destination*

**Live! 360<sup>SM</sup> is a unique conference** where the IT and Developer community converge to debate leading edge technologies and educate themselves on current ones. These six co-located events incorporate knowledge transfer and networking, along with finely tuned education and training, as you create your own custom conference, mixing and matching sessions and workshops to best suit your needs.

**Choose the ultimate education destination: Live! 360.**

EVENT PARTNERS



PLATINUM SPONSOR



GOLD SPONSOR



SUPPORTED BY







**6** Great Conferences  
**1** Great Price



## Connect with Live! 360



twitter.com/live360  
@live360



facebook.com  
Search "Live 360"



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

**REGISTER TODAY  
AND SAVE \$500!**



Use promo code  
L360MAY2

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

# Take the Tour

**Visual Studio LIVE!**  
EXPERT SOLUTIONS FOR .NET DEVELOPERS

**Visual Studio Live!:** VSLive!™ is a victory for code, featuring unbiased and practical development training on the Microsoft Platform.

**SQL Server LIVE!**  
TRAINING FOR DBAs AND IT PROS

**SQL Server Live!:** This conference will leave you with the skills needed to Lead the Data Race, whether you are a DBA, developer, IT Pro, or Analyst.

**TECHMENTOR**  
IN-DEPTH TRAINING FOR IT PROS

**TechMentor<sup>SM</sup>:** This is IT training that finishes first, with zero marketing speak on topics you need training on now, and solid coverage on what's around the corner.

**Office & SharePoint LIVE!**  
ON-PREMISE, CLOUD & CROSS-PLATFORM TRAINING

**Office & SharePoint Live!:** Provides leading-edge knowledge and training for SharePoint both on-premises and in Office 365 to maximize the business value.

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

**Modern Apps Live!:** Presented in partnership with Magenic, this unique conference leads the way to learning how to architect, design and build a complete Modern App.

**APPDEV TRENDS**  
ENTERPRISE FOCUSED. CODE DRIVEN.

**NEW!**

**App Dev Trends:** This new technology conference focuses on the makers & maintainers of the software that Power organizations in nearly every industry in the world – in other words, enterprise software professionals!





# The Joy of UX

Ten years ago, I published my book, “Why Software Sucks,” highlighting the UX shortcomings of current programs. I thought it would save the world, but it didn’t. When I asked devs why they didn’t fix the problems I described, they said, “We don’t know how, and we’re too busy.”

I implored Microsoft to fix the former problem, but it wouldn’t. As a product-oriented company, Microsoft explains in great detail *how* to implement a feature, but almost never when to use it and when not to. And in the few spots where Microsoft does provide such guidance, the company undermines it by doing exactly the opposite in its own applications (see my May 2013 column at [msdn.com/magazine/dn198249](http://msdn.com/magazine/dn198249)).

Enough! I can’t stand it any longer. I’ve written another book to fix both of these problems together and save the world. Addison-Wesley published it last month. I call it “The Joy of UX.” Any similarity to “The Joy of Sex” is one heck of a coincidence, isn’t it?

Written for an audience  
who knew nothing about  
cooking, that book famously  
opened with the instructions,  
“Stand facing the stove.”

I also wanted to emulate the original “Joy of Cooking.” Written for an audience who knew nothing about cooking, that book famously opened with the instructions, “Stand facing the stove.” As one reviewer writes, it targeted readers who were, “... busy, not really interested in cooking, but eager to bring off a dashing effect with a minimum of effort.”

That describes you, dear reader, doesn’t it? You know you’d make more money with a better UX. But you don’t have resources for a full UX department, and if your company already has one, you can’t get much attention from them. Raising your UX game is up to you, the front-line geek. You wish you could read Alan Cooper’s “About Face” cover to cover, but you don’t have time, and its level of detail makes it hard to apply.

Fear not. Plattski’s got you covered. My “Joy” is short, only 212 pages. I’ve extracted the 20 percent of principles that you use 80 percent of the time. I’ve compiled them into seven easy steps, the

totality of which I have named (with characteristic modesty) the Plattski UX Protocol. Figure out who the user is. Figure out what problem the user is trying to solve. Make low-fidelity mockups in a sketch editor, then try them on actual users or their representatives. Modify mockups based on the results. (And a few other easy steps.) Rinse. Repeat. Profit.

The last two chapters contain case studies applying these principles to two real-life programs—the MBTA commuter rail mobile app, and Beth Israel Hospital’s PatientSite.org Web site. Early readers tell me that this is their favorite part of the book. You’ll find a sample chapter and more case studies at [joyofux.com](http://joyofux.com).

My other books have targeted specific technologies, and waned as those technologies aged. This one is different. The same principles, and therefore the same development steps, apply to all applications, regardless of platform. It doesn’t matter whether you’re on Xamarin or HTML5 or Windows Forms or Web Forms or even VB6. You don’t have to upgrade to a fancy graphical environment to make your users much happier. Just start listening to them, as I show you.

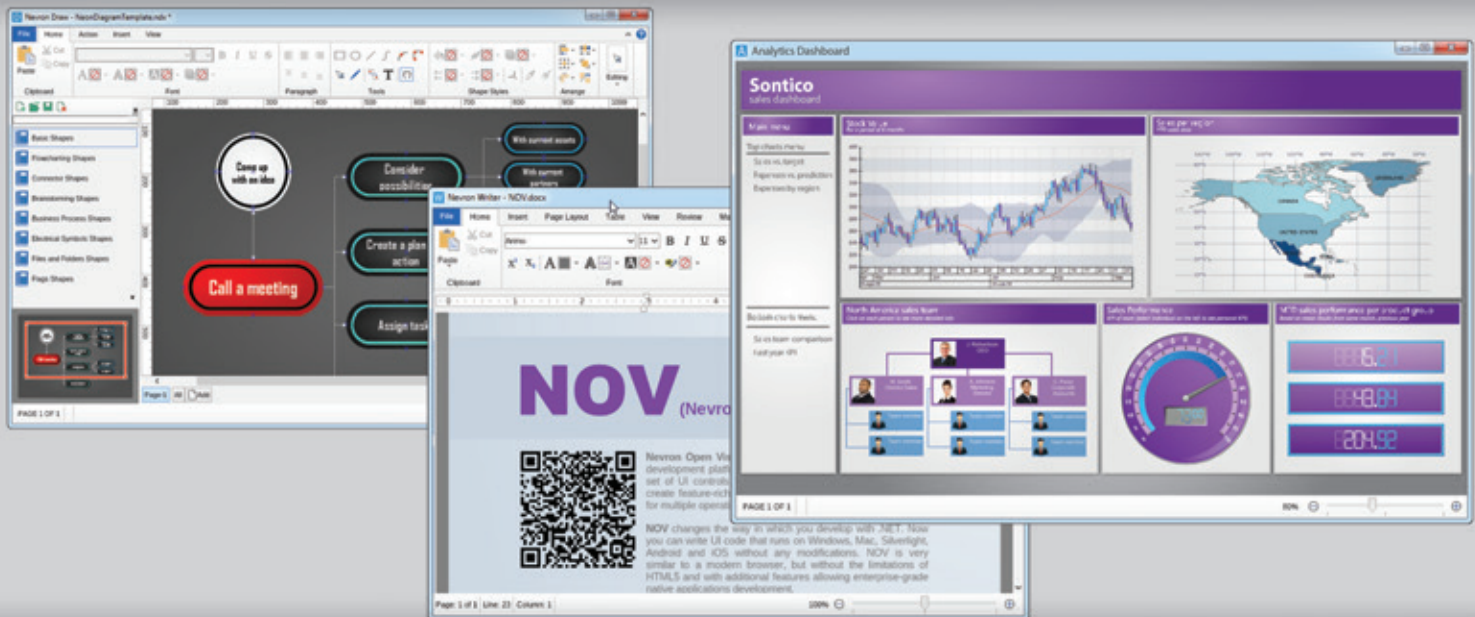
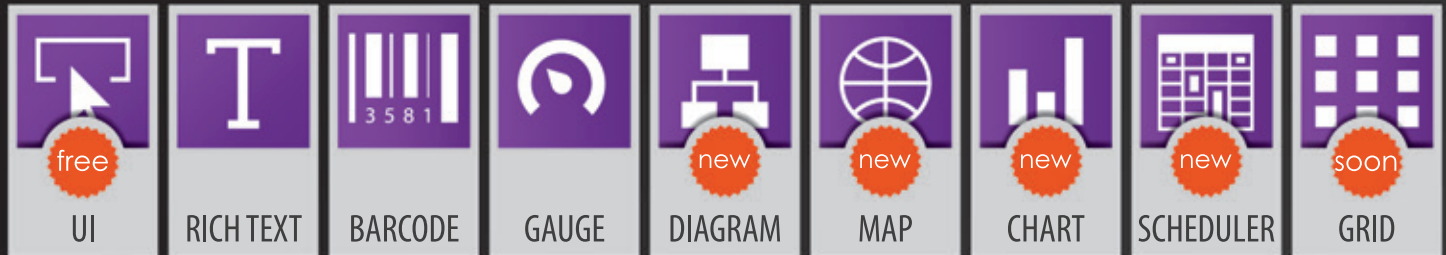
Above all, start these steps early in the development process. Teams way too often say, “We’ll work on the UX once we have the architecture fleshed out.” The UX *is* your architecture, as I wrote in my September 2011 column ([msdn.com/magazine/hh394140](http://msdn.com/magazine/hh394140)). Until you’ve involved your users, you don’t know what they need or want. An app’s basic feature set always changes, often drastically, after users react to the first mockups.

If you really want to get moving, I’ll bring this guidance to you personally through my jumpstart workshops, in which I come to your company and guide you through these steps on your own projects. Because summer is my slow season, I’m offering a half-price deal through Sept. 1. Check it out on the book’s Web site and give me a call.

The “Joy of Cooking” has remained in print continuously since 1936 and sold more than 18 million copies through eight editions. If I keep “The Joy of UX” up-to-date, it might outlast me. Perhaps my daughters will take it over, as Marion Rombauer Becker took over the original “Joy of Cooking” from her mother Irma Rombauer. Stand facing the users, my friends. ■

**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](http://rollthunder.com).

# The Complete UI suite for .NET



Nevron Open Vision is the first and only .NET component suite that allows you to develop cutting edge Applications, Dashboards and User Interfaces for Windows (WinForms and WPF) and Mac (MonoMac and Xamarin.Mac) from a Single Code Base.

It includes advanced UI controls such as Chart, Diagram, Grid, Text Editor, Scheduler, Gauges and Barcodes as well a complete set of UI controls (ListBox, TreeView, ComboBox, Color Pickers etc.).

The controls in those suites seamlessly integrate in






All controls in the suite are completely windowless, styleable and open for customization.

Learn more at [www.nevron.com](http://www.nevron.com) today

[www.nevron.com](http://www.nevron.com) | [email@nevron.com](mailto:email@nevron.com) | +1 888-201-6088 (Toll free, USA and Canada)

Microsoft, .NET, ASP.NET, SharePoint, SQL Server and Visual Studio are registered trademarks of Microsoft Corporation in the United States and/or other countries. Some Nevron components are only available for certain platforms. For details visit [www.nevron.com](http://www.nevron.com) or send an e-mail to [support@nevron.com](mailto:support@nevron.com).

SYNCFUSION

# DASHBOARD PLATFORM

BUSINESS DASHBOARDS SIMPLIFIED



- ★ The complete solution for creating and sharing interactive business dashboards.
- ★ Includes a user-friendly drag-and-drop designer and widgets to visualize your dashboards.
- ★ Integrates seamlessly with the Syncfusion Big Data Platform.
- ★ Connects to commonly used data sources like Microsoft Excel, SQL Server, Oracle, MySQL and Spark SQL.

Deploy to 50 users for only **\$1,995** per year

FREE COMMUNITY LICENSE AVAILABLE!

Ready to get started?

**Download a free trial**

[www.syncfusion.com/msdndashboard](http://www.syncfusion.com/msdndashboard)

