SignalR, Entity Framework, Python, ASP.NET Core, Visual Basic

# CODE

JUL
AUG
2018

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 5.95 Can $ 8.95


## OPEN SOURCE

.NET CORE 2.1 AWAKENS

WHAT'S NEW IN ASP.NET CORE 2.1 .NET CORE 2.1 FLAGSHIP TYPES ASP.NET CORE SIGNALR FOR REAL-TIME APPLICATIONS ENTITY FRAMEWORK CORE 2.1 IS PRODUCTION READY JAVASCRIPT CORNER TRY/CATCH MICROSOFT COGNITIVE SERVICES IDENTIFY VOICES LEARN PYTHON WITH VISUAL STUDIO PREPARE VISUAL BASIC FOR CONVERSION TO C#

**WWW.DOT.NET**

08
0 74470 56130 9

# Features



# Columns



# Departments




US subscriptions are US $29.99 for one year. Subscriptions outside the US pay US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com.

Subscribe online at codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A.

Canadian Subscriptions: Canada Post Agreement Number 7178957. Send change address information and blocks of undeliverable copies to IBC, 7485 Bath Road, Mississauga, ON L4T 4C1, Canada.

LEADTOOLS

# One SDK, Multiple Platforms

LEADTOOLS toolkits will help you create desktop, web, server, and mobile applications with the greatest collection of programmer-friendly and cross-platform document, medical, and multimedia technologies.

OCR | FORMS | BARCODE | DICOM | PACS

+ MANY MORE

.NET | Xamarin | Windows | Java | Linux | iOS | macOS | Android | HTML5

**Get Started Today**

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

# Act 1: Into the Great Unknown

In my last editorial, "A Software Pre-Mortem," I discussed a new project my team was just starting. It's now been a bit over two months and the project is beginning to pick up real steam. The team picked up an additional developer and is about to add ten more people. Yes, you read that right; we're going from a six-person development team to a 16-person team.

Our goal is to take the lessons we learned in the early phases of this project and apply them to a much larger development team. In this editorial, I want to discuss a few things we've learned and how we intend to apply these lessons to a larger team.

> In 1975, Fred Brooks documented the challenges of adding numerous developers to a project too quickly in "The Mythical Man-Month."

## Learning What We Didn't Know

When we started this project, the development team chose several forms and processes from the legacy system that represented functionality and would be spread throughout the project. Some of these were "simple" data entry screens. Others were complex business processes and others were parts of more complex features. The purpose of this phase of development was to ferret out a general idea of what was missing from the framework. We found numerous gaps in the framework and in our process.

### *Conversions Need Analysis and Design*

When converting projects, it's very tempting to just take a feature and start to convert it right away. We learned very quickly that this wouldn't work well when going from one code base to another. The legacy system (20 years old) had incurred a lot of technical debt due to its design. Throughout the legacy codebase, code was tucked away in every "nook and cranny." This means that code could be found everywhere from abstract classes, to individual event hooks, to external modules and third-party tools. When building a feature, it's difficult to ascertain where the code you're converting really lives. Adding to this challenge is the fact that the new architecture is a multi-layered application with the MVVM pattern being used for the front end and a service-oriented architecture on the back end.

We adopted a new process before development could start on converting features. We needed to spend time doing a thorough examination of the legacy code base and finesse those features into our current layered architecture. We developed a rough outline of what each developer needed to do. The basic outline is as follows:

1. Analyze functional documentation and screens
2. Describe what the service needs to do and its goals
3. Extract all repository calls (stored procedures)
    a. Create an interface of repo methods
4. Design a service that will achieve goals of the feature
    a. Create the interface
5. Extract all user interactions
    a. Alerts
    b. Yes/No
    c. Other user prompts
6. Review and approval of design by senior developer

Conversions require special attention to the details found in the current code base as well as the documentation. Having developers go through this process makes it more likely that details aren't missed. Another benefit of this process is that managers get a better idea of feature sizing; what might have seemed simple is often more complex or vice versa.

### *Documentation and Training Are Paramount*

I believe that a frequently overlooked aspect of development is documentation and training. I also believe that the quickest way to achieve productivity is to provide developers with good documentation and training. During the initial phase of the project, we had a few sessions with all of the developers to train on the various aspects of each project. We spent time demonstrating new tools, and new framework features and concepts. We discovered that we could deliver huge productivity gains by having these training sessions. Every developer found these sessions rewarding and now they're becoming a regular part of our development process.

## Creating "The Way"

It's funny, the things that trip up developers. Here's one I found to be profound: the deceptively simple concept of where to place a file in the project solution. The question: When I create a new entity (class, view, view model, service, repository, etc.) where should I put it in the solution structure? When building large scale applications, this really matters. We must have a standard way of adding elements. As a matter of fact, we must have a standard way of adding everything to the project structure. As the saying goes, we need "A place for everything and everything in its place." It was during the initial phases that we standardized file locations.

Of course, it's not only file locations that are important to standardize. It's important to standardize how parts of the application will interact. We specified how modules would communicate with each other, and where the user interface elements would go versus the service elements. This became known as "The Way."

### *Measuring Progress*

This project is of a critical nature to our client and our stakeholders have placed a priority on how this project progresses. We've gone through several fits and starts when it comes to tracking progress and just recently came up with an acceptable solution. The beauty is that the tracking solution we chose matches "The Way" I mentioned above.

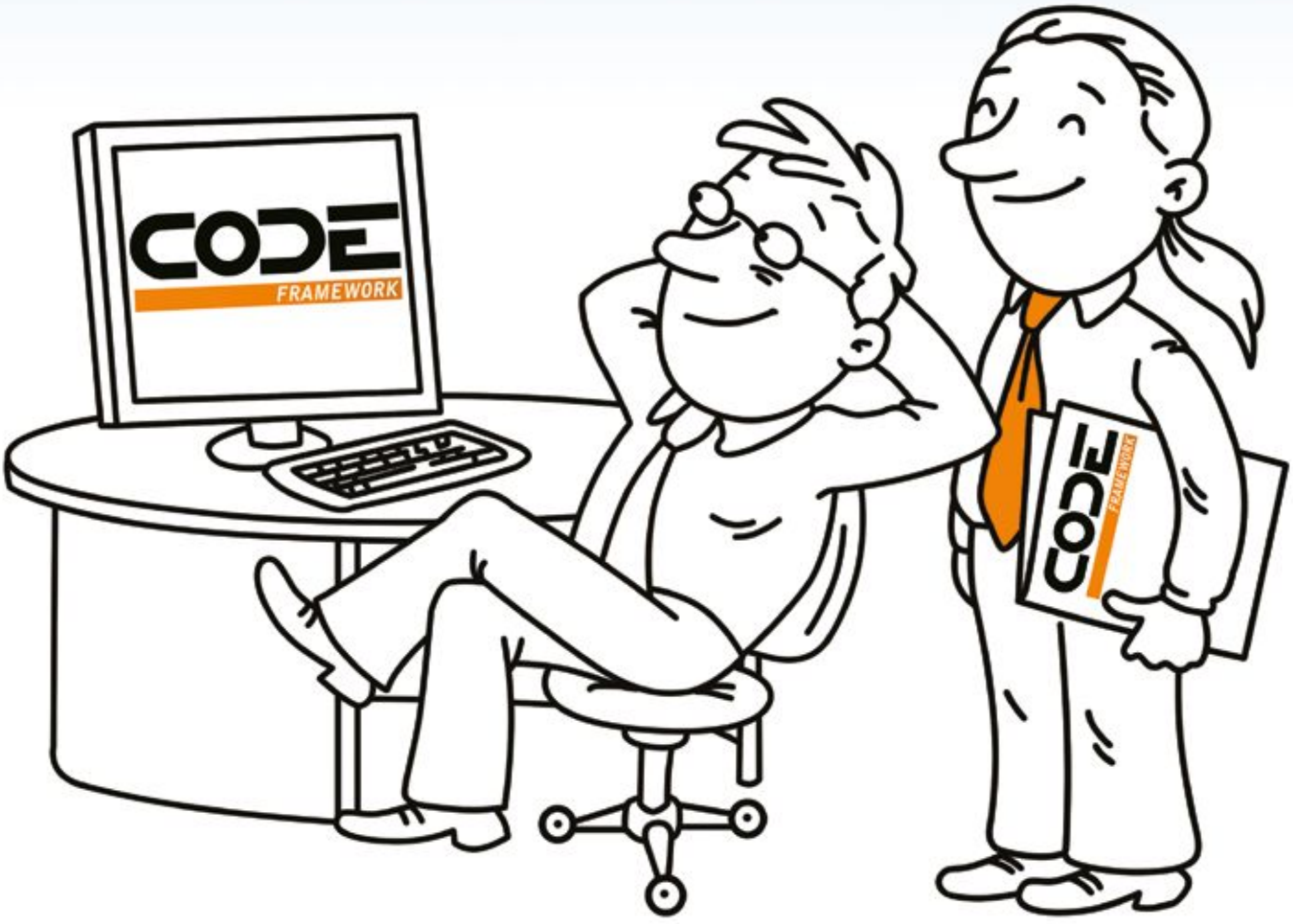
Every software project I work on presents its own unique challenges and this project is no exception. The sheer size of this project required new tools, techniques, and methods. It even required a wholesale new way of tracking progress as previous projects didn't have the scope and scale of this one.

The most important aspect of this project so far has been flexibility. Each day presents new challenges and it's flexibility that rules the day.

Rod Paddock
CODE

This issue has been largely written and reviewed in collaboration with Microsoft and Microsoft employees from the ASP.NET Core team. For more on their work, check out: **www.dot.net**.

# CODE Framework:
# Business Applications Made Easy




Architected by Markus Egger and the experts at CODE Magazine, CODE Framework is the world's most productive, maintainable, and reusable business application development framework for today's developers. CODE Framework supports existing application interoperability or can be used as a foundation for ground-up development. Best of all, it's free, open source, and professionally supported.

Download CODE Framework at www.codemag.com/framework

*Helping Companies Build Better Software Since 1993*

**www.codemag.com/framework**
832-717-4445 ext. 9 • info@codemag.com

CODE
FRAMEWORK

# JavaScript Corner: Try Catch

Every modern programming language has the ability to handle and throw exceptions, and JavaScript is no exception. In this column, I discuss why, how, and when you should make use of JavaScript's try catch statement as well as throwing your own custom errors. A try catch in any programming language is a block in which code can execute in a way where an exception can safely occur in a way that won't result in the application's abnormal termination. With a try catch, you can handle an exception that may include logging, retrying failing code, or gracefully terminating the application. Without a try catch, you run the risk of encountering unhandled exceptions. Try catch statements aren't free in that they come with performance overhead. Like any language feature, try catches can be overused.

**John V. Petersen**
johnvpetersen@gmail.com
about.me/johnvpetersen
@johnvpetersen

Based near Philadelphia, Pennsylvania, John has been an information technology developer, consultant, and author for over 25 years.

**Note:** This column and the other columns in this series are meant to add color to the docs and concepts therein, rather than as a less comprehensive duplication of that content.

## Context

How many times have you used an application and something goes wrong, interrupting your work without warning and without any context for what went wrong? You call support and over the next few hours or perhaps days, there's an attempt by many to work on the problem. If you're lucky, the problem gets resolved quickly so that you can move on with your work. Too often, there are delays in solving the problem because the support and development teams lack the necessary information to know what happened in order to diagnose, solve, and remedy the problem.

The irony is that during development, when the decision of whether to employ **try catch** was discussed (assuming the discussion was had), somebody argued that **try catch** is too expensive from a performance standpoint to warrant its use. Typically, when these conclusions are made, they're made without any evidence to substantiate the conclusion because such a performance argument is quantified as to degree, likelihood, or cost. Ironically, in an endeavor that's all about logic, we end up dealing with many development decisions that are gravely logically flawed. All of a sudden, it becomes crystal clear why there's so much bad software.

> In an endeavor that's all about logic, we often make development decisions that are logically flawed.

## Core Concepts

This column discusses the following JavaScript elements:

- **try:** In a try block, one or more statements in that block are attempted. A block is the space that exists between curly braces ({}). At least one **catch** or **finally** statement must follow a **try**. The technicality that you don't need a **catch** statement notwithstanding, I can't imagine why you wouldn't implement at least one **catch** statement after a **try** because the catch is the one place where you can intercept the thrown exception. As is the case for any code block, a **try** can contain nested **try catch** blocks.
- **catch:** In the that event an exception is encountered in a **try** block, the **catch** statement immediately following the **try** is triggered. A **catch** accepts one parameter, as specified by the previous **throw** statement. Within a **catch** block, you can test for specific errors and conditionally execute code for those errors.
- **finally:** While handling exceptions, you may need cleanup code that executes regardless of whether or not an exception occurs. The code in the **finally** block immediately following the **try** block always executes.
- **throw:** The **throw** statement allows you to invoke a user-defined error. Depending on criteria you choose, you force the calling code to handle the error.
- **Error object:** The Error object is a base object for generic errors and for the many built-in error types in JavaScript. In addition, the Error Object can be extended to create user-defined errors.

### *Be Aware of Non-standard JavaScript*

Although there is the ECMAScript standard for JavaScript, it's important to be aware of your code's context and the non-standard features it may afford you. A good example is the conditional **catch** statement:

```
````````````catch(e if e instanceof MyError) {}
```

This code, which is another way to conditionally trigger a catch statement isn't standard ECMAScript and may not work in the environment where your code executes. This is a feature that Mozilla (FireFox) created. Accordingly, this feature works in FireFox, but it won't work in Chrome or Safari. This is what makes JavaScript challenging at times; browser-specific features that may be present. In addition, different browsers may implement new JavaScript (ECMAScript) standards at different times.

## Application

A basic question to confront is when you should wrap code in a **try catch** block. One way to answer that question is to consider what **try catch** is not. **Try catch's** purpose is not to remediate poor code. A common anti-pattern is to wrap code that intermittently throws an error for some unknown reason in order to apply remedial code in the **catch** block.
```

A common remedial code example includes the application of default values for variables. The problem with this implementation of **try catch** is that it masks what could be a serious underlying problem. Instead, **try catch** should be implemented in cases where errors may occur. Examples include the unavailability of an external service, invalid log-in credentials, invalid user input, etc. As much as possible, you should strive to catch specific errors. As a catch-all, it's a good practice to have an unconditional **catch** block as a fail-safe in the event an error is thrown that you didn't anticipate. An unconditional block should always be the last catch block in your statement. Catching errors allows you to log errors and, if possible, retry code so that work can progress. If such efforts are not successful, catching errors allows errors to be handled gracefully in a way that preserves the user experience. The following examples highlight some approaches you may choose to employ in your applications.

The following examples rely on this simple class:

```
class Result {
    constructor() {
        this.connection = null;
        this.error = null;
    }
}
```

### *Basic Try Catch, Finally, and Throw*

In this next example, there's a stub function that attempts a connection. For demonstration purposes, if a string with a length of at least one isn't supplied, an error is thrown. In all cases, an instance of the Result class is returned. The **catch** block in this example is an unconditional catch in that it executes for any error. Regardless of whether an error occurs, the **finally** block executes.

```
function tryConnection(connString) {
  const result = new Result();
   try {
     if (connString == undefined ||
       connString == null ||
       connString.length == 0)
       throw new Error("You must supply a valid
          connection string.");
      result.connection = connString;
  } catch (e) {
      result.error = e;
  }
    finally {
      return result;
    }
}
```

### *Custom Error Class*

The following example illustrates a simple custom error class. In the constructor, a call is made to the parent class' constructor.

```
class ConnectionError extends Error {
       constructor(connString) {
   super("There was a connection problem with
          the connection string: " + connString);
   }
}
```

With a custom error class in place, you can now implement it. The following is a modified version of the first

**try catch** example. The modified code is in red. Instead of throwing a generic error, the new custom error is thrown. In the **catch** block, there's specific code that tests for the type of error thrown. For the custom error, the connection string is the only passed argument.

```
function tryConnection(connString) {
    const result = new Result();
    try {
        if (connString == undefined ||
            connString == null ||
            connString.length == 0)
            throw new ConnectionError
              (connString);
        result.connection = connString;
    }
    catch (e) {
        if (e instanceof ConnectionError) {
            console.trace();
            console.log(e);
            // Alternatively, the console trace
            // and log  code could hosted in the
            // custom error class.
        }
        result.error = e;
    }
    finally {
       return result;
   }
}
```

### *Take Note of JavaScript's Built-in Error Definitions*

JavaScript has a number of built-in error definitions. Examples include **RangeError: invalid date** and **JSON.parse: bad parsing**. Whenever possible, use built-in errors before you invest the effort into creating your own custom error class.

## Key Take-aways

Errors are a fact of life in applications. The issue isn't whether or not errors occur. Rather, the issue is how they're handled. The key is to gain as much feedback as possible in order to apply corrective code. Different errors demand different solutions. It's a good idea to create and implement custom errors that extend the base error class.

> Try catch is not meant to be a tool that makes up for bad code.

Implementing **try catch** isn't meant to be a tool to make up for bad code. If a variable isn't supposed to be null or undefined, be sure to have unit tests that cover those scenarios. It's far better to fix issues at the source and right when they occur than trying to address issues downstream when the prospect of recovery may be less certain.

With respect to risk management, before implementing any remedial solution, you must measure relative costs. Context matters. Ask yourself if an ounce of prevention (**try catch**) is worth a pound of cure (expensive after-the-fact support). If performance is claimed, quantify it. If you don't undertake performance testing, in spite of what you may think, performance is not a real priority. Use **try catch** when there's a good prospect of errors occurring. Error scenarios tend to fall into a few predictable categories that involve other resources that may not be available, such as database, cloud assets, or anything that requires access to external resources.

John V. Petersen

CODE

### JavaScript Book Recommendation

If you're looking for a good book on improving your JavaScript code, check out Secrets of the JavaScript Ninja by Bear Bibeault and John Ressig from Manning Press. It's the kind of book you don't need to read cover to cover; you can flip around. The book assumes that the reader is at the intermediate level. In my opinion, this book is a must-have for anybody who includes JavaScript in their toolbox.

# Identify Voices with Microsoft Cognitive Services

During the cold war, both American and Russian spies were trying to outdo each other. In 1980, FBI agents recorded a phone call in which a man secretly arranged to meet with the soviet embassy in Washington DC. Alas, they could not pinpoint who the person was on the phone call. By 1985, the NSA were able to identify the specific person by using a technology

**Sahil Malik**
www.winsmarts.com
@sahilmalik

Sahil Malik is a Microsoft MVP, INETA speaker, a .NET author, consultant, and trainer.

Sahil loves interacting with fellow geeks in real time. His talks and trainings are full of humor and practical nuggets. You can find more about his training at http://www.winsmarts.com/training.aspx.

His areas of expertise are cross-platform mobile app development, Microsoft anything, and security and identity.

that could identify people by the sound of their voices. I don't know how true this story is (https://theintercept.com/2018/01/19/voice-recognition-technology-nsa/). I wasn't there. It doesn't sound implausible. It makes me wonder what technologies they have today!

Microsoft Cognitive Services, or as I like to call it, AI for the common folk, allows your applications to hear, see, and understand just as you would. In my previous article (CODE Magazine May/June 2018), I showed you how Cognitive Services can recognize people by simply submitting their photograph. It sounds like science fiction, but it was actually quite easy.

In this article, I'll demonstrate how you can build a library of voices associated with speakers. All I need is a 30 second audio sample of the person talking. Then I can submit an audio clip, something that the computer has never heard before. And the computer will recognize (hopefully) who that voice belongs to. This is possible because our voices are like fingerprints. They are unique, enough so that a computer can recognize a person from their voice with a very high degree of accuracy.

This is built purely using Microsoft Cognitive Services, which means that it's easy! If you can call a REST service, you can build a speaker recognition system by recognizing their voice.

In other words, something that only the NSA could do in 1985, you will be able to do by the end of this article.

Let's get started!

## The Concepts

As the name suggests, speaker recognition is the ability of a computer to listen to a voice and identify who the voice belongs to. There are two parts to this. They are very similar.

The first part is called **identification**, where you pass in a sample audio of spoken text and the computer tells you who that speaker is. Samples don't need to have identical text. The second part is where you use the voice for **authentication** or **verification**. The spoken phrase and the identity of the speaker need to be correct for an authentication to be performed.

In this article, I will focus only on speaker identification.

Before we dive into the nitty-gritty of code, you need to understand exactly what I intend to do. The process of speaker recognition first involves enrolling a speaker. This means that you create an identification profile and you enroll at least 30 seconds of audio of that person, not including silence, on their profile. You can do so using multiple files; if you upload a single file of 15 seconds in length, the computer will tell you that it needs an additional 15 seconds to complete the enrolling of a speaker.

You repeat this process for all speakers that you wish to enroll. You create an identification profile and enroll 30 seconds of audio for each one of the speakers. At the end of this, you'll have a collection of identification profiles.

Once you have some identification profiles, you supply a sample audio clip of any one of those speakers. This has to be spoken text but it doesn't have to exactly match the file that has been used to create the identification profile.

Because each voice is as distinctive as a thumbprint, the computer examines the sample of the voice and compares it against a list of identification profiles. The computer tells you who it thinks this person's voice belongs to.

And all of this is made possible using the speaker recognition API in Azure. In order to use it, you provision it in Azure, get the access keys, and call the necessary functionality over REST APIs.

## The Source Code

I want to keep this article pertinent to the important bits. If you're interested in a full working example of the code used in this article, you may download it from the following github repo.

```
https://github.com/maliksahil/speakerrecognition
```

Just remember to use your own API keys.

That code sample is in both C# written in dotnetcore, so you can run it on any platform, and Typescript. Whichever you prefer. The logic is identical, this article only shows Typescript code for brevity.

With that behind us, let's dive into code.

## Set up the Speaker Recognition API

Just like most other Cognitive Services APIs, before you can run speaker recognition API, you need to register it in Azure. To do that, sign into the Azure portal. Once

signed in, click on the **create resource** link and search for **speaker recognition**. Azure prompts you with a **speaker recognition API** choice; choose to create it.

You'll need to give it a name and place it in a location. Not all APIs are available in every data center; I put mine in West US. For the purposes of this article, you can go with the free tier. You'll also have to put this newly created API in a resource group. You can create a new one or put it in an existing resource group. Just remember to delete this API once you're done with it.

> Remember to delete the API once you've finished using it.

Once you click the Create button, the instance of the speaker recognition API is created. With the newly created resource, visit it on the **All Resources** link and copy a few things out of this newly provisioned speaker recognition API.

From the Overview tab, copy the endpoint. Under the Keys tab, copy the access keys.

Place both the endpoint and the access keys in the provided source code's config.ts file, as can be seen in **Listing 1**. I've removed the access keys from the listing because they were from my Azure subscription. You need to use yours.

## Authentication

All of the requests you make to the speaker identification API need to be authenticated. You need to use the authentication keys to perform this authentication. These authentication keys are passed in a special header called **Ocp-Apim-Subscription-Key**. Because every method needs to pass in this header, I chose to externalize the creation of this header in its own function, as can be seen in **Listing 2**.

## Create the Identification Profile

Next, you need to create the identification profile for a single user. Remember, you need to create identification profiles for each of the users from among whom you wish to recognize the spoken voice.

Creating the identification profile is a matter of calling the speaker recognition API endpoint + "/identificationProfiles". You make a REST call, and in the body of that message, you specify the locale, such as **en-us**. You also need to authenticate, for which you'll use the method shown in **Listing 2**.

The result of this POST method is a GUID, which is the identificationProfileId. This ID is your unique identifier per user; the idea is that when you ask the computer to recognize the speaker in an input audio file, you pass in a number of identification profile IDs, among which is the speaker who needs to be recognized. The full implementation of the createIdentificationProfile method can be seen in **Listing 3**. Note that it returns a Promise<string>, which is the identification profile ID GUID.

**Listing 1:** The config.ts file with the endpoint and access keys

```
export let config = {
    speakerRecognitionAPI: {
        endPoint:
     "https://westus.api.cognitive.microsoft.com/spid/v1.0",
        key1: "<removed>",
        key2: "<removed>"
    }
};
```

**Listing 2:** The authentication method

```
function getRequestOptions(): request.CoreOptions {
    return {
        headers: {
            "Content-Type": "application/octet-stream",
            "Ocp-Apim-Subscription-Key": config.speech.speakerRecognitionAPI.key1
        }
    };
}
```

**Listing 3:** The createIdentificationProfile method

```
export function createIdentificationProfile(): Promise<string> {
    const promise = new Promise<string>((resolve, reject) => {
       const requestOptions = getRequestOptions();
       requestOptions.headers['Content-Type'] = 'application/json';
       requestOptions.body = JSON.stringify({
            'locale': 'en-us'
        });

        request.post(
            config.speech.speakerRecognitionAPI.endPoint +
              '/identificationProfiles',
            requestOptions,
            (err, response, body) => {
                if (err) { reject(err); }
                else { resolve((
       JSON.parse(body).identificationProfileId)); }
            }
        );
    });
    return promise;
}
```

You can similarly delete an identification profile by using a DELETE request to the following URI:

```
config.speech.speakerRecognitionAPI.endPoint +
   '/identificationprofiles/' +
   identificationProfileId
```

Remember to authenticate using the method in **Listing 2.**

## Perform an Enrollment

Once you've registered a speaker by creating their identification profile, you need to enroll them. Enrolling the speaker means supplying 30 seconds of audio as spoken by the speaker. You can choose to supply a number of audio files, as long as they add up to 30 seconds, you're good to go.

When you supply an audio file for enrollment, you need to specify which identification profile ID this enrollment is for. In response, you will be informed if the enrollment is complete or incomplete. The enrollment is incomplete if you have less than 30 seconds of audio, in which case you'll also be informed how much more audio Cognitive Services needs to complete the enrollment.

> In order to enroll a speaker in an identification profile, you need 30 seconds of speaking, minus all silences.

In order to perform the enrollment, you need to issue a POST request to the following URI:

```
config.speech.speakerRecognitionAPI.endPoint +
 '/identificationProfiles/' +
 identificationProfileId +
 '/enroll';
```

As usual, remember to authenticate using the method shown in **Listing 2**.

In the body of your authentication POST message, include the audio file that you wish to enroll. The method used for reading this file can be seen in **Listing 4**.

Performing the enrollment is an asynchronous process. This means that you don't immediately receive a response for a successful or unsuccessful enrollment. Instead, you communicate through a URL to which you can periodically make a GET request to check the status of your enrollment. Remember not to make this request too often or you will get a HTTP 429 error. When you do receive a response, the response will be a JSON object with the property on it called **status**. if the value succeeds, your request has been accepted.

This doesn't mean that enrollment has finished. To check for that, you need to check the enrollmentStatus property and if that property says **Enrolling,** you need to supply more audio. The response also contains how much more audio you need to supply among its properties.

The full implementation of the createEnrollment method can be seen in **Listing 5.**

## Perform Identification

With the identification profiles set up and enrolled, the next thing you need to do is to identify the speaker who's given an input audio sample. The way this works is that you issue a POST request to the following URL:

```
config.speakerRecognitionAPI.endPoint +
'/identify?identificationProfileIds=' +
identificationProfileIds
```

The identificationProfileIds parameter is a comma-separated list of all the currently enrolled profile IDs.

As usual, remember to authenticate using the method shown in **Listing 2**.

When you do issue this POST request, in response, you receive a URL that you are supposed to check using a

**Listing 4:** The readFile method

```
function readFile(filePath: string) {
    const fileData =
       fs.readFileSync(filePath).toString("hex");
    const result = [];
    for (let i = 0; i < fileData.length; i += 2) {
        result.push(
         parseInt(
           fileData[i] + "" + fileData[i + 1], 16))
    }
    return new Buffer(result);
}
```

**Listing 5:** The createEnrollment method

```
export function createEnrollment(
    fileName: string, identificationProfileId: string):
    Promise<RequestStatus> {
    const promise = new Promise<RequestStatus>((resolve, reject) => {
        const requestOptions = getRequestOptions();
        requestOptions.body = readFile(__dirname + "/" + fileName)
        const uri = config.speakerRecognitionAPI.endPoint +
            '/identificationProfiles/' + identificationProfileId +
            '/enroll';
        request.post(
            uri,
            requestOptions,
            (err, response, body) => {
                if (err) { reject(false); }
                else {
const requestUrl = response.headers['operation-location'].toString();
console.log(requestUrl);
const timer = setInterval(() => {
    request.get(requestUrl, requestOptions,
        (err, response, body) => {
            const enrollmentStatus = new RequestStatus(JSON.parse(body));
            if (enrollmentStatus.status === 'succeeded') {
                clearInterval(timer);
                if (enrollmentStatus.processingResult.enrollmentStatus
                    === "Enrolling") {
                    console.log(
'Supply a little more audio, atleast 30 seconds. Currently at: '
            + enrollmentStatus.processingResult.speechTime);
                }
                resolve(enrollmentStatus);
            } else {
                console.log('Waiting for enrollment to finish');
            }
        });
}, 1000);
                }
            }
        )
    });
    return promise;
}
```

```
Waiting for enrollment to finish
Waiting for enrollment to finish
Supply a little more audio, atleast 30 seconds. Currently at: 27.54
1.wav enrolled
```

**Figure 1:** Output of enrolling an incomplete audio clip

```
Waiting for identification to finish
Waiting for identification to finish
Waiting for identification to finish
Waiting for identification to finish
Identified profile is:
          7043adc1-f9ef-4b47-8fa9-2b898c348869
          and the confidence is:
          High
```

**Figure 2:** Output of identifying the speaker

**Listing 6:** Identify the speaker

```
identification.identifySpeaker(
    'obama_input.wav',
  sahilIdentificationProfile + ',' +
obamaIdentificationProfile).then(
        result => {
    console.log(
        `Identified profile is:
            ${result.processingResult.identifiedProfileId}
            and the confidence is:
            ${result.processingResult.confidence}`)
});
```

periodic GET request. The result of that GET request is a JSON object. On the return JSON object, you check for a property called **status**. If the value of status is **succeeded**, you should also see the identification profile ID along with a confidence level that the computer thinks this audio belongs to a specific speaker.

Let's see all of this in action.

## Run the Code

Now that all of the code is written, let's create two identification profiles. I'm trying to identify two speakers. One is me, the second is former-president Obama.

For myself, I don't have a single 30-second audio sample, so I'll need multiple calls be able to enroll myself. Let's see how.

First, I need to create an identification profile. The following code snippet allows me to do that.

```
identification.createIdentificationProfile()
  .then(
    identificationProfile => {
        console.log(identificationProfile);
    });
```

Running this code snippet gives me a GUID of **537fa705-e576-4844-b0a8-be6e9d6f50b9**. Okay so, identificationProfileId for speaker Sahil is 537fa705-e576-4844-b0a8-be6e9d6f50b9.

Next, let's enroll my 30-second audio sample using the code snippet below:

```
identification.createEnrollment(
    '/Data/Sahil/1.wav',
    sahilIdentificationProfile).then(
        (result: RequestStatus) => {
            console.log('1.wav enrolled');
        });
```

The output of the above call can be seen in **Figure 1.**

As you can see, I need to supply a little more audio. I went ahead and supplied another few seconds of my audio, and the output now changes to this:

```
2.wav enrolled
```

Great! Now let's enroll former-president Obama. I ran the same code again and created an identificationProfile ID with the following GUID:

```
7043adc1-f9ef-4b47-8fa9-2b898c348869
```

I also went ahead and created an enrollment for our former president. This audio sample was > 30 seconds, so I was able to perform the enrollment in a single call. You can find all of the audio files I used in the associated code download.

Great! Now let's perform the final test: identifying the speaker. The call to identify the speaker can be seen in **Listing 6**. You might note that Obama_input.wav isn't part of my enrollment.

When I run the code in **Listing 6**, I get the output as seen in **Figure 2**.

It correctly recognized the speaker as the former president. If that doesn't impress you, I don't know what will.

## Conclusion

As I write this article, Google, Microsoft, and Facebook just concluded their annual flagship conferences. It almost seems like all the speakers on the stage have lost the ability to make a full sentence without using the term AI.

It is a buzzword and it's overused. Don't get desensitized to it because it holds tremendous value.

So far, I've covered what you could do with AI capabilities right out of the box using Microsoft Cognitive Services. It was easy! If you can call a REST service, your apps can now see, hear, speak, and understand the world around them, just like you do.

What if I told you that I wrote majority of this article using speech recognition? I did. My editor is pulling her hair right now (sorry Melanie), but it's true. I have limited keystrokes in my fingers and I wish to use them wisely. I have written 20+ books and thousands of articles, and I wrote many of them with speech recognition. I'll let you in on a secret, I use Google's Cognitive Services to do that, and over the years, I've built a profile for myself. The speech recognition has gotten so good and personalized to me, that it frankly leaves me amazed. Don't be-

lieve me? Go to docs.google.com and go to tools\voice typing. Try it, you'll be amazed! And it gets better the more you use it.

> Voice recognition gets better the more you use it.

I see applications for AI everywhere. I'm amazed at how little this field is tapped yet, and how much potential there is. Can you sense the excitement? Every time you pick up your smart phone and take your friend's picture, the phone recognizes faces and adjusts the contrast, focus, and brightness accordingly. How do you think it's recognizing faces? Artificial intelligence.

In subsequent articles, I hope to open this Pandora's box a little more.

I'll leave you with one thought. In 1985, NSA was using AI to do speaker recognition. They were so accurate and confident of their capabilities across millions of users that this became a reliable tool for thwarting espionage, or perhaps for committing espionage. What technology do think the NSA uses today? What technology does Azure, or AWS, or Google have in their data centers? Here's the good news: With the advent of the cloud, you have all that computing power at your fingertips, all of that power in your hands. Use it wisely!

Until next time, happy and responsible AI programming!

Sahil Malik
CODE

## GitHub Has the Code

The code for this article can be downloaded from GitHub: https://github.com/maliksahil/speakerrecognition.

# What's New in ASP.NET Core 2.1

ASP.NET Core 2.1 is the latest update to Microsoft's open source and cross-platform Web framework for .NET and is now available for production use. As you can see in Anthony Chu's article (in this same issue of CODE Magazine) ASP.NET Core 2.1 now includes support for real-time Web applications with SignalR. In additional to SignalR, ASP.NET Core 2.1 includes a variety of new features and improvements that help make Web development with .NET fast, reliable, and secure. Let's take a look at getting started with ASP.NET Core 2.1 and what features are new in this release.

**Daniel Roth**
daroth@microsoft.com
@danroth27

Daniel Roth is a Principal Program Manager on the ASP.NET team at Microsoft. He has previously worked on various parts of .NET, including System.Net, WCF, XAML, and ASP.NET. His passions include building frameworks for modern Web frameworks that are simple and easy to use.

## Get ASP.NET Core 2.1 with .NET Core 2.1

ASP.NET Core 2.1 is available with .NET Core 2.1. To set up your development computer to use ASP.NET Core 2.1, you'll need to install the .NET Core 2.1 SDK from https://dot.net/core. The .NET Core 2.1 SDK includes the .NET Core 2.1 runtime and the .NET Core command-line interface (CLI) for creating and working with .NET projects and solutions. In addition to carrying ASP.NET Core 2.1, .NET Core 2.1 includes loads of new functionality that's also worth checking out:

- Improved build performance
- A turbo-charged HttpClient
- Creation and sharing of .NET Core global tools
- New runtime types for efficient memory usage (Span<T>, Memory<T>, and friends)
- Alpine Linux support
- ARM32 support
- Brotli compression support
- New cryptography APIs and improvements
- SourceLink compatible package building capabilities
- Tiered JIT compilation

To use ASP.NET Core 2.1 (or any .NET Core 2.1 project) with Visual Studio on Windows, you'll need to install Visual Studio 2017 Update 7 or later. On a Mac, you'll need Visual Studio for Mac 7.5 or later. Note that .NET Core 2.1 is not yet included with Visual Studio, so you still need to separately install the .NET Core 2.1 SDK.

After you've installed the .NET Core 2.1 SDK, you can check that you now have the correct version by running **dotnet --version** from the command-line. The displayed version should be 2.1.300.

## Create Your First ASP.NET Core 2.1 App

Once you've installed the .NET Core 2.1 SDK, you can create your first ASP.NET Core 2.1 by running **dotnet new web -o WebApp1**. You can then run the app by changing to the directory of the new app and running **dotnet run**. Once the app is built and running, browse to **http://localhost:5000** to see "Hello world!" rendered in the browser.

If you look at the generated project file (WebApp1/WebApp1.csproj) it should look like this:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference
        Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

The generated project is set up to use the Web SDK (Microsoft.NET.Sdk.Web), which contains convenient tasks and targets for things like Web publishing and Razor (.cshtml) compilation. The project is also set up to target .NET Core 2.1 (netcoreapp2.1). There's a single package reference that brings in ASP.NET Core, called Microsoft.AspNetCore.App, which enables roll-forward semantics for servicing updates. The version of the Microsoft.AspNetCore.App package doesn't need to be specified because it's inferred by the Web SDK to match the version of ASP.NET Core bundled with the .NET Core SDK.

## HTTPS by Default

You may have noticed from the console output that in addition to listening on **http://localhost:5000**, the running app is also listening on **https://localhost:5001**. This is because ASP.NET Core 2.1 now enables HTTPS by default in development and has features that simplify setting up HTTPS in production.

Securing Web apps with HTTPS is more important than ever before. Browser enforcement of HTTPS is becoming increasingly strict. Sites that don't use HTTPS can be labeled as insecure. Browsers are also starting to enforce that new and existing Web features must only be used from a secure context. New privacy requirements, like the Global Data Protection Regulation (GDPR), require the use of HTTPS to protect user data. By using HTTPS during development, you're better prepared to run your app under HTTPS in production.

When you first ran any command from the .NET Core 2.1 SDK, it set up a development certificate for you. ASP.NET Core uses this development certificate to listen on HTTPS to see if the certificate is present. Initially, the development certificate is untrusted, so if you try to browse to **https://localhost:5001,** your browser will likely complain about the certificate.

To set up trust for the certificate on Windows or Mac OS, run **dotnet dev-certs https –trust**. On Windows, a dialog pops up to confirm that you want to trust the certificate.

On Mac OS, the certificate gets added to your keychain as a trusted certificate. On Linux, there isn't a standard way across distros to trust the certificate, so you'll need to perform the distro-specific guidance for trusting the development certificate.

## HTTPS Enforcement

ASP.NET Core 2.1 also includes features for enforcing HTTPS at runtime. You can redirect all HTTP traffic to HTTPS and direct browsers to enforce access to your site over HTTPS using the HTTP Strict Transport Security (HSTS) protocol. The ASP.NET Core project templates enable these HTTPS enforcement features for you.

To create a new ASP.NET Core Web app that has HTTPS enforcement enabled, run **dotnet new razor -o WebApp2** from the command-line, which generates a new ASP.NET Core project using ASP.NET Core Razor Pages. If you run the app and browse to the site, you should see that all HTTP requests are redirected to HTTPS. This redirection is handled using the new HTTPS redirection middleware, which is configured in the app's **Startup.Configure** method.

The call to **app.UseHttpsRedirection()** handles redirection of all HTTP request to HTTPS *if the current app is listening on HTTPS*. If HTTPS is being handled externally from the app, you can manually specify the HTTPS port using the ASPNETCORE_HTTPS environment variable or in code via the options pattern.

The HSTS middleware is also added via the **app.UseHsts()** call, but only when running in non-development environments. This is because HSTS enforcement is set up in the browser per host, and you probably don't want HSTS enforcement for all localhost requests.

## Razor Class Libraries

ASP.NET Core helps you write dynamic UI-rendering logic using a mixture of HTML and C# called Razor (cshtml). Razor class libraries are a new feature in ASP.NET Core that let you compile Razor views and pages into reusable class libraries that can be packaged and shared. You can create a new Razor class library by running **dotnet new razorclasslib** from the command-line or using the new project template in Visual Studio, as shown in **Figure 1**.

Once the Razor class library is referenced by the app, any views or pages it contains will then become part of the app. The app can tweak and customize the prebuilt UI by overriding specific views, pages, or partials in the application. For example, you can add a Razor class library to your app with the following page (Areas/MyFeature/Pages/Page1.cshtml):

```
@page
@model RazorClassLib1.MyFeature.Pages.Page1Model

<h1>Page1 from Razor class library</h1>

<partial name="_OverrideMe" />
```

This code uses the following partial view (Areas/MyFeature/Pages/Shared/_OverrideMe.cshtml):

```
<h2>Override me</h2>
```

After you reference this Razor class library from the app, you can browse to /MyFeature/Page1 and see the page, as shown in **Figure 2**.

The page doesn't have any layout specified, so it doesn't look like the rest of the app. To specify that the content of the Razor class library should use the app's layout,

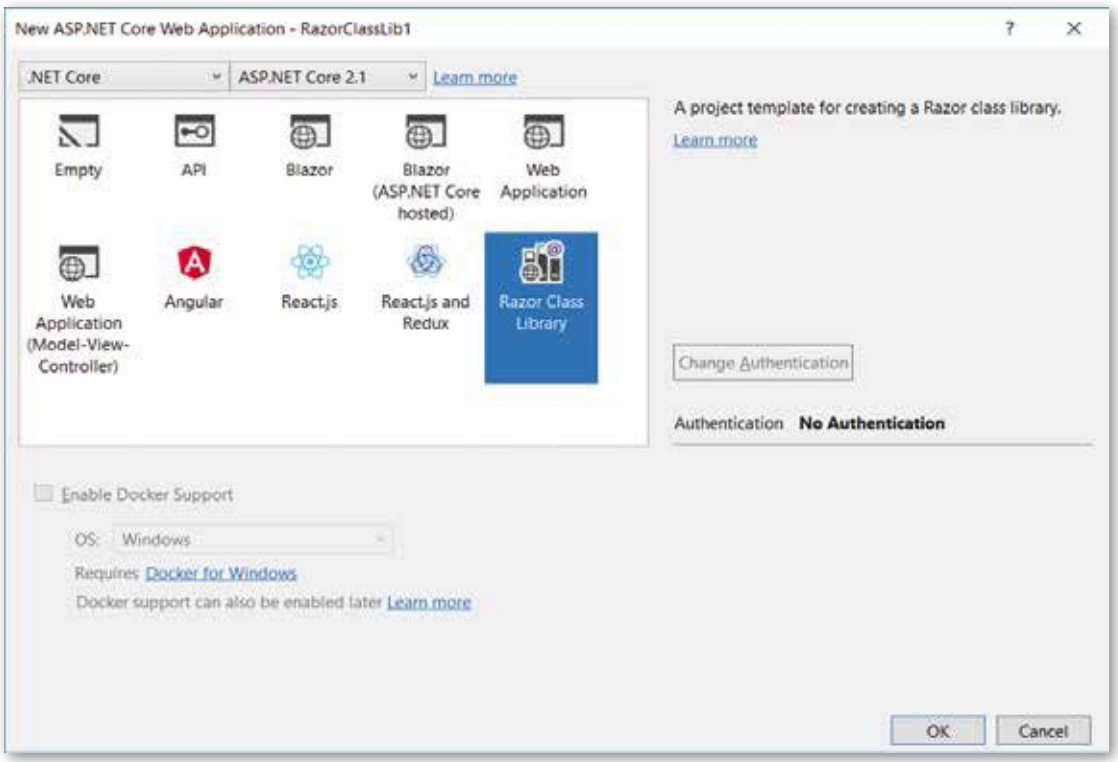



**Figure 1:** Creating a new Razor Class Library from Visual Studio

### Understanding .NET Core SDK Versions

If the output of **dotnet --version** says 2.1.200, you do ***not*** have the .NET Core 2.1 SDK on your computer. The SDK version is technically independent of the runtime version it carries, which catches a lot of users by surprise. The SDK version 2.1.200 carries .NET Core 2.0, not 2.1. Starting with SDK version 2.1.300, the major and minor version of the SDK indicates the runtime version it carries. In the future, the SDK patch version will increment by 100 for feature updates and by one for patch updates.


**Figure 2:** Rendering a Razor Page from a Razor Class Library

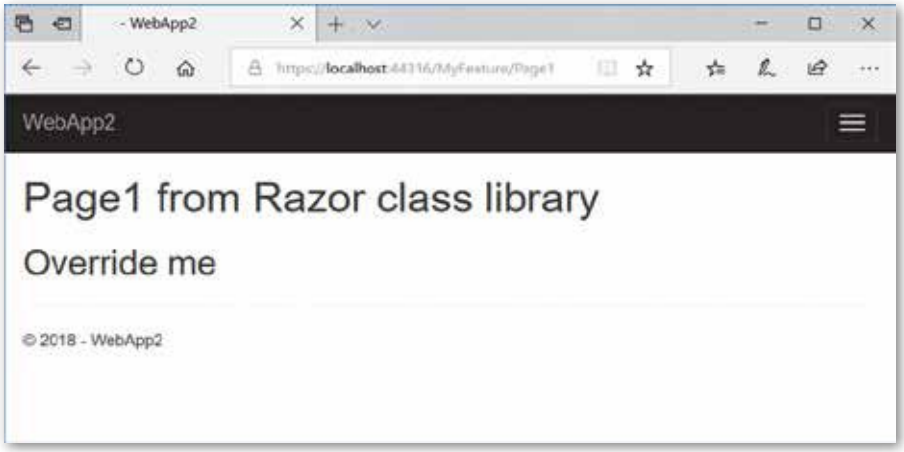



**Figure 3:** The Razor class library with app styles

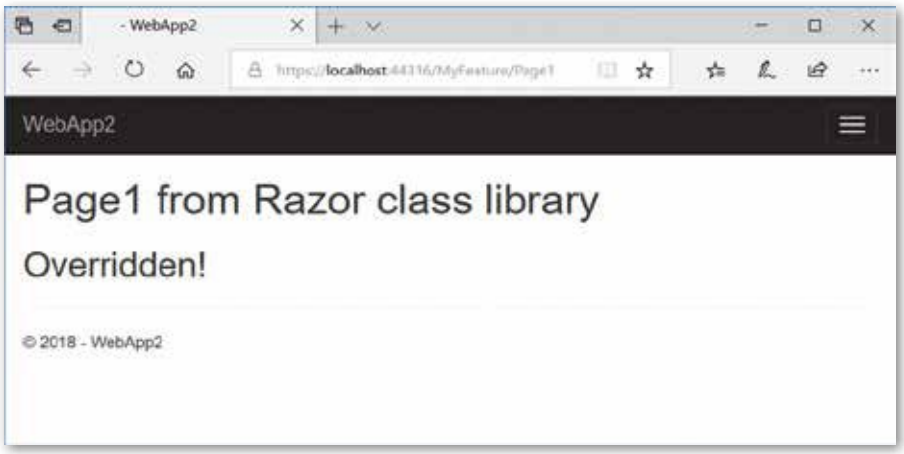



**Figure 4:** The overridden partial in the Razor class library

you can add a **_ViewStart.cshtml** to the app using the same folder hierarchy as the library (Areas/MyFeature/Pages/_ViewStart.cshtml):

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
}
```

If you rerun the app, the page from the Razor class library now picks up the app's layout, as shown in **Figure 3**.

You can also override specific pages, views, and partials in the Razor class library. To override the partial view used by the page, add the following partial view to the app (Areas/MyFeature/Pages/Shared/_OverrideMe.cshtml):

```
<h2>Overridden!</h2>
```

Refresh the page and you should see the change shown in **Figure 4**.

To share a Razor class library, run **dotnet pack** on the library project and then publish the produced NuGet package to your favorite feed.

## Identity UI Library

In previous releases of ASP.NET Core, the ASP.NET Core Identity UI logic for registering new users, handling logins, and managing user accounts could only be added to a new project on project creation, resulting in dozens of files and hundreds of lines of code being added to the new project. If you started a new project without identity and then decided it was needed, there was no way to add it other than to create a new project with identity and copy the code over. Because the code was part of the application, maintaining the identity UI code was the responsibility of the app owner, as Microsoft has no way to patch it other than to advise users to update their code.

> Adding identity support to an existing application is now as easy as adding the identity UI package and configuring the required services.

ASP.NET Core 2.1 includes a Razor class library implementation of the ASP.NET Core Identity UI. Adding identity support to an existing application is now as easy as adding the identity UI package (Microsoft.AspNetCore.Identity.UI) and configuring the required services. Any patches or updates to the identity UI can be handled using the normal servicing mechanisms. To help with customizing the many pages that make up the identity UI, ASP.NET Core 2.1 also includes an identity scaffolder that can be used to add identity to an existing project and to override specific pages.

To add identity support to an existing ASP.NET Core project, right-click on the project and select **Add > New Scaffolded Item**. In the Add Scaffold dialog, choose the Identity scaffolder and select OK. The Add Identity dialog appears, as shown in **Figure 5,** and gives you lots of options for customizing the scaffolded identity code. You can select an existing layout, choose which pages you want to override, and create a data context class for storing the identity data with an optional custom user class.

After the identity scaffolder has finished running, a new Identity area has been added to your application with an **IdentityHostingStartup** class that adds the required identity-related services to the application. The Scaf-

foldingReadme.txt file contains instructions to set up the database and any additional required middleware. Once you've added the generated login partial to your layout, the app is ready to start registering and logging in new users.

## Privacy and GDPR Compliance

Protecting the privacy of users is important for any website that handles user data. ASP.NET Core 2.1 makes it easier to comply with the latest privacy requirements, including the EU General Data Protection Regulation (GDPR). In addition to enabling HTTPS by default, ASP.NET Core apps are equipped with infrastructure for managing cookie consent and deleting and exporting user data.

When you browse to your app for the first time, the cookie consent UI appears, providing a link to the app's privacy policy page and an option to consent to the use of cookies on the site, as shown in **Figure 6**. You can configure which cookies are considered essential to the app's functionality and which cookies should only be used once the user has given their consent.

Logged-in users also have the option to delete their data or export it through the user profile page in the identity UI, as shown in **Figure 7**.

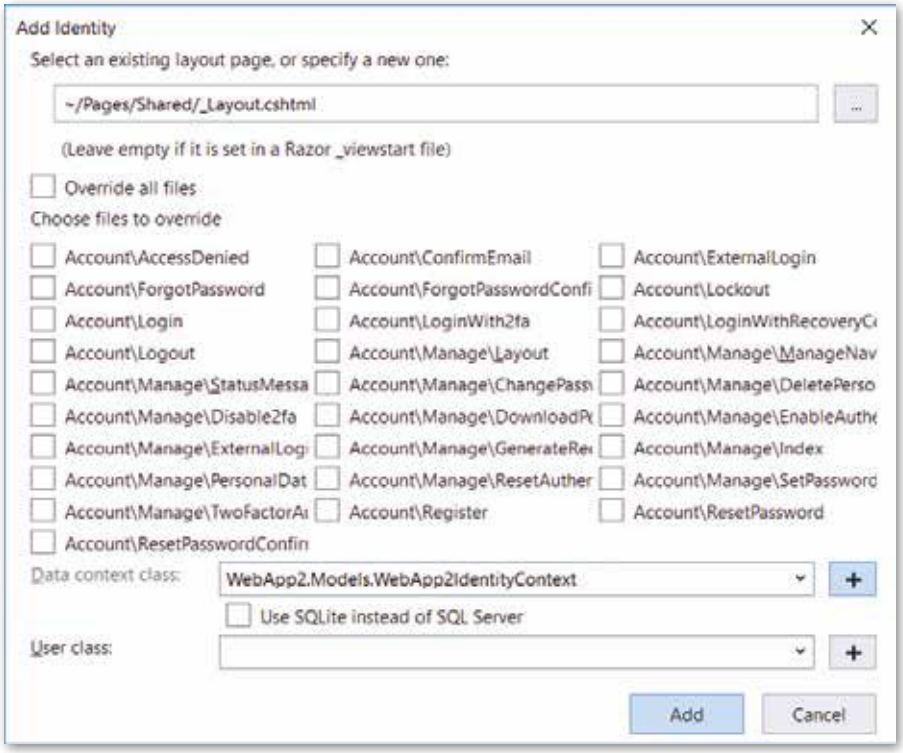



**Figure 5:** Add Identity scaffolding options.

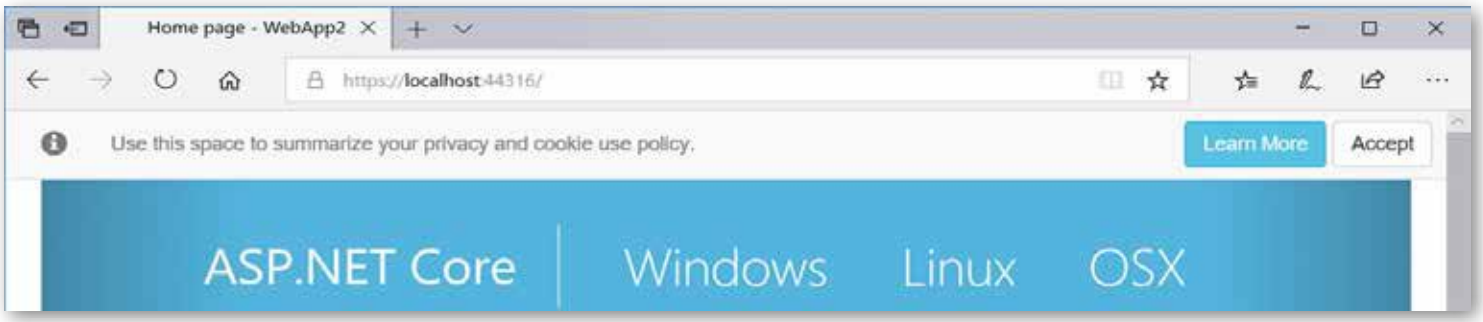



**Figure 6:** The default cookie consent UI and privacy policy link

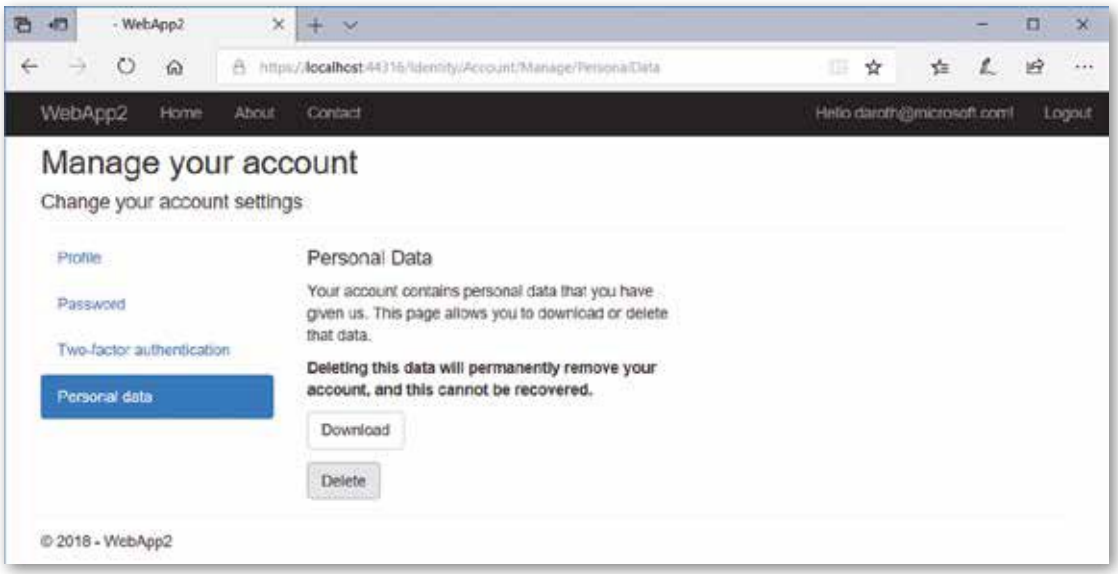



**Figure 7:** The default identity UI enables users to export and delete their user data.

**Listing 1:** A controller using the new Web API controller conventions and ActionResult<T>

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    private readonly ProductsRepository _repository;

    public ProductsController(
        ProductsRepository repository)
    {
        _repository = repository;
    }

    [HttpGet]
    public IEnumerable<Product> Get()
    {
        return _repository.GetProducts();
    }

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        if (!_repository.TryGetProduct(id, out var product))
        {
            return NotFound();
        }
        return product;
    }

    [HttpPost]
    public ActionResult<Product> Post(Product product)
    {
        _repository.AddProduct(product);
        return CreatedAtAction(nameof(Get),
            new { id = product.Id }, product);
    }
}
```

### Generate a C# Class from JSON

You can use Visual Studio to quickly generate a C# class from a JSON example. To create a GitHubUser type that matches the JSON returned from the GitHub API, copy the JSON example from https://developer.github.com/v3/users/ and then in Visual Studio select **Edit > Paste Special > Paste JSON** as Classes. Visual Studio generates a class for you that matches the inferred schema from the JSON example.

## [ApiController] and ActionResult<T>

ASP.NET Core 2.1 introduces new Web API controller-specific conventions that make it easier to build clean and descriptive Web APIs. The following Web API-specific conventions can be applied to a controller using the new **[ApiController]** attribute:

- Automatically respond with a 400 when validation errors occur; there's no need to check the model state in your action method.
- Infer smarter defaults for action parameters: **[FromBody]** for complex types, **[FromRoute]** when possible, and otherwise **[FromQuery]**.
- Require attribute routing; actions aren't accessible by convention-based routes.

You can also now return **ActionResult<T>** from your Web API actions, which enables returning arbitrary action results or a specific return type (thanks to some clever use of implicit cast operators).

You can check out a Web API controller that uses these new conventions in **Listing 1**.

## IHttpClientFactory

ASP.NET Core applications also need to be able to make requests to HTTP endpoints. The **HttpClient API** is the preferred way to send HTTP requests in .NET, but its usage suffers from a few pitfalls. First, to make efficient use of network connections on the server, **HttpClient** instances typically need to be re-used instead of creating new instances and then throwing them away. Also, creating new **HttpClient** instances throughout your code makes it difficult to configure them in any sort of reasonable and cross-cutting way.

ASP.NET Core 2.1 provides **HttpClient** instances *as a service* through the new **IHttpClientFactory** interface. You can centrally configure multiple named or typed HttpClients in your **Startup.ConfigureServices** method, including specifying base addresses and default headers. You can also configure an outgoing request-handling pipeline using a message handler that can deal with concerns like retry logic and circuit breakers.

Configuring a named **HttpClient** for a specific Web API looks like this:

```
services.AddHttpClient("Example", client =>
{
    client.BaseAddress =
        new Uri("https://api.example.com/");
});
```

You can then request this specific HttpClient in your code using the **IHttpClientFactory** service.

```
public ExampleModel(IHttpClientFactory factory)
{
    Client = factory.CreateClient("Example");
}
```

You can also define typed clients that get injected with specifically configured HttpClient instances. For example, here's how you might set up a typed client for calling the GitHub API:

```
services.AddHttpClient<GitHubClient>(client =>
{
    client.BaseAddress =
        new Uri("https://api.github.com");
    client.DefaultRequestHeaders.Add("Accept",
        "application/vnd.github.v3+json");
    client.DefaultRequestHeaders.Add(
        "User-Agent", "James Bond");
})
.AddTransientHttpErrorPolicy(policy =>
   policy.WaitAndRetryAsync(3, count =>
   TimeSpan.FromSeconds(Math.Pow(2, count))));
```

The typed **GitHubClient** then gets the configured **HttpClient** injected in its constructor. Like this:

```
public class GitHubClient
{
    public GitHubClient(HttpClient client)
    {
        Client = client;
    }
```

```
    public HttpClient Client { get; }

    public async Task<GitHubUser> GetUser(
        string username)
    {
        var result = await Client
            .GetAsync($"/users/{username}");
        result.EnsureSuccessStatusCode();
        return await result
            .Content.ReadAsAsync<GitHubUser>();
    }
}
```

You can then inject the typed client into your controllers or pages:

```
public class ContactModel : PageModel
{
    public ContactModel(GitHubClient ghClient)
    {
        GhClient = ghClient;
    }

    public GitHubClient GhClient { get; }

    public string Message { get; set; }

    public async Task OnGet()
    {
        var user = await GhClient
            .GetUser("danroth27");
        Message = $"Name: {user.name}," +
            "Company: {user.company}";
    }
}
```

To set up output messaging handling, you can add message handlers to your HttpClient instances generated by the factory. For example, you could add retry logic using the messaging handler you implemented yourself.

```
services.AddHttpClient("Example", client =>
{
    client.BaseAddress =
        new Uri("https://api.example.com/");
})
.AddHttpMessageHandler(() => new MyRetryHandler());
```

ASP.NET Core 2.1 provides integration with Polly, a .NET resilience and transient-fault-handling library that allows you to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a thread-safe manner. This integration is provided through the Microsoft.Extensions.Http.Polly package. For example, using Polly to add retry logic with an exponential back-off looks like this:

```
services.AddHttpClient<GitHubClient>(client =>
{
    client.BaseAddress =
        new Uri("https://api.github.com");
    client.DefaultRequestHeaders.Add("Accept",
        "application/vnd.github.v3+json");
    client.DefaultRequestHeaders.Add(
        "User-Agent", "James Bond");
})
```

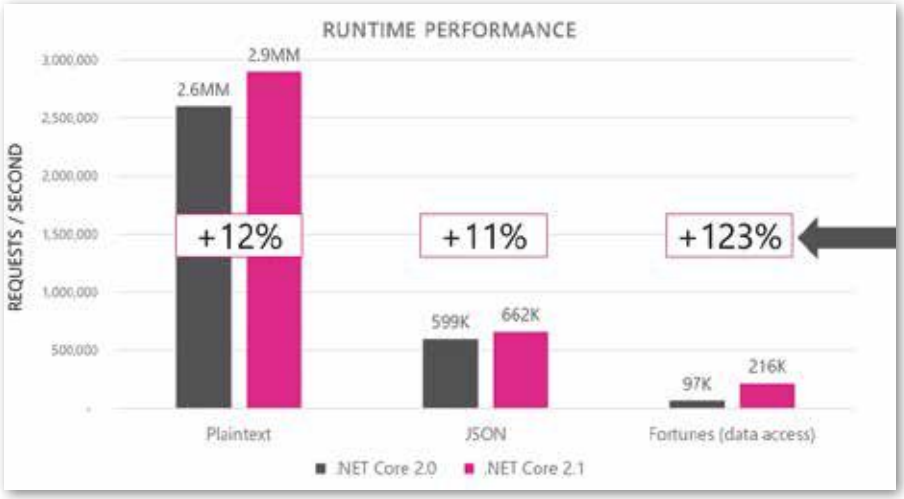



**Figure 8:** .NET Core 2.1 runtime performance on the TechEmpower benchmarks.

```
.AddTransientHttpErrorPolicy(policy =>
    policy.WaitAndRetryAsync(3, count =>
    TimeSpan.FromSeconds(Math.Pow(2, count))));
```

## Performance

ASP.NET Core benefits from runtime performance improvements in .NET Core 2.1, including some dramatic improvements in data access performance. One of the ways that .NET Core performance is measured and compared to other frameworks is using the public TechEmpower benchmarks. In internal lab runs at Microsoft, .NET Core 2.1 throughput performance is 10% greater than .NET Core 2.0 for the plain text and JSON scenarios, and a whopping 123% greater for the more realistic data access scenario, solidifying .NET Core's position as one of the fastest frameworks available (see **Figure 8**).

## Summary

ASP.NET Core and .NET Core have come a long way since they first shipped two years ago. ASP.NET Core 2.1 is now a very complete and feature-rich offering for building cross-platform Web and server applications. In addition to the features discussed in this article, ASP.NET Core 2.1 includes lots of other improvements, including improved testability of MVC applications and better integration with single-page application frameworks like Angular and React. Whether you are already on the .NET Core train or thinking about jumping on board for the first time, ASP.NET Core 2.1 is worth a look.

Daniel Roth

# Introducing .NET Core 2.1 Flagship Types: Span<T> and Memory<T>

.NET Core 2.1 is the latest release of the general-purpose development platform maintained by Microsoft and the .NET open source community. .NET Core is cross-platform and open source and consists of a .NET runtime, a set of reusable framework libraries, a set of SDK tools, and language compilers. Amongst many great features, this new release focuses on performance and brings us the System.Memory library that's available right out of the box and is also available as a standalone package on NuGet. Today, .NET developers write performance-critical server applications and scalable cloud-based services that are sensitive to memory consumption. To address these developer scenarios, .NET Core 2.1 introduces two flagship types into the ecosystem, namely Span<T> and Memory<T>, which are used to provide scalable APIs that don't allocate buffers and avoid unnecessary data copies.

**Ahson A. Khan**
ahkha@microsoft.com
www.linkedin.com/in/ahsonkhan
https://github.com/ahsonkhan

Ahson A. Khan is a software engineer at Microsoft and works on the .NET team within Cloud and Enterprise. His primary focus over the last two years has been on developing high-performance libraries as part of the cross-platform and open source .NET Core project. Ahson has a Bachelor's of Software Engineering from the University of Waterloo.

## Hot off the Press: Span<T>

**Span<T>** is a newly defined type in .NET within the System namespace that provides a safe and editable view into any arbitrary contiguous block of memory with no-copy semantics. You can use Span<T> as an abstraction to uniformly represent arrays, strings, memory allocated on the stack, and unmanaged memory. In some ways, it's analogous to C# arrays, but with the added ability to create a view of a portion of the array without allocating a new object on the heap or copying the data. This feature is called **slicing** and types with this feature are known as sliceable types. Span promises type and memory safety with checks to avoid out-of-bounds access, but that type of safety comes with certain usage restrictions enforced by the C# compiler and runtime. There's also a corresponding read-only flavor of Span<T>, unsurprisingly called **ReadOnlySpan<T>**. Span<T> and the other types discussed here are part of the .NET Core 2.1 release and require C# 7.2 language version to use.

> Span promises type and memory safety.

**Note:** Going forward, I'll refer to Span<T> and ReadOnlySpan<T> as span, for brevity. When discussing something unique to ReadOnlySpan, or specific to the element type T, I'll make an explicit distinction.

## Let's Take a Peek Inside Span

Roughly speaking, you can visualize span as a struct containing two fields: a pointer and a length (see **Figure 1**).


**Figure 1:** You can conceptualize Span<T> like this.

Now, suppose you have an array of bytes allocated somewhere on the heap. You can wrap a span around this byte array by passing it to the span constructor. Doing so assigns the pointer field to the memory address where the data starts (0th element of the array) and sets the length field to the number of consecutive accessible elements (in this case, it's the length of the array), as shown in **Figure 2**.

If you're only interested in a portion of the array, you can slice the span to get the desired view (**Figure 3**). Slicing is quite efficient because you don't need to allocate anything on the heap or copy any data when you're creating the new span.

Let's see how you can do all this in C# code. Span has a constructor that accepts an array and there's an extension method on the array itself to support fluent interface (method chaining). The implicit cast from array to span makes the conversions easy, especially when passing arrays to methods that accept spans.

```
byte[] array = new byte[4];

// using ctor: public Span(T[] array)
Span<byte> span = new Span<byte>(array);
// using AsSpan extension method
Span<byte> alt = array.AsSpan();

Span<byte> slice = alt.Slice(start:1, length:2);
```

Keep in mind that spans are only a **view** into the underlying memory and aren't a way to instantiate a block of memory. Span<T> provides read-write access to the memory and ReadOnlySpan<T> provides read-only access. Therefore, creating multiple spans on the same array creates multiple views of the same memory. Like with arrays,

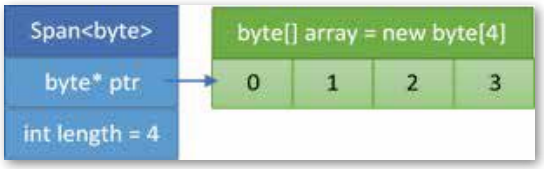



**Figure 2:** Span<byte> wrapping a byte array points to its start.


**Figure 3:** Slicing a Span<byte> changes its pointer and length fields.

you can use the Span<T> indexer to access or modify the underlying data directly. Furthermore, slices of spans allow you to safely access the data within the viewing window because spans enforce boundary checks. Let's try to modify the underlying array using the span indexer and see how the changes affect the elements of the array (observe the modifications to the array in **Figure 4)**. Notice that span prohibits access to elements outside the window by throwing an IndexOutOfRangeException.

> Span<T> provides read-write access to the memory and ReadOnlySpan<T> provides read-only access.

```
string[] array = { "a", "b", "c", "d", "e" };
// Using Span ctor (array, start, length)
// Note that the spans overlap
var firstView = new Span<string>(array, 0, 3);
var secondView = new Span<string>(array, 2, 3);

firstView[0] = "w";
// array = { "w", "b", "c", "d", "e" }
firstView[2] = "x";
// array = { "w", "b", "x", "d", "e" }
secondView[0] = "y";
// array = { "w", "b", "y", "d", "e" }

// Throws IndexOutOfRangeException
firstView[4] = "a";
```

Span<T>, as conceptualized in **Figure 1**, is available as part of .NET Core 2.1 and can be used by any application that targets it, right out of the box. However, this variety of span (which I'll refer to as "built-in span"), is only available on applications running on .NET Core 2.1. This is because you require a newer version of the runtime and just-in-time compiler (JIT) that understands the semantics of span as described so far. The actual implementation of span contains a field of an internally defined type (instead of a raw pointer), which the runtime recognizes as a JIT intrinsic. When the JIT encounters this field at runtime, it generates assembly code as if span had contained a **ref T** field instead. That's how you get the pointer-like semantics and performance characteristics described earlier while sidestepping the memory safety concerns that are common when using pointers directly. Because span contains what can be thought of as a ref T field, it's considered to be a **ref-like** type. For my purposes here, that essentially means that the lifetime of span is constrained to the execution stack. I'll further explore the implications of ref-like types later in this article. Also, if you are curious, you can view the original specification document for span by following the links in the reference section (https://aka.ms/span-spec).

At this point, you might be wondering if you can still use span in applications that target older runtimes, like .NET Core 2.0, or ones running on .NET Framework for desktop. You certainly can, by using the portable implementation of span. **Portable span** doesn't require special handling from the JIT compiler or a runtime with special

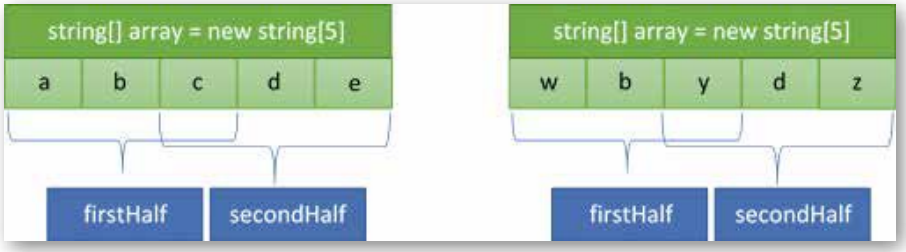



**Figure 4:** An example of how the contents of an array can be changed by multiple spans using the indexer. Original (left), after (right).

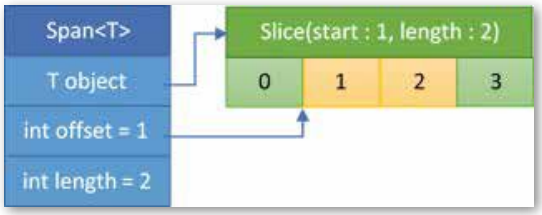



**Figure 5:** Slicing portable span affects the offset field while keeping the object field unchanged.

knowledge of the type. It essentially consists of three fields (instead of the two described previously): an object, an offset, and a length (see **Figure 5**).

> The lifetime of span is constrained to the execution stack.

Due to the extra field and computation of the offset, some portable span APIs are slightly slower than the *built-in span* APIs that come with .NET Core 2.1. You can use portable span by adding a reference to the **System.Memory** NuGet package in your applications directly (https://aka.ms/nuget-package). The System.Memory package is compatible with any .NET platform that implements the **.NET Standard 1.1** specification. Basically, you can reference the System.Memory package on all the active .NET platforms that developers target, as detailed in **Figure 6**.

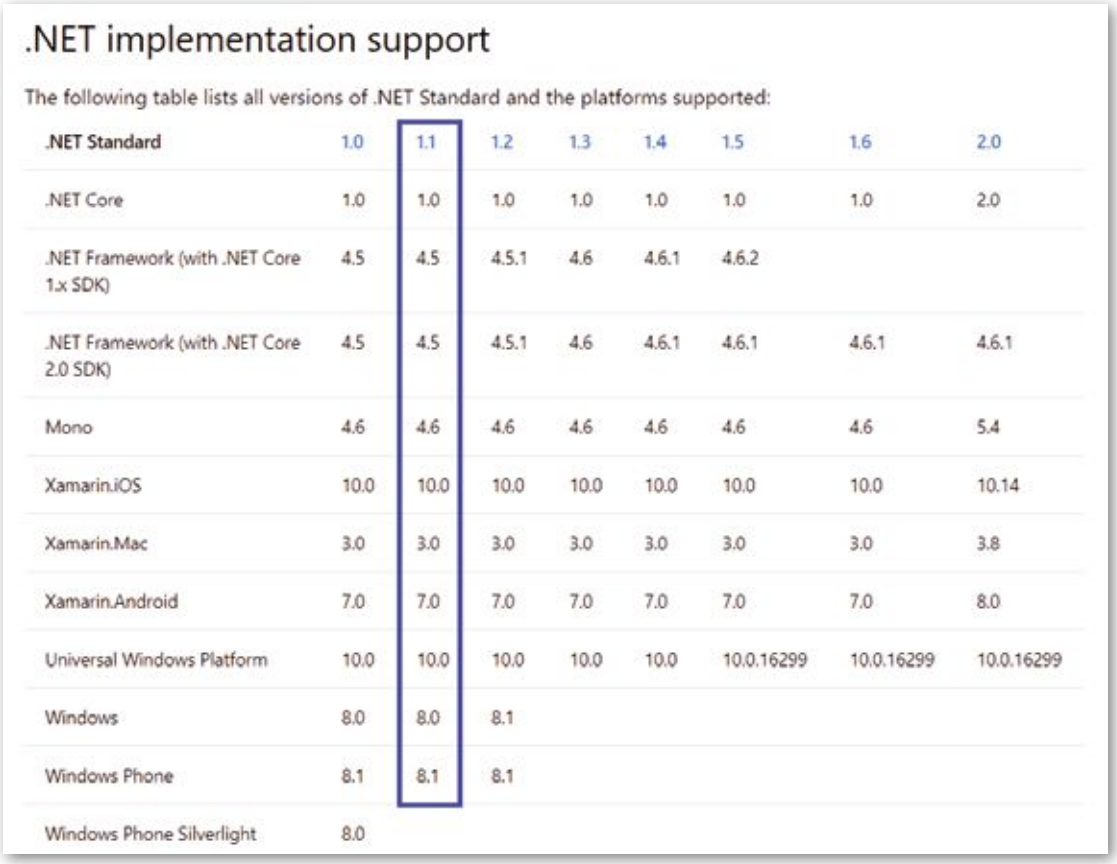
> If you're unfamiliar with .NET Standard, please refer to the references side bar for more details.

## Some Disassembly Required

Let's walk through some code samples that showcase usage and performance of Span APIs in more detail.

### *Sample 1: Return the Sum of the Elements of a Byte Array*

Consider the naive implementation of a method that accepts a byte array as a parameter and returns the sum of all the elements by looping through it.



## .NET implementation support

The following table lists all versions of .NET Standard and the platforms supported:

| .NET Standard | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 2.0 |
|---|---|---|---|---|---|---|---|---|
| .NET Core | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 2.0 |
| .NET Framework (with .NET Core 1.x SDK) | 4.5 | 4.5 | 4.5.1 | 4.6 | 4.6.1 | 4.6.2 | | |
| .NET Framework (with .NET Core 2.0 SDK) | 4.5 | 4.5 | 4.5.1 | 4.6 | 4.6.1 | 4.6.1 | 4.6.1 | 4.6.1 |
| Mono | 4.6 | 4.6 | 4.6 | 4.6 | 4.6 | 4.6 | 4.6 | 5.4 |
| Xamarin.iOS | 10.0 | 10.0 | 10.0 | 10.0 | 10.0 | 10.0 | 10.0 | 10.14 |
| Xamarin.Mac | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.8 |
| Xamarin.Android | 7.0 | 7.0 | 7.0 | 7.0 | 7.0 | 7.0 | 7.0 | 8.0 |
| Universal Windows Platform | 10.0 | 10.0 | 10.0 | 10.0 | 10.0 | 10.0.16299 | 10.0.16299 | 10.0.16299 |
| Windows | 8.0 | 8.0 | 8.1 | | | | | |
| Windows Phone | 8.1 | 8.1 | 8.1 | | | | | |
| Windows Phone Silverlight | 8.0 | | | | | | | |

**Figure 6:** These .NET platforms support .NET Standard 1.1 and can use portable span.

```
public static int ArraySum(byte[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

What changes would you have to make to use span instead? Can you spot the difference?

```
public static int SpanSum(Span<byte> data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

In case you missed it, the only change necessary is to the type of the input parameter (from **byte[]** to **Span<byte>**). Additionally, no code changes are needed at the call site in this case, because you can rely on the implicit cast from array to span.

```
static void Main(string[] args)
{
    byte[] data = { 1, 2, 3, 4, 5, 6, 7 };
    ArraySum(data); // returns 28
    SpanSum(data);  // returns 28
}
```

You might be asking what's the catch? Is there a performance downside of using a span over an array? Let's answer this by comparing the disassembly of the two methods (the annotated machine instructions are shown in **Figure 7** and **Figure 8**). If you focus on the instructions comprising the loop body (lines 7-12 versus 8-13), you'll notice that the disassembly is essentially identical (barring some registers being swapped). This was made possible by meticulous optimizations to the JIT.

As an aside, the other differences within the disassembly (outside the loop) are due to quirks in how the JIT assigns arguments and local variables to registers and should have a negligible effect on runtime performance. Let's confirm that by measuring the execution time of the two methods using the .NET benchmarking library—BenchmarkDotNet.

```
private byte[] data;

[GlobalSetup]
public void Setup()
{
    data = new byte[10_000];
    new Random(42).NextBytes(data);
}

[Benchmark(Baseline = true)]
```

```
public int ArraySum() => ArraySum(data);

[Benchmark]
public int SpanSum() => SpanSum(data);
```

> There's practically no difference in performance between iterating spans and arrays.

As you can see from the output, there is practically no difference in performance between iterating spans and arrays. The small difference in the execution time is within the margin of error.

### *Sample 2: Return the Sum of a String Containing Comma Separated Integers*

Let's parse some numbers by using the familiar and widely available string APIs. Notice that you pass in a substring to the int.Parse() method.

```
public static int StringParseSum(string data)
{
    int sum = 0;
    // allocates
    string[] splitString = data.Split(',');
    for (int i = 0; i < splitString.Length; i++)
    {
        sum += int.Parse(splitString[i]);
    }
    return sum;
}
```

Now, to re-write the method with span, you can leverage the span-based overload of int.Parse, which is one of the many overloads added in .NET Core 2.1. The distinction here is that you pass in a slice of a span to the parse method instead of a substring.

```
ReadOnlySpan<char> span = data;
int sum = 0;
while (true)
{
    int index = span.IndexOf(',');
    if (index == -1)
```

```
1.  4560  xor      eax,eax                  ; eax = sum = 0
2.  4562  xor      edx,edx                  ; edx = i = 0
3.  4564  mov      r8d,dword ptr [rcx+8]    ; r8d = data.Length
4.  4568  test     r8d,r8d                  ; if (Length <= 0) return
5.  456B  jle      00007FFE86184580
6.
7.  456D  movsxd   r9,edx                   ; sign-extend i (start of loop)
8.  4570  movzx    r9d,byte ptr [rcx+r9+10h] ; r9d = data[i]
9.  4576  add      eax,r9d                  ; sum += data[i]
10. 4579  inc      edx                      ; i++
11. 457B  cmp      r8d,edx                  ; if (Length > i) goto start of loop
12. 457E  jg       00007FFE8618456D
13.
14. 4580  ret
```

**Figure 7:** Here's the disassembly of the method iterating over a byte array.

```
1.  45A0  mov      rax,qword ptr [rcx]      ; rax = ref data.GetRef()
2.  45A3  mov      edx,dword ptr [rcx+8]    ; edx = data.Length
3.  45A6  xor      ecx,ecx                  ; ecx = sum = 0
4.  45A8  xor      r8d,r8d                  ; r8d = i = 0
5.  45AB  test     edx,edx                  ; if (Length <= 0) return
6.  45AD  jle      00007FFE861845C2
7.
8.  45AF  movsxd   r9,r8d                   ; sign-extend i (start of loop)
9.  45B2  movzx    r9d,byte ptr [rax+r9]    ; r9d = data[i]
10. 45B7  add      ecx,r9d                  ; sum += data[i]
11. 45BA  inc      r8d                      ; i++
12. 45BD  cmp      r8d,edx                  ; if (i < Length) goto start of loop
13. 45C0  jl       00007FFE861845AF
14.
15. 45C2  mov      eax,ecx
16. 45C4  ret
```

**Figure 8:** The disassembly of iterating over a byte span is like iterating over an array.

```
BenchmarkDotNet=v0.10.14, OS=Windows 10.0.16299.371 (1709/FallCreatorsUpdate/Redstone3)
Intel Xeon CPU E5-1620 v2 3.70GHz, 1 CPU, 8 logical and 4 physical cores
Frequency=3604597 Hz, Resolution=277.4235 ns, Timer=TSC
.NET Core SDK=2.1.300-rc1-008662
  [Host]     : .NET Core 2.1.0-rc1-26423-06 (CoreCLR 4.6.26423.02, CoreFX 4.6.26423.06), 64bit RyuJIT
  DefaultJob : .NET Core 2.1.0-rc1-26423-06 (CoreCLR 4.6.26423.02, CoreFX 4.6.26423.06), 64bit RyuJIT
```

| Method | Mean | Error | StdDev | Scaled | ScaledSD |
|---|---|---|---|---|---|
| ArraySum | 5.649 us | 0.1751 us | 0.1946 us | 1.00 | 0.00 |
| SpanSum | 5.589 us | 0.0162 us | 0.0144 us | 0.99 | 0.03 |

**Figure 9:** The benchmarking results show that array and span performance is identical.

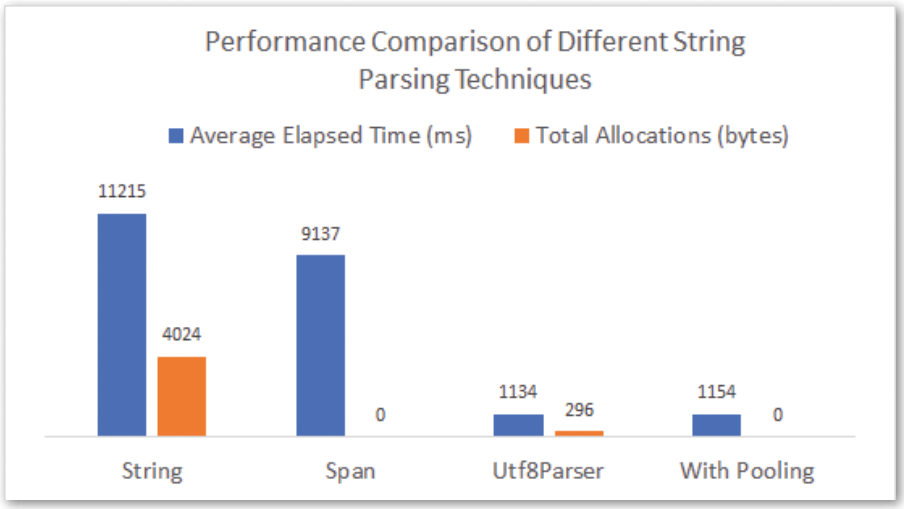



**Figure 10:** Comparing the performance of using string APIs versus span-based APIs to parse a string of integers.

```
    {
        sum += int.Parse(span);
        break;
    }
    sum += int.Parse(span.Slice(0, index));
    span = span.Slice(index + 1);   // skip ','

}
return sum;
```

Let's go one step further with the string parsing code to explore the benefits of using the new span-based **Utf8Parser** APIs (which is also within the System.Memory library accessible from the System.Buffers.Text namespace). Strings in C# are encoded as UTF-16 characters. By making a small compromise between execution-time performance and the number of bytes allocated, you can encode the incoming string data as UTF-8 characters by calling the GetBytes method on the System.Text.Encoding class.

```
// allocates
Span<byte> utf8 = Encoding.UTF8.GetBytes(data);
int sum = 0;
while (true)
{
    Utf8Parser.TryParse(span, out int value,
        out int bytesConsumed);
    sum += value;
    if (utf8.Length - 1 < bytesConsumed)
        break;
    // skip ','
    utf8 = utf8.Slice(bytesConsumed + 1);
}
return sum;
```

You can get rid of the allocation due to the encoding step by renting an array from an **ArrayPool** that's available within the System.Buffers namespace. You can pass this rented array to the GetBytes method overload that accepts a span (with an implicit cast from array to span) and return it back to the pool once you're done with it. The following sample uses the default implementation of ArrayPool to get an array of the required length that will contain the encoded bytes that need to be parsed. The parsing loop remains unchanged and continues to use the Utf8Parser API, just like before.

```
int minLength = encode.GetByteCount(data);
byte[] array = pool.Rent(minLength);
Span<byte> utf8 = array;
int bytesWritten = encode.GetBytes(data, utf8);
utf8 = utf8.Slice(0, bytesWritten);

int sum = 0;
// Same parser loop as before

pool.Return(array);
return sum;

private static Encoding encode = Encoding.UTF8;
private static ArrayPool<byte> pool =
    ArrayPool<byte>.Shared;
```

Like sample 1, there's no change necessary at the call site. That's because there is an implicit cast from string to ReadOnlySpan<char> in .NET Core 2.1. If, however, you're targeting an older runtime and using portable span, the implicit cast on string isn't available. To work around that, there's an extension method on the string called AsSpan(), which returns a ReadOnlySpan<char>. This extension method, along with several others, is available in the **MemoryExtensions** class within the System.Memory assembly.

```
static void Main(string[] args)
{
    string data = "1, 2, 3, 4, 5, 6, 7";
```

```
    StringParseSum(data); // 28
    SpanParseSum(data); // 28
    SpanParseSumUsingUtf8Parser(data);  // 28
    Utf8ParserWithPooling(data); // 28

    // Only required for older runtimes
    SpanParseSum(data.AsSpan());
}
```

Let's compare the different string parsing techniques you've defined. Notice from **Figure 10** that you gain a 20% performance improvement by directly using the APIs available on span instead of string, and you get rid of all intermediary allocations. Although you end up allocating a small byte array, if you encode the incoming string data as UTF-8, by leveraging the Utf8Parser APIs, you get a ten-times reduction in execution time! Additionally, you can pool your arrays to eliminate all the allocations without noticeably impacting performance.

Are you interested in seeing other performance improvements across .NET Core 2.1? See the links in the References sidebar for more places to read about it.

### *Sample 3: Return the Last Element of the Array by Reversing It First, While Keeping the Original Intact*

If you recall, I mentioned that Span<T> can be used to represent arbitrary memory. Now that you have learned how to create a span from arrays and strings, what if you wanted to create a span around memory that's allocated on the stack to retain memory safety guarantees? You could do this by refactoring an array-based implementation that starts off allocating on the heap.

```
public static int HeapAllocReverseArray(
    int[] data)
{
    // Heap-allocated array for defensive copy.
    int[] array = new int[data.Length];
    Array.Copy(data, array, data.Length);
    Array.Reverse(array);
    return array[0];
}
```

Let's say that you know the input array is small enough to fit on the stack (assume, for now, that you already verified that the length is below a certain threshold). In that case, you can avoid the heap allocation by using memory allocated on the stack. Leveraging stack allocated memory is only a viable strategy for small, constant-sized buffers, say 128 bytes, because the default stack size per thread on Windows is one megabyte. This forces you to use unsafe code because you're now dealing with pointers.

```
public static int UnsafeStackAllocReverse(
    int[] data)
{
    unsafe
    {
        // We lose safety and bounds checks.
        int* ptr = stackalloc int[data.Length];
        // No APIs available to copy and reverse
        for (int i = 0; i < data.Length; i++)
        {
            ptr[i] = data[data.Length - i - 1];
        }
        return ptr[0];
    }
}
```

You can re-write this by wrapping a Span<T> around the stackalloc pointer (by calling the constructor that lets you create a span from unmanaged memory via a pointer). You're still in unsafe territory due to the use of pointers. However, you can now take advantage of the newly added APIs like CopyTo and Reverse to have a cleaner implementation.

```
public static int UnsafeStackAllocReverse(
    int[] data)
{
    unsafe
    {
        int* ptr = stackalloc int[data.Length];
        // Using Span ctor that takes a pointer
        var span = new Span<int>(
            ptr, data.Length);
        // Easy to use span APIs
        data.CopyTo(span);
        span.Reverse();
        return span[0];
    }
}
```

Building on top of the previous code snippet, you can go one step further, and remove all use of pointers and unsafe code. You do this by leveraging inline initialization of stackalloc span, a new language feature in C# 7.2. This way, you end up with a completely safe, span-based implementation of the sample. Additionally, you can now write the method without needing separate code paths between the stack-allocated and heap-allocated copy of the array.

```
public static int SafeStackOrHeapAllocReverse(
    int[] data)
{
    // Chose an arbitrary small constant
    Span<int> span = data.Length < 128 ?
                stackalloc int[data.Length] :
                new int[data.Length];
    data.CopyTo(span);
    span.Reverse();
    return span[0];
}
```

Due to its design as a ref-like type, span comes with a set of restrictions that are enforced by the C# compiler and the core runtime.

## Is There Anything Span Can't Do?!

There are some trade-offs to consider when determining where using span is beneficial. You should be using spans predominantly in synchronous, performance-sensitive code paths where you want to avoid excessive data


**Figure 11:** This is how you can conceptualize Memory<T>.

copies and allocations. This includes any scenario that involves substantial string manipulation or buffer management, or where you previously had to rely on writing unsafe code to get pointer-like performance in your libraries and server applications. Due to its design as a ref-like type, span comes with the following set of restrictions that are enforced by the C# compiler and the core runtime:

- Span can only live on the execution stack.
- Span cannot be boxed or put on the heap.
- Span cannot be used as a generic type argument.
- Span cannot be an instance field of a type that itself is not stack-only.
- Span cannot be used within asynchronous methods.

You can find additional background on the restrictions on ref-like types by following the link in the References sidebar.

> Unlike Span<T>, Memory<T> doesn't come with the stack-only limitations because it's not a ref-like type.

Due to limitations and complexity from using span, if your scenario involves writing user-facing and UI-heavy applications, you should continue to use the well-understood strings and arrays. Furthermore, if your scenario involves prototyping or rapid application development (RAD), where developer productivity takes precedence over application performance, using spans won't be as beneficial.

Span<T> gains certain benefits because of these restrictions. These limitations enable efficient buffer access, safe and concrete lifetime semantics that are tied to stack unwinding, and they circumvent concurrency issues like struct tearing. To support the developer scenarios that cannot be addressed by span due to its usage constraints, .NET Core 2.1 also provides another type called Memory<T>.

## Hot off the Press: Memory<T>

**Memory<T>** is another new sliceable type within the System namespace that acts as a complement to Span<T>. Just like span, it provides a safe and sliceable view into any contiguous buffer, such as arrays or strings. Unlike Span<T>, Memory<T> doesn't come with the stack-only limitations because it's not a ref-like type. Therefore, like any other C# struct, you can put it on the heap, use it within collections or with async await, store it as a field, or box it. Whenever you need to manipulate or process the underlying buffer referenced by Memory<T>, you can access the Span<T> property and get efficient indexing capabilities. This makes Memory<T> a more general purpose and high-level exchange type than Span<T>. Like span, Memory<T> also has a read-only counterpart, aptly named **ReadOnlyMemory<T>**. The original specification document for Memory<T> is available on GitHub for you to review (https://aka.ms/memory-spec).

**Note:** Going forward, to disambiguate from "memory" I'll refer to Memory<T> and ReadOnlyMemory<T> explicitly with the generic type T specified.

> You can treat Memory<T> as a factory for spans.

## Let's Take a Peek Inside Memory<T>

You can visualize Memory<T> as a struct containing three fields: an object, an index, and a length (see **Figure 11**).

The object field lets Memory<T> behave like a union type because it can represent an array/string or a **MemoryManager** (more on that later). As a performance optimization, the highest order bit of the index field is used to discern between the types of the object that the Memory<T> is wrapped around because that field can't be negative (negative indices don't have any semantic meaning). For it to be the heap-able counterpart to span, many of the Memory<T> APIs are like the ones available on span (array-based constructors, Slice overloads, CopyTo, implicit operators, etc.).

Memory<T> contains two unique APIs that are worth highlighting. The first one is the span property that I mentioned earlier, which can be passed around within the sequential sections of your application and let you index into the underlying buffer as you would with an array. The second one is the Pin method, which gives you a **MemoryHandle** to the original buffer and informs the garbage collector (GC) to not move the buffer in memory. MemoryHandle is essentially a disposable wrapper around the **GCHandle** and is only necessary for advance scenarios, such as interop with native code.

For common uses where Memory<T> represents an array or string, and you aren't dealing with manual management of the lifetime of those objects, you can simply treat Memory<T> as a factory for spans. However, given that its functionality can be extended via the MemoryManager to support different ownership and lifetime semantics, I want to provide you with some usage guidelines and best practices when dealing with Memory<T>. These guidelines apply to ReadOnlyMemory<T> as well.

- Where possible, synchronous methods should accept Span<T> arguments instead of Memory<T>, whereas asynchronous methods should accept Memory<T> arguments.
- Methods without return types (i.e., void methods) that take Memory<T> as an argument shouldn't use it after the method returns (within background threads, for instance).
- Async methods that take Memory<T> as an argument and return a task shouldn't use it after the method returns (i.e., after the method caller that's waiting on that task continues its execution).
- If you define a type that either takes in a Memory<T> in its constructor or has a settable Memory<T> property on it, that type consumes the Memory<T> instance provided to it. Each Memory<T> instance can have just one consumer at a time.

You can read the detailed Memory<T> usage guidelines by following the link in the References sidebar.

## Owners, Consumers, and Managers: Here be Dragons

You may have noticed that I previously mentioned some terminology like ownership and consumer. Let's briefly discuss the lifetime semantics of Memory<T> and some of the challenges it hopes to address. Within libraries and user applications, buffers encapsulating runtime memory occasionally get passed around between methods and need to be accessible from multiple execution threads. This leads to the issue of lifetime management of the memory and requires an understanding of three core concepts:

- **Ownership**: Every buffer has a single owner, responsible for its lifetime management and cleanup. This ownership can be transferred.
- **Consumption**: At any given time, there's a single consumer of the buffer that can read from it or write to it.
- **Leasing**: The lease is the restricted time duration, during execution, that any given component can act as a consumer of the buffer.

These are useful concepts for you to keep in mind because they help in identifying and avoiding **use-after-free** bugs in your code, which is a common class of problem when dealing with memory management. The **IMemoryOwner<T>** interface and the **MemoryManager<T>** abstract class that implements it aim to provide crisp delineation between transfer of ownership and consumption of memory so that Memory<T> can be extended to support advance developer scenarios (for example when writing Web servers). IMemoryOwner controls the lifetime of Memory<T>. You can get an instance of IMemoryOwner by renting it from a pool by calling **Rent** on an implementation of **MemoryPool<T>** and you can release it back to the pool by disposing of it. Like how you would use an ArrayPool, here's the code to write if you want to pool instances of Memory<T> and leverage IMemoryOwner to properly manage its lifetime and release.

```
// Your workhorse method
public void Worker(Memory<byte> buffer);

private static MemoryPool<byte> pool =
        MemoryPool<byte>.Shared;

public void UserCode()
{
    using (IMemoryOwner<byte> rental
            = pool.Rent(minBufferSize: 1024))
    {
        Worker(rental.Memory);
    }
    // The memory is released back to the pool
}
```

There's a lot you can do with these new types. If you're interested, there are additional code samples and details

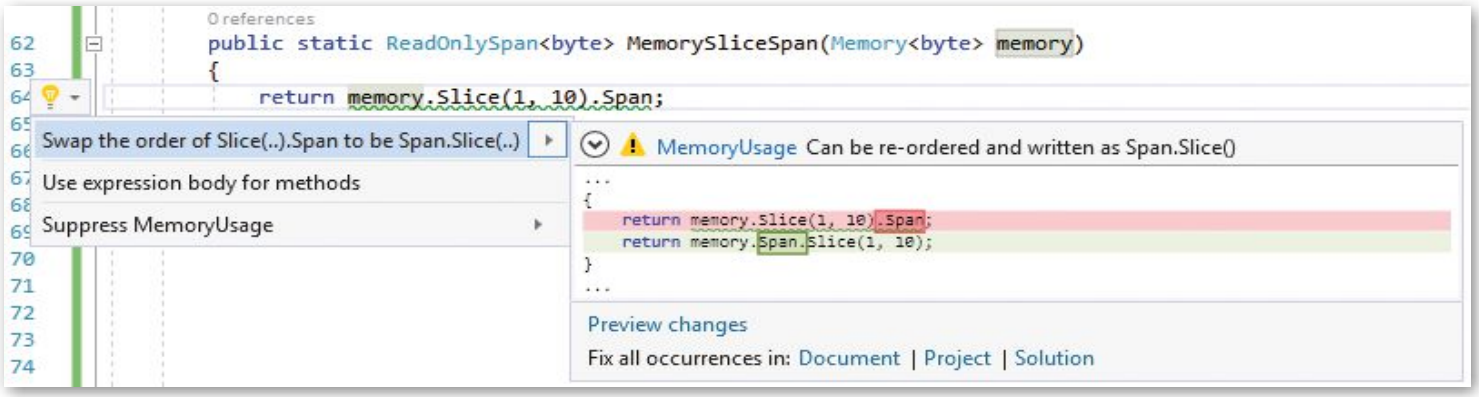



**Figure 12:** The code analyzer suggesting a fix for optimal use of Memory<T> APIs in Visual Studio.

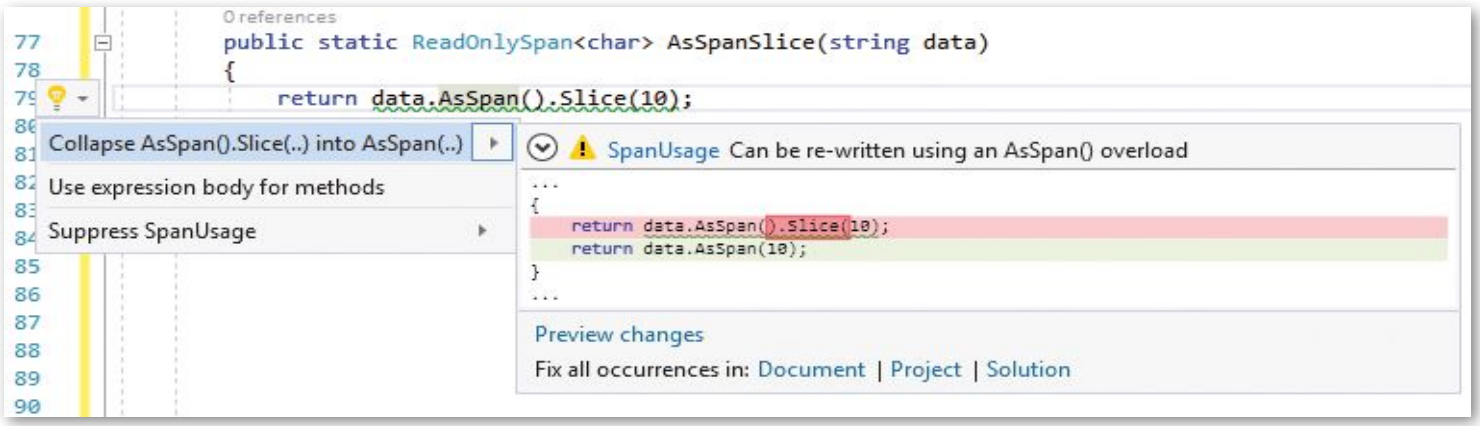



**Figure 13:** The code analyzer suggests a fix to use Span<T> APIs optimally in Visual Studio.

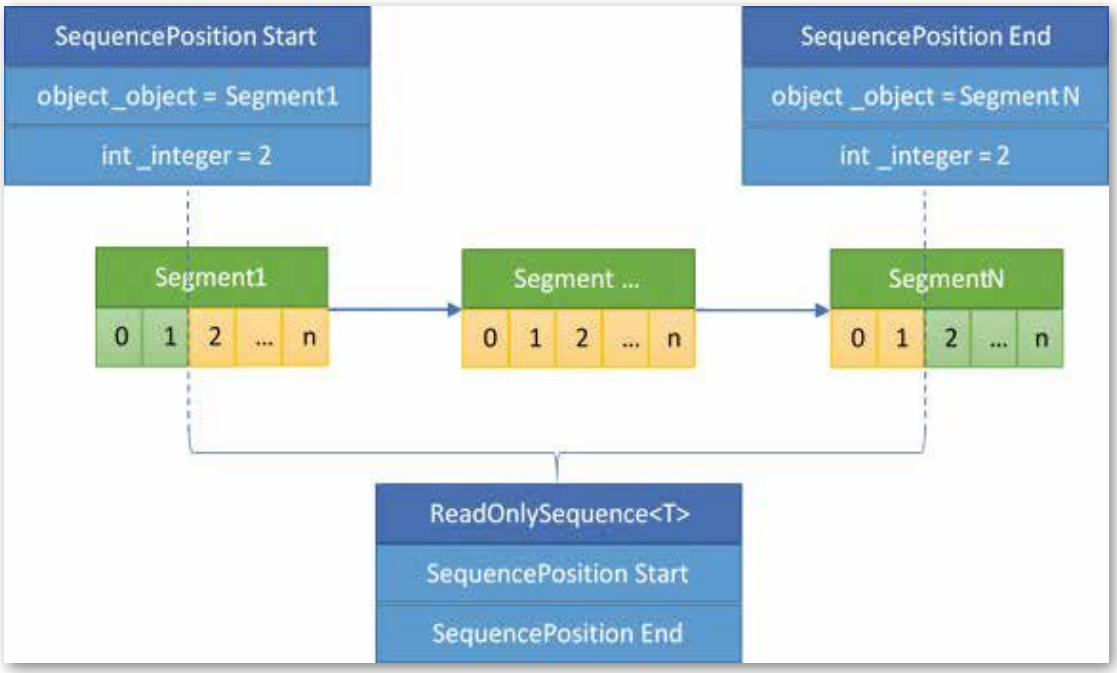



**Figure 14:** The conceptual representation of ReadOnlySequence, which consists of a start and end SequencePosition.



on replacing the implementation of Memory<T>, subclassing MemoryManager<T>, and creating your own custom MemoryPool<T> within the references sidebar.

## Tricks of the Trade

Here are some straightforward but subtle guidelines to ensure that you're using the new Span<T> and Memory<T> APIs as efficiently as possible.

### *Tip 1: Use Memory.Span.Slice(...) Instead of Memory.Slice(...).Span*

Because Memory<T> is a larger struct, slicing it tends be relatively slower compared to slicing Span<T>. Additionally, accessing the Span<T> property on Memory<T> has a non-trivial cost. Therefore, minimize accessing the span and cache it in a local variable. Try to pass the span to the sequential, synchronous methods rather than to Memory<T> directly, especially where you tend to slice the data in a loop. You can see the offending code and suggested fix in **Figure 12.** Slicing the Memory<T> can still be useful for scenarios where you want to persist sections of the data on the heap, which can't be done with spans (for instance, if you need to store it in a collection or pass it to an asynchronous method).

### *Tip 2: Use AsSpan(...) Instead of AsSpan().Slice(...)*

You may prefer the readability of the fluent interface coding style and chain method calls together. One such scenario where you might be inclined to use the fluent pattern is if you have an array or string and want to get a subsection as a span. To do so, you could call AsSpan() and then slice the span based on the start index and length. It's slightly faster if you call the AsSpan() overload that takes the start index and length as parameters, which essentially does the cast and slice in one step. You can see the offending code and suggested fix in **Figure 13**.

**Figure 12** and **Figure 13** show the MemoryUsage and SpanUsage code analyzers in action with code fix suggestions within Visual Studio. You can install and contribute to these code analyzers and help catch instances of the sub-optimal usage patterns described above (see the References sidebar for more on this).

## What About Discontiguous Buffers?

Both Span<T> and Memory<T> provide functionality for contiguous buffers such as arrays and strings. System.Memory contains a new sliceable type called **ReadOnlySequence<T>** within the System.Buffers namespace that offers support for discontiguous buffers represented by a linked list of ReadOnlyMemory<T> nodes. I won't go into too much detail here, but the basic structure of a ReadOnlySequence is based on two *cursors* within the sequence, known as **SequencePosition** (presented in **Figure 14**).

The following code snippet shows us how you can create a single segment ReadOnlySequence directly from an array or ReadOnlyMemory. Furthermore, just like Span<T> and Memory<T>, you can slice ReadOnlySequence<T> without having to deal with stitching together the fragmented buffers yourself.

```
int[] array = { 1, 2, 3, 4, 5 };
var sequence = new ReadOnlySequence<int>(array);
sequence = sequence.Slice(1, 3);
// sequence = { 2, 3, 4 }

ReadOnlyMemory<int> memory = array;
sequence = new ReadOnlySequence<int>(memory);
sequence = sequence.Slice(1, 3);
// sequence = { 2, 3, 4 }
```

To use ReadOnlySequence as a linked list of buffers, you must subclass and provide a concrete implementa-

tion of the abstract **ReadOnlySequenceSegment<T>** class, which conceptually represents a node within the sequence. You can then pass the start and end segment to the appropriate ReadOnlySequence constructor. The ReadOnlySequenceSegment class is defined as follows:

```
public abstract class ReadOnlySequenceSegment<T>
{
    // The value of the current node.
    public ReadOnlyMemory<T> Memory
            { get; protected set; }

    // The next node within the sequence.
    public ReadOnlySequenceSegment<T> Next
            { get; protected set; }

    // The sum of node lengths before current.
    public long RunningIndex
            { get; protected set; }
}
```

Let's assume that you have the following concrete implementation of ReadOnlySequenceSegment.

```
class Segment<T> : ReadOnlySequenceSegment<T>
{
    public Segment(ReadOnlyMemory<T> memory)
        => Memory = memory;

    public Segment<T> Add(ReadOnlyMemory<T> mem)
    {
        var segment = new Segment<T>(mem);
        segment.RunningIndex = RunningIndex +
                    Memory.Length;

        Next = segment;
        return segment;
    }
}
```

You can then create a ReadOnlySequence from instances of Segment<T> to represent disjointed buffers and slice them just like single segment sequences.

```
int[] array1 = { 1, 2, 3 };
int[] array2 = { 4, 5, 6 };
var segment1 = new Segment<int>(array1);
Segment<int> segment2 = segment1.Add(array2);

sequence = new ReadOnlySequence<int>(
    segment1, startIndex: 0,
    segment2, endIndex: 3);

sequence = sequence.Slice(1, 3);
// sequence = { 2, 3, 4 }
```

Because ReadOnlySequence provides APIs that obfuscate the underlying structure of how the segments compose the sequence, you can view it as a single continuous buffer to build higher-level stream-like APIs. The PipeReader and PipeWriter classes within the System.IO.Pipelines namespace that are used by the ASP.NET Core Web server are an example of such APIs (the references sidebar has the link to its original specification document).

## Summary

If you're working on making your server applications more efficient, developing highly scalable APIs, or even if you just like to keep up with what's on the cutting edge of .NET, I encourage you to give Span<T> and Memory<T> a try. These sliceable types were designed with a focus on runtime and memory efficiency for cloud-based scenarios and adhere to the philosophy of no allocations and no data copies. I hope you are eager to try out these new .NET Core 2.1 features within your libraries and applications and take advantage of all the performance optimizations that accompanied this release. Better yet, I welcome you to join in on the fun as part of the .NET open source community on GitHub to help build the next set of features and libraries for everyone to use.

You can download all the code samples shown in this article from the CODE Magazine link associated with this article. Run them yourself!

Ahson A. Khan

CODE

## Useful References

Span<T> spec document:
https://aka.ms/span-spec

System.Memory NuGet package containing Span<T>, Memory<T>, and friends:
https://aka.ms/nuget-package

.NET Standard specification:
https://aka.ms/net-standard

Performance Improvements in .NET Core 2.1:
https://aka.ms/core21-perf

BenchmarkDotNet:
https://benchmarkdotnet.org

Compile time enforcement of safety for ref-like types:
https://aka.ms/span-safety

Memory<T> spec document:
https://aka.ms/memory-spec

Memory<T> usage guidelines:
https://aka.ms/memory-guidelines

Memory<T> API documentation and samples: https://aka.ms/memory-docs

Basic Span<T>/Memory<T> code analyzer:
https://aka.ms/span-analyzer

Pipelines spec document:
https://aka.ms/pipelines-spec

# Build Real-time Applications with ASP.NET Core SignalR

A lot is demanded from Web applications today. Web apps are expected not only to function properly, but they need to do so with a great user experience. Users expect applications to be fast. They also expect applications to deliver information in real-time, without the need to refresh the browser. All major programming platforms have frameworks that make it easier to build real-time applications. The most well-known real-time framework is probably Socket.IO on Node.js. ASP.NET has a popular real-time framework called SignalR.

In this article, you'll learn about the origins of SignalR and how it has been rewritten to run on ASP.NET Core and address the needs of today's real-time applications. You'll also learn how to get started building ASP.NET Core SignalR applications.

**Anthony Chu**
@nthonyChu

Anthony Chu is a Cloud Developer Advocate at Microsoft. He's been a software developer for over 15 years and builds applications with ASP.NET and Node.js. He specializes in serverless and container technologies, including Azure Functions, Kubernetes, and Service Fabric. Anthony is based in Vancouver, BC, Canada, and you can find him at technical conferences around the world or follow him on Twitter at @nthonyChu.

## The History of SignalR

SignalR was created in 2011 by David Fowler and Damian Edwards, who now play key roles in the direction and development of ASP.NET. It was brought into the ASP.NET project and released as part of ASP.NET in 2013. At the time, the WebSocket protocol had just been standardized and most browsers didn't support it.

To achieve real-time messaging for the Web, developers used inefficient techniques such as AJAX polling and long-polling, and technologies like server-sent events that weren't broadly implemented by browsers. SignalR was set up to solve this problem and provide easy support for real-time capabilities on the ASP.NET stack by creating server- and client-side libraries that abstract away the complications of these technologies.

Behind the scenes, SignalR negotiates the best protocol to use for a specified connection based on what's supported by both the server and the client. It then provides a consistent API for sending and receiving messages in real-time. Because it's so easy to use, ASP.NET developers quickly adopted SignalR, making it the de facto stack for real-time ASP.NET development.

A lot can change on the Web in five years. When SignalR was created, almost every Web application used jQuery, so it was an easy decision for the SignalR JavaScript client to depend on jQuery as well. SignalR also contained complex logic focused on managing the different protocols and workarounds to achieve real-time messaging on the Web before the WebSocket protocol was widely adopted by browsers.

Fast forward to 2018. A lot of jQuery's functionality can be achieved with plain JavaScript, and full-fletched front-end frameworks such as Angular, React, and Vue have emerged to replace jQuery as the new foundations on which single-page applications (SPAs) are built. WebSockets are available in all major browsers and are now the standard for real-time Web communications.

Other features in SignalR were intended to make it easier to use, such as automatic reconnection and turnkey scale-out, but they ended up adding complexity and inefficiencies to the framework.

So when ASP.NET was reimagined from the ground up to create a faster, cross-platform ASP.NET Core, the team also took the time to rewrite SignalR from scratch, taking into account all that was learned from the first two versions of SignalR, as well as making it extensible enough to future-proof against new protocols and transport technologies that may emerge.

## Introducing ASP.NET Core SignalR

SignalR on .NET Core runs on ASP.NET Core 2.1, which can be downloaded at http://aka.ms/DotNetCore21.

Overall, ASP.NET Core SignalR maintains a lot of the same core concepts and capabilities as SignalR. Hubs continue to be the main connection point between the server and its clients. Clients can invoke methods on the hub, and the hub can invoke methods on the clients. The hub has control over the connections on which to invoke a certain method. For example, it can send a message to a single connection, to all connections belonging to a single user, or to connections that have been placed in arbitrary groups.

There were several noteworthy changes between ASP.NET SignalR and ASP.NET Core SignalR; let's go over some of them right now.

### *JavaScript Client Library*

In the browser, the biggest change is the removal of the jQuery dependency. The JavaScript/TypeScript client library can now be used without referencing jQuery, allowing it to be used with frameworks such as Angular, React, and Vue without friction. In addition, this allows the client to be used in a Node.js application.

To align with the expectations of the front-end development community, the JavaScript client is now acquired through npm. It's also hosted on Content Delivery Networks (CDNs).

In addition to the JavaScript/TypeScript client library, ASP.NET Core SignalR also ships with a .NET client NuGet package. SignalR had clients for other languages such as Java, Python, Go, and PHP, all created by Microsoft and the open source community; you can expect the same for ASP.NET Core SignalR as its adoption increases.

### *Built-in and Custom Protocols*

ASP.NET Core SignalR ships with a new JSON message protocol that's incompatible with earlier versions of SignalR. In addition, it has a second built-in protocol based on MessagePack, which is a binary protocol that has smaller payloads than the text-based JSON.

If you want to implement a custom message protocol, ASP.NET Core SignalR has extensibility points that allow new protocols to be plugged in.

### *Dependency Injection*

ASP.NET didn't have dependency injection built in, so SignalR provided a GlobalHost class that included its own dependency resolver. Now that ASP.NET Core ships with an inversion of control (IoC) container, ASP.NET Core SignalR simply leverages the built-in framework for dependency injection.

Hubs in ASP.NET Core SignalR now support constructor dependency injection without extra configuration, just like ASP.NET Core controllers or razor pages do. It's also easy to gain access to a hub's context from outside the hub itself by retrieving an IHubContext<T> from the IoC container and using its methods to send messages to the hub's clients.

### *Scale-out*

SignalR shipped with built-in support for scale-out using Redis, Service Bus, or SQL Server as a backplane. A backplane allows different instances of the same ASP.NET SignalR application to communicate with one another to broadcast messages to the correct clients, regardless of which instance the clients are connected to. This proved to be difficult to implement correctly and added a lot of overhead; it also didn't consider that different applications have different scale-out needs. The result was a scale-out functionality that was complex, inefficient, and didn't work well for many scenarios.

ASP.NET Core SignalR was redesigned with a simpler and more extensible scale-out model. It no longer allows a single client to connect to different server-side instances between requests. This means that sticky sessions are required to ensure server affinity for clients using protocols other than WebSockets. ASP.NET Core SignalR currently provides a scale-out plug-in for Redis.

Later in this article, you'll also learn about a new, fully managed Azure SignalR Service that allows you to massively scale out your ASP.NET Core SignalR applications with only minor code changes. Azure SignalR Service also enables non-.NET and serverless applications to provide real-time messaging to SignalR compatible clients.

### *Reconnections*

Another design decision that seemed like a good idea when SignalR first came out was automatic reconnections. SignalR included reconnection logic on both the clients and the server. Clients attempted to reconnect if a connection was lost, and the server buffered unsent messages and replayed them when a client reconnected. This also proved to be buggy and inefficient, and the implementation didn't make sense for all applications.

ASP.NET Core SignalR doesn't support automatic reconnection or automatic buffering of messages. Instead, it's up to the client application to decide when it needs to reconnect; and it's up to the server to implement message buffering if required.

## Get Started with ASP.NET Core SignalR

Let's create a simple chat application to demonstrate how to use ASP.NET Core SignalR.

Like typical ASP.NET Core development, you can use the **dotnet** command line interface (.NET Core SDK 2.0 and later) and an editor such as Visual Studio Code, Visual Studio 2017, or Visual Studio for Mac to build ASP.NET Core SignalR applications.

### *Create the Initial Application*

This article builds on a new ASP.NET Core Razor Pages application with individual authentication. You can create one in Visual Studio's new ASP.NET Core project dialog or run the following .NET CLI command:

```
dotnet new razor --auth Individual
```

A new ASP.NET Core Razor Pages application with individual authentication is created. By default, users are stored in a SQLite database.

To use ASP.NET Core SignalR, it must be added to the project from NuGet. The latest version at the time of writing this is RC1.

```
dotnet add package Microsoft.AspNetCore.SignalR
  --version 1.0.0-rc1-final
```

### *Add a SignalR Hub*

A hub is the central point in an ASP.NET Core application through which all SignalR communication is routed. Create a hub for your chat application by adding a class named Chat that inherits from Microsoft.AspNetCore.SignalR.Hub:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
  public class Chat : Hub
  {
    public async Task SendMessage(
      string message)
    {
      await Clients.All.SendAsync(
        "newMessage", "anonymous", message);
    }
  }
}
```

The SendMessage method is invoked by the clients whenever a message needs to be sent. It uses the **Client.All** property of the hub to invoke a method named newMessage on all connected clients with arguments for the sender's username (currently "anonymous") and the message. You'll implement the client-side SignalR code a bit later.

For the hub to function, SignalR needs to be enabled in the application. To do this, make a few changes to Startup.cs.

### Inversion of Control

Inversion of control (IoC) is a powerful design principle that decouples an application from its dependencies. A common way to achieve IoC is by using a technique called dependency injection. Instead of creating its own dependencies, a class declares its dependencies in their constructors. At runtime, concrete instances of the class' dependencies are "injected" into its constructor during instantiation. Frameworks such as ASP.NET Core provide integrated IoC containers that automatically manage the instantiation and lifetimes of objects and their dependencies.

In ConfigureServices, call the AddSignalR extension method to configure the IoC container with services required by SignalR. Like this:

```
public void ConfigureServices(
  IServiceCollection services)
{
  // ...

  services.AddSignalR();
}
```

And in Configure, add a call to UseSignalR to map a route (/chat) for the Chat hub.

```
public void Configure(
  IApplicationBuilder app,
  HostingEnvironment env)
{
  // ...

  app.UseAuthentication();

  app.UseMvc();

  app.UseSignalR(builder =>
  {
    builder.MapHub<Chat>("/chat");
  });
}
```

Now it's time to add some client-side code that will interact with the Chat hub. In Pages/Index.cshtml, reference the SignalR browser JavaScript library. This can be done by using npm and a tool like WebPack to install the package and copy the client-side JavaScript files to the ww-wroot folder. You can also reference the script on a CDN (note that the original URL has single @ signs, but @ is a special character in Razor and is escaped with @@):

```
<script src="https://unpkg.com/@@aspnet/signalr@@
1.0.0-rc1-final/dist/browser/signalr.js">
</script>
```

Also create a simple textbox for messages and an unordered list to display chat messages.

```
<div class="signalr-demo">
  <form id="message-form">
    <input type="text" id="message-box" />
  </form>
  <hr/>
  <ul id="messages"></ul>
</div>
```




**Figure 1:** Run the SignalR chat application.

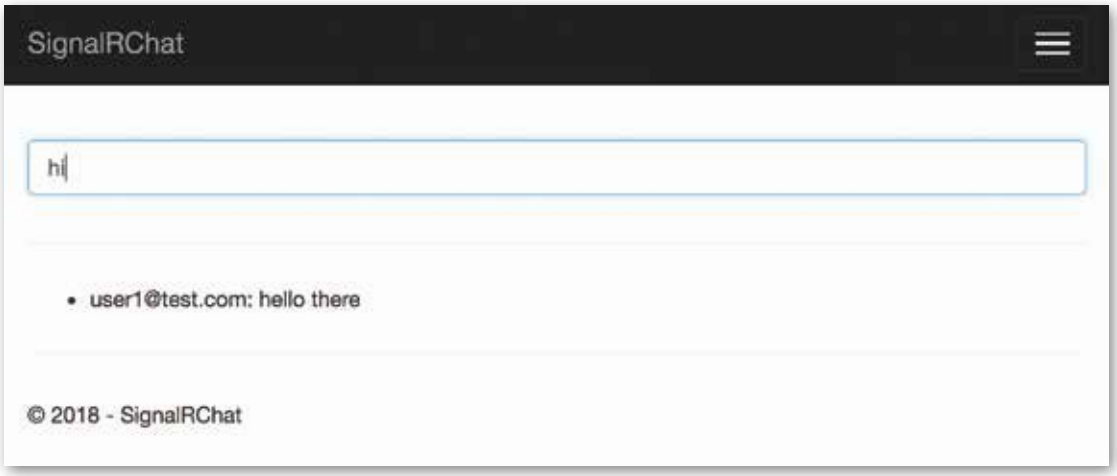



**Figure 2:** The authenticated username appears in the chat.

The final step is to add some JavaScript to build and start a HubConnection (see **Listing 1**). Add a function to execute when newMessage is invoked. Also add some code to invoke SendMessage on the server to send a new chat message.

> A common way to include a client's identity on AJAX requests is via bearer tokens in an Authorization header.

If you run the application and open it up on two or more browsers, the simple chat application should be functional, as shown in **Figure 1**.

## Add Authentication and Authorization

SignalR authentication and authorization use the same claims-based identity infrastructure provided by ASP.NET Core. Just like authorization in ASP.NET Core, you can use the AuthorizeAttribute to require authorization to access a SignalR hub or a SignalR hub's methods. Also modify the SendAsync call to use the logged in username.

```
[Authorize]
public class Chat : Hub
{
  public async Task SendMessage(string message)
  {
    await Clients.All.SendAsync(
      "newMessage", Context.User.Identity.Name,
      message);
  }
}
```

By default, ASP.NET Core identity uses cookies for authentication. When a Web client connects to a SignalR hub, any existing authentication cookies are sent in the request headers. To access a hub that has authorization enabled, you'll need to log into your application before connecting to it. Now you'll see the authenticated user's username, as shown in **Figure 2.**

**Listing 1:** Build and start a HubConnection with JavaScript

```
<script>
  const messageForm =
    document.getElementById('message-form');
  const messageBox = document.getElementById('message-box');
  const messages = document.getElementById('messages');

  const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chat")
    .configureLogging(signalR.LogLevel.Information)
    .build();
  connection.on('newMessage', (sender, messageText) => {
  console.log(`${sender}: ${messageText}`);
  const newMessage = document.createElement('li');
  newMessage.appendChild(
    document.createTextNode(`${sender}: ${messageText}`));
    messages.appendChild(newMessage);
  });
  connection.start()
    .then(() => console.log('connected!'))
    .catch(console.error);

  messageForm.addEventListener('submit', ev => {
    ev.preventDefault();
    const message = messageBox.value;
    connection.invoke('SendMessage', message);
    messageBox.value = '';
  });
</script>
```

**Listing 2:** Accept the token from the query string

```
var key = new SymmetricSecurityKey(
  System.Text.Encoding.ASCII.GetBytes(
    Configuration["JwtKey"]));
services.AddAuthentication()
  .AddJwtBearer(options =>
  {
    options.TokenValidationParameters =
    new TokenValidationParameters
    {
      LifetimeValidator =
        (before, expires, token, parameters) =>
          expires > DateTime.UtcNow,
      ValidateAudience = false,
      ValidateIssuer = false,
      ValidateActor = false,
      ValidateLifetime = true,
      IssuerSigningKey = key,
      NameClaimType = ClaimTypes.NameIdentifier
    };

    options.Events = new JwtBearerEvents
    {
      OnMessageReceived = context =>
      {
        var accessToken =
          context.Request.Query["access_token"];

        if (!string.IsNullOrEmpty(accessToken))
        {
          context.Token = accessToken;
        }
        return Task.CompletedTask;
      }
    };
  });

services.AddAuthorization(options =>
{
  options.AddPolicy(JwtBearerDefaults.AuthenticationScheme,
    policy =>
    {
    policy.AddAuthenticationSchemes(
      JwtBearerDefaults.AuthenticationScheme);
    policy.RequireClaim(ClaimTypes.NameIdentifier);
  });
});
```

# Build your world with .NET

## Any app, any platform

- Web
- Cloud
- Desktop
- Gaming
- Mobile
- Artificial Intelligence/ Machine Learning (AI/ML)
- IoT (Internet of Things)

Azure

## Scalable application architectures

- Microservices
- Cloud Design Patterns
- Serverless

kubernetes

## Deployment to any point

**Containerization and Hosting**
- App Service, DockerHub, Kubernetes, AKS

**Application Monitoring**
- App Insights

Visual Studio

nuget

## Tools to get it done

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- Visual Studio App Center
- Visual Studio Team Services
- .NET CLI (command-line-interface)

Windows

Linux

android

iOS

macOS

## Components to get you started

- .NET Standard Libraries
- NuGet
- Identity and Security
- Data

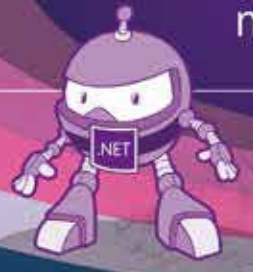
## A solid core

**Multi-language**
- C#
- F#
- Visual Basic

**Cross-platform**
- Windows (Desktop/Server/Nano)
- macOS
- Red Hat Enterprise Linux (RHEL)
- Debian
- Ubuntu
- Alpine

**Performance and Reliability**

**Open Source**



## Any app, any platform

**Web:** Create scalable, high-performance websites and services that run on Windows, macOS, and Linux. https://aka.ms/startweb

**Cloud:** Create powerful, intelligent cloud apps with .NET using a fully managed platform. https://aka.ms/startcloud

**Desktop:** Create beautiful and compelling native desktop apps on Windows and macOS. https://aka.ms/startdesktop

**Mobile:** Use a single code base to build native mobile apps for iOS, Android, and Windows. https://aka.ms/startmobile

**Gaming:** Develop 2D and 3D games for the most popular desktops, phones, and consoles. https://aka.ms/startgaming

**AI/ML (Artificial Intelligence and Machine Learning):** Infuse AI and Machine Learning into your .NET apps such as vision algorithms, speech processing, predictive models, and more. https://aka.ms/startai

**IoT (Internet of Things):** Make IoT apps with native support for the Raspberry Pi and other single-board computers. https://aka.ms/startiothings

## Scalable application architectures

**Microservices:** Microservices are highly scalable, resilient, and composable units of deployment for modern applications and .NET is a perfect platform for creating them. Get started with modern .NET application architectures. https://aka.ms/startarch

**Cloud Design Patterns:** Learn about the essential design patterns that are useful for building reliable, scalable, secure applications in the cloud. https://aka.ms/startpatterns

**Serverless:** Serverless computing is the abstraction of servers, infrastructure, and operating systems. When you build serverless apps, you can take your mind off infrastructure concerns. https://aka.ms/startserverless

## Deployment to any point

**Containerization and Hosting:** Containers simplify deployment and testing by bundling a service and its dependencies into a single unit, which is then run in an isolated environment. Orchestrators such as Kubernetes automate deployment, scaling, and management of containerized applications.

- Docker container images for .NET on Linux and Windows are available on Docker Hub. https://aka.ms/startdocker
- Simplify the deployment, management, and operations of Kubernetes with AKS. https://aka.ms/startaks
- Quickly create powerful cloud apps with or without containers using a fully managed platform with App Service. https://aka.ms/startappservice

**Application Monitoring:** Get actionable insights through application performance management and instant analytics with App Insights. https://aka.ms/startappinsights

## Tools to get it done

**Visual Studio Family:** Visual Studio provides best-in-class tools for any developer on any operating system. From advanced IDEs and editors to agile tools, CI/CD, monitoring, and learning, they have you covered. Get started for free. www.visualstudio.com

**.NET CLI:** The .NET CLI (command-line-interface) is a command line tool you can use with any editor to build many types of .NET apps. Get it with the .NET Core SDK. www.dot.net/core

## Components to get you started

**.NET Standard Libraries:** .NET Standard allows sharing code and binaries between all .NET apps. It is an API specification that is implemented by all the flavors of .NET, making it simple to share libraries across the entire platform. https://aka.ms/netstandardapis

**NuGet:** From data components to UI controls and thousands more reusable libraries, the NuGet package manager helps you create .NET apps faster. www.nuget.org

## A solid core

**Multi-language:** You can write your .NET apps in multiple programming languages.

- **C# (c-sharp)** is a simple, modern, object-oriented, and type-safe programming language with roots in the C family of languages, making it immediately familiar to C, C++, Java, and JavaScript programmers. https://aka.ms/start-csharp
- **F# (f-sharp)** is a functional programming language that also includes object-oriented and imperative programming. https://aka.ms/start-fsharp
- **Visual Basic** is an approachable language with a simple syntax for building type-safe, object-oriented programs. https://aka.ms/start-vb

**Cross-platform:** Your .NET apps will run on a variety of operating systems, depending on the app you're building. For instance, web apps can be hosted on Windows, macOS, or multiple distros of Linux. Or build mobile apps for Android and iOS all with .NET.

**Performance:** .NET is fast. Really fast! .NET performs faster than any other popular framework on TechEmpower benchmarks. From providing safer, faster memory access with Span<T> to a faster just-in-time compiler, great performance is at the core of .NET. https://aka.ms/dotnetperf

**Open Source:** .NET is open source under the .NET Foundation.
The .NET Foundation is an independent organization to foster open development and collaboration around the .NET ecosystem.
www.dotnetfoundation.org

.NET foundation

### *Add JSON Web Token Security*

For ASP.NET Core SignalR to support authentication for non-browser-based clients, you need to implement an alternative authentication mechanism that doesn't rely on cookies.

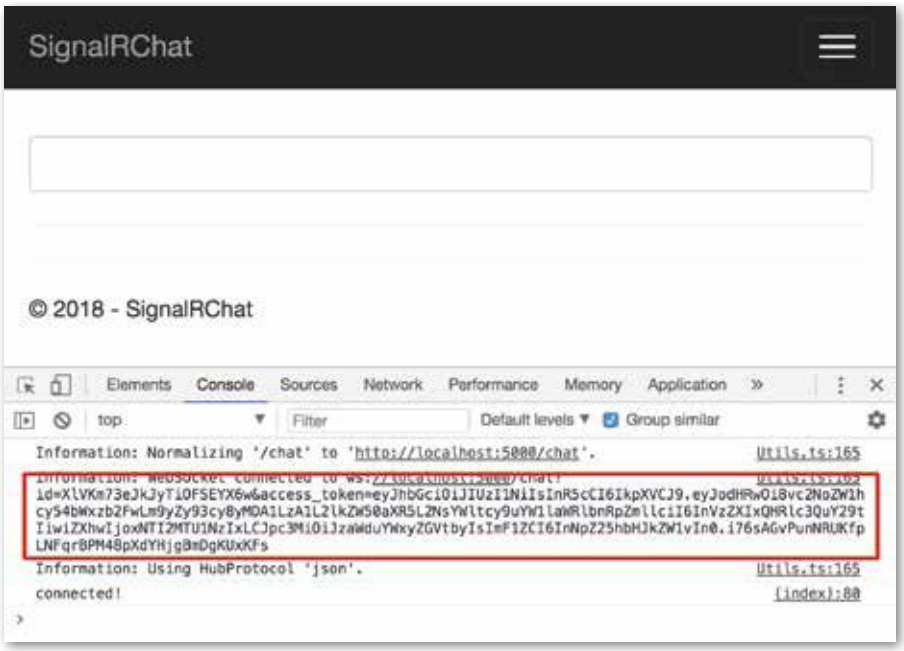



**Figure 3:** The endpoint with the token is used to connect to the SignalR hub

A common way to include a client's identity on AJAX requests is via bearer tokens in an Authorization header. Unfortunately, you cannot set the Authorization header on WebSocket requests using JavaScript in the browser. To get around this limitation, the SignalR client library supports passing the token in a query string value named **access_token**.

To accept the token from the query string, configure ASP.NET Core's authentication middleware in the ConfigureServices method in Startup.cs to set the user identity on the request using a JSON Web token (JWT) if it's available in the query string (**Listing 2**).

Next, create an ASP.NET Core controller named TokenController and add an action to exchange an identity cookie for a token, as shown in **Listing 3.**

Next, change the AuthorizeAttribute on the hub to use the JWT bearer authentication scheme. Cookie authentication will no longer work on the hub; from now on, you need to supply a valid JWT when connecting to the hub.

```
[Authorize(AuthenticationSchemes =
  JwtBearerDefaults.AuthenticationScheme)]
public class Chat : Hub
{
  // ...
}
```

Last, change the client-side JavaScript to request a token from this endpoint and use it to connect to the SignalR

**Listing 3:** Add an action to exchange an identity cookie for a token

```
add an action to exchange an identity cookie for a token
using System.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using SignalRChat.Data;

namespace SignalRChatDemo.Controllers
{
  public class TokenController : Controller
  {
    private readonly
      SignInManager<ApplicationUser> signInManager;
    private readonly IConfiguration config;

    public TokenController(
      SignInManager<ApplicationUser> signInManager,
      IConfiguration config)
    {
      this.signInManager = signInManager;
      this.config = config;
    }

    [HttpGet("api/token")]
    [Authorize]
    public IActionResult GetToken()
    {
      return Ok(GenerateToken(User.Identity.Name));
    }

    private string GenerateToken(string userId)
    {
      var key = new SymmetricSecurityKey(
        System.Text.Encoding.ASCII.GetBytes(
          config["JwtKey"]));

      var claims = new[]
      {
        new Claim(ClaimTypes.NameIdentifier, userId)
      };
      var credentials =
        new SigningCredentials(
          key, SecurityAlgorithms.HmacSha256);
      var token = new JwtSecurityToken(
        "signalrdemo", "signalrdemo", claims,
        expires: DateTime.UtcNow.AddDays(1),
        signingCredentials: credentials);

      return new JwtSecurityTokenHandler().WriteToken(token);
    }
  }

  public class LoginRequest
  {
    public string Username { get; set; }
    public string Password { get; set; }
  }
}
```

**Listing 4:** A console application that uses the library to connect

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http.Connections;
using Microsoft.AspNetCore.SignalR.Client;
using Newtonsoft.Json;

namespace SignalRChatClient
{
  class Program
  {
    static readonly HttpClient httpClient = new HttpClient();
    static readonly string baseUrl = "http://localhost:5000";
    static async Task Main(string[] args)
    {
      Console.Write("Username: ");
      var username = Console.ReadLine();

      Console.Write("Password: ");
      var password = "";
      while(true)
      {
        var key = Console.ReadKey(intercept: true);
        if (key.Key == ConsoleKey.Enter) break;
        password += key.KeyChar;
      }

      var hubConnection = new HubConnectionBuilder()
        .WithUrl($"{baseUrl}/chat", options =>
        {
          options.AccessTokenProvider = async () =>
          {
            var stringData = JsonConvert.SerializeObject(new
            {
              username, password
            });
            var content = new StringContent(stringData);
            content.Headers.ContentType =
              new MediaTypeHeaderValue("application/json");
            var response = await
              httpClient.PostAsync(
                $"{baseUrl}/api/token", content);
            response.EnsureSuccessStatusCode();
            return await
              response.Content.ReadAsStringAsync();
          };
        })
        .Build();

      hubConnection.On<string, string>("newMessage",
        (sender, message) =>
          Console.WriteLine($"{sender}: {message}"));

      await hubConnection.StartAsync();

      System.Console.WriteLine("\nConnected!");

      while(true)
      {
        var message = Console.ReadLine();
        await hubConnection.SendAsync(SendMessage", message);
      }
    }
  }
}
```

hub. The HubConnection can be configured with an access token factory to include a token when creating the connection.

```
const options = {
  accessTokenFactory: getToken
};

const connection = new signalR.HubConnectionBuilder()
  .withUrl("/chat", options)
  .configureLogging(signalR.LogLevel.Information)
  .build();

// ...

function getToken() {
  const xhr = new XMLHttpRequest();
  return new Promise((resolve, reject) => {
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      if (this.status == 200) {
        resolve(this.responseText);
      } else {
        reject(this.statusText);
      }
    };
    xhr.open("GET", "/api/token");
    xhr.send();
  });
}
```


**Figure 4:** Running the console SignalR chat app

The application continues to work, but if you inspect the requests on the network, you'll see that it's requesting a token and appending it to the SignalR hub negotiation and subsequent WebSocket connection requests (**Figure 3**).

> SignalR makes it easy to add presence to an application.

That works great for browsers, but what about clients that don't work well with cookies? You can add another endpoint on the ASP.NET Core application for non-Web clients to exchange a user's valid username and password for a token. Add the following action to the TokenCon-

**Listing 5:** The tracker class implementation

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SignalRChat
{
  public class PresenceTracker
  {
    private static readonly Dictionary<string, int>
      onlineUsers = new Dictionary<string, int>();

    public Task<ConnectionOpenedResult>
      ConnectionOpened(string userId)
    {
      var joined = false;
      lock(onlineUsers)
      {
        if (onlineUsers.ContainsKey(userId))
        {
          onlineUsers[userId] += 1;
        }
        else
        {
          onlineUsers.Add(userId, 1);
          joined = true;
        }
      }
      return Task.FromResult(
        new ConnectionOpenedResult { UserJoined = joined });
    }

    public Task<ConnectionClosedResult>
      ConnectionClosed(string userId)
    {
      var left = false;
      lock(onlineUsers)
      {
        if (onlineUsers.ContainsKey(userId))
        {
          onlineUsers[userId] -= 1;
          if (onlineUsers[userId] <= 0)
          {
            onlineUsers.Remove(userId);
            left = true;
          }
        }
      }
      return Task.FromResult(
        new ConnectionClosedResult { UserLeft = left });
    }

    public Task<string[]> GetOnlineUsers()
    {
      lock(onlineUsers)
      {
        return Task.FromResult(onlineUsers.Keys.ToArray());
      }
    }
  }

  public class ConnectionOpenedResult
  {
    public bool UserJoined { get; set; }
  }

  public class ConnectionClosedResult
  {
    public bool UserLeft { get; set; }
  }
}
```

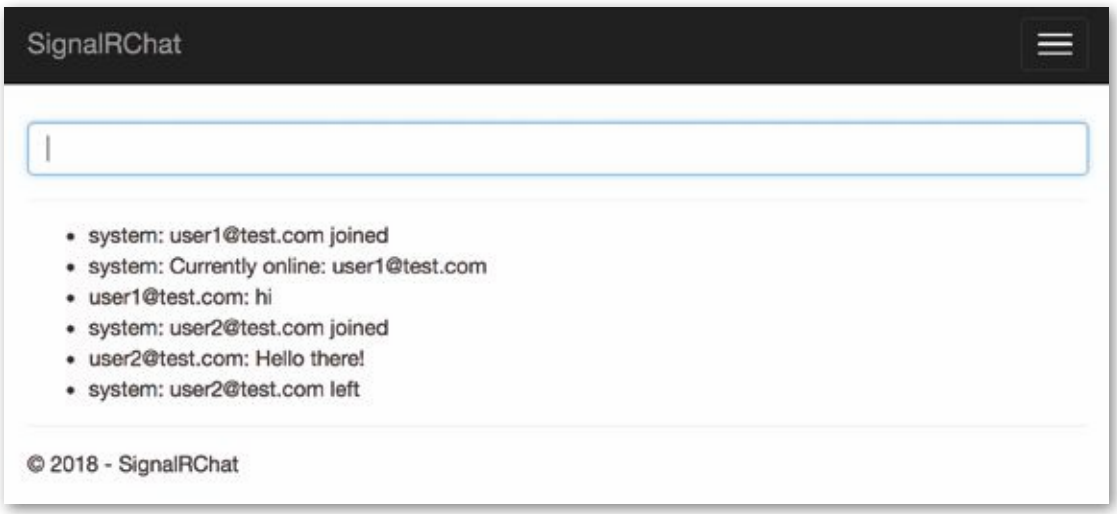



**Figure 5:** System messages when users join and leave

troller in the ASP.NET Core application to validate the credentials and return a JWT.

```
[HttpPost("api/token")]
public async Task<IActionResult>
  GetTokenForCredentialsAsync(
    [FromBody] LoginRequest login)
{
  var result = await
    signInManager.PasswordSignInAsync(
      login.Username, login.Password,
      false, true);
  return result.Succeeded ?
    (IActionResult)Ok(
```

```
      GenerateToken(login.Username)) :
    Unauthorized();
}
```

## A .NET Standard 2.0 Client for SignalR

So far, you've seen how to use ASP.NET Core with the JavaScript SignalR client library in the browser. SignalR also has a .NET Standard 2.0 client library that can be used to connect to a SignalR hub from applications built on .NET Core, .NET Framework, and more.

To use the SignalR client library, import the Microsoft.AspNetCore.SignalR.Client package from NuGet.

```
dotnet add package
  Microsoft.AspNetCore.SignalR.Client
  --version 1.0.0-rc1-final
```

**Listing 6:** The tracker shows who's come and gone

```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
public class Chat : Hub
{
  private readonly PresenceTracker presenceTracker;

  public Chat(PresenceTracker presenceTracker)
  {
    this.presenceTracker = presenceTracker;
  }

  public override async Task OnConnectedAsync()
  {
    var result = await presenceTracker.ConnectionOpened(
      Context.User.Identity.Name);
    if (result.UserJoined)
    {
      await Clients.All.SendAsync("newMessage", "system",
        $"{Context.User.Identity.Name} joined");
    }

    var currentUsers = await
      presenceTracker.GetOnlineUsers();
    await Clients.Caller.SendAsync("newMessage", "system",
      $"Currently online:\n{string.Join("\n", currentUsers)}"
    );

    await base.OnConnectedAsync();
  }

  public override async Task
    OnDisconnectedAsync(Exception exception)
  {
    var result = await presenceTracker.ConnectionClosed(
      Context.User.Identity.Name);
    if (result.UserLeft)
    {
      await Clients.All.SendAsync("newMessage", "system",
        $"{Context.User.Identity.Name} left");
    }
    await base.OnDisconnectedAsync(exception);
  }

  public async Task SendMessage(string message)
  {
    await Clients.All.SendAsync("newMessage",
      Context.User.Identity.Name, message);
  }
}
```

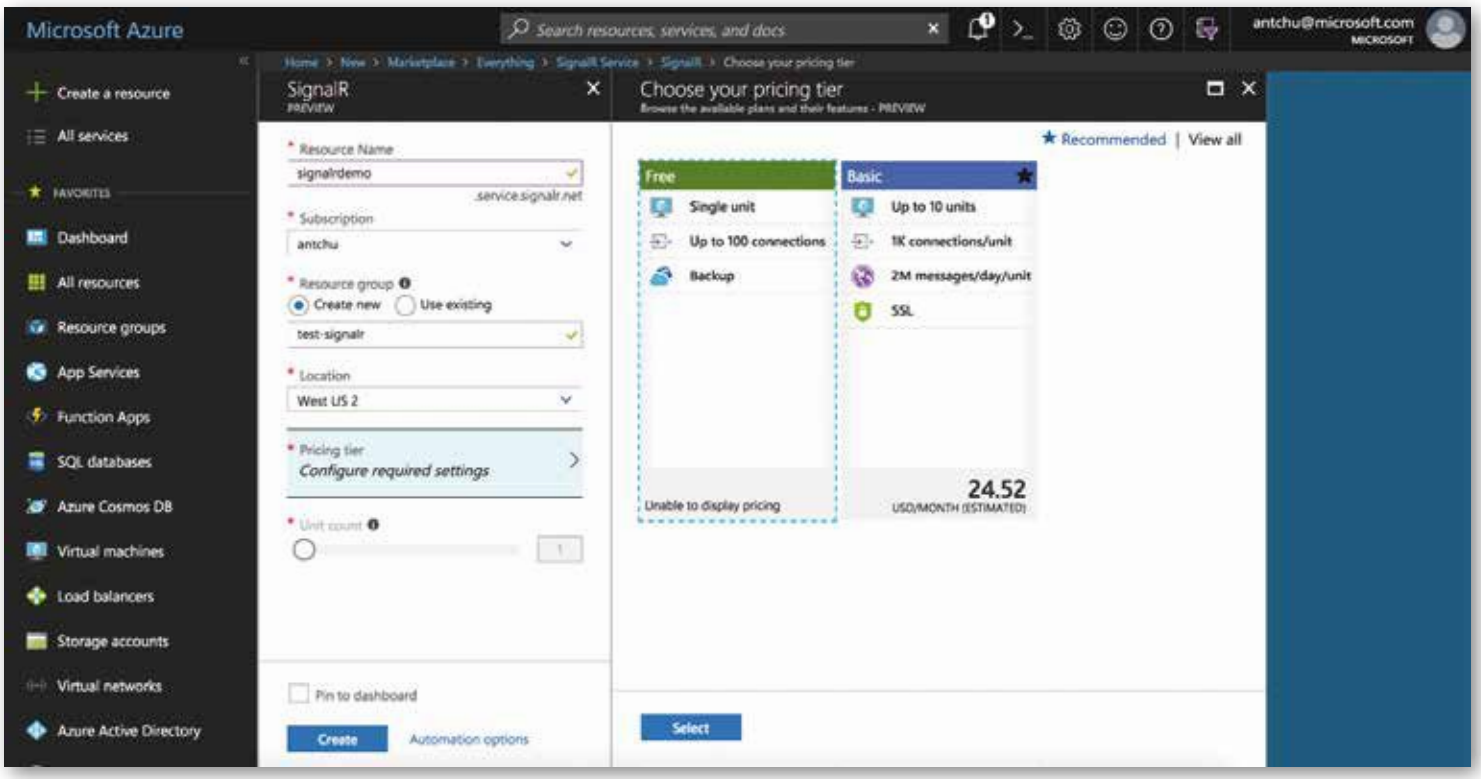



**Figure 6:** Create an Azure SignalR Service instance in the Azure portal.

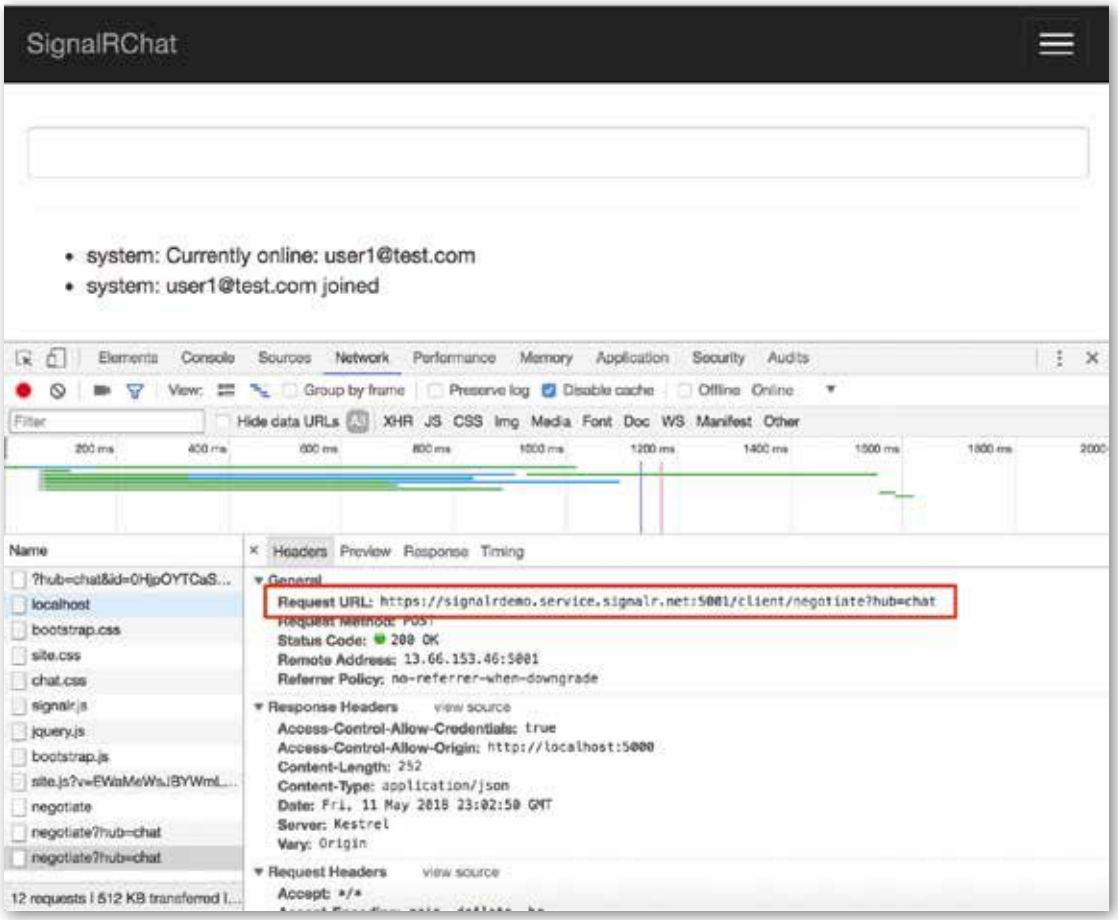



**Figure 7:** The application now connects to the Azure SignalR Service

### To Learn More

For more about Azure SignalR Service, go to http://aka.ms/signalr_service.

The following console application uses the HubConnectionBuilder from the SignalR client library to configure the hub connection. The syntax is similar to the JavaScript client. You can add methods that are invoked by the hub. **Listing 4** is a console application that uses the library to connect to the SignalR hub you built earlier.

When run, the application prompts for a username and password, and then uses the credentials to request a token from the token endpoint you created previously. It then connects to the SignalR hub and displays any messages that are sent. Send messages by typing into the console, as shown in **Figure 4**.

## Reconnections

ASP.NET SignalR automatically handles restarting the connection when a disconnection occurs. Reconnection behavior is often specific to each application. For this reason, ASP.NET Core SignalR doesn't provide a default automatic reconnection mechanism.

For most common scenarios, a client only needs to reconnect to the hub when a connection is lost or a connection attempt fails. To do this, the application can listen for these events and call the start method on the hub connection.

Here's how it looks in JavaScript, adding reconnection logic when a connection is closed or a connection attempt has failed.

```
connection.onclose(reconnect);

startConnection();

function startConnection() {
  console.log('connecting...');
  connection.start()
    .then(() => console.log('connected!'))
    .catch(reconnect);
}

function reconnect() {
  console.log('reconnecting...');
  setTimeout(startConnection, 2000);
}
```

## Presence

A common requirement for real-time applications is to track which users are currently online. ASP.NET Core SignalR makes it easy to add presence to an application.

A basic way to add presence to an ASP.NET Core SignalR is to track users in memory. Because a single user identity can potentially have more than one connection to the hub, the application needs to track the number of connections per user.

**Listing 5** shows a simple class that implements a user tracker. Whenever a connection is opened or closed, Con-

nectionOpened or ConnectionClosed is called. Based on the number of connections for the user associated with the connection event, the methods return a status to indicate if user has joined or left.

In the hub, call the tracker in **Listing 6** and broadcast a message when a user has joined or left. Also, send the list of currently online users when a connection is created, as shown in **Listing 6.**

Now when users join or leave the chat application, a system message appears, as shown in **Figure 5.**

This example gets you started in creating a presence system for your SignalR hub, but a more robust solution will be required to handle scale-out and server restarts.

## Azure SignalR Service

In order for ASP.NET Core applications running SignalR to scale out to more than one instance, a backplane must be set up. Instances communicate over the backplane to ensure that messages reach the correct destinations no matter which clients are connected to which instances.

Backplanes can be built using technologies with **publish-subscribe** features, such as Azure Service Bus, Redis, or SQL Server. However, a much simpler way to scale out your ASP.NET Core SignalR application is to use a new Azure service called Azure SignalR Service. SignalR Service handles the scale-out for you, so you can support large numbers of connections without setting up a backplane yourself. SignalR Service integrates into an existing ASP.NET Core SignalR application using a NuGet package and requires minimal modifications to your code.

To get started, use the Azure Portal to create a new SignalR Service instance, as shown in **Figure 6**.

After the service instance is created, install the SignalR Service library into your ASP.NET Core SignalR application using the Microsoft.Azure.SignalR NuGet package. By default, the library looks for the SignalR Service connection string in an application setting named Azure:SignalR:ConnectionString.

To enable Azure SignalR Service in your application, add a call to AddAzureSignalR() to the SignalR configuration in Startup.cs.

```
services.AddSignalR().AddAzureSignalR();
```

Then replace the call to UseSignalR with a call to UseAzureSignalR.

```
app.UseAzureSignalR(builder =>
{
  builder.MapHub<Chat>("/chat");
});
```

Now when you run the application, instead of directly connecting to the hub in your ASP.NET Core application, clients connect to SignalR Service. All communication between clients and your application's SignalR hub go through the SignalR Service. This allows you to scale the service up and down at any time to handle different levels of traffic without any modifications to the application's code or hosting environment.

If you inspect the HTTP requests in your browser (**Figure 7**), you can see that the application now connects to Azure SignalR Service instead of to the SignalR hub in your Web application.

Note that the ASP.NET Core JWT authorization added earlier must be disabled in order for SignalR Service to integrate with this application (cookie authorization is fine). SignalR Service supports JWT authorization as well, but its integration is beyond the scope of this article.

## Conclusion

As you can see, adding real-time Web functionalities to your cross-platform Web applications is easy using ASP.NET Core SignalR. And with the Azure SignalR Service, you now have a fully managed backplane for highly scalable applications.

Anthony Chu

# Entity Framework Core 2.1: Heck Yes, It's Production Ready!

Although Entity Framework Core (EF Core) 2.1 is a minor release on top of EF Core 2.0, it seems that with this version, EF Core has turned the corner regarding core features, advanced features, and stability. This article looks at some of the new features and improvements in 2.1 and begins by taking a look at the bigger picture of EF Core's goals, why it's moving in the direction that it's going, and where we hope to see EF Core in the future.

**Julie Lerman**

thedatafarm.com/blog
twitter.com/julielerman

Julie Lerman is a Microsoft Regional director, Docker Captain, and a long-time Microsoft MVP who now counts her years as a coder in decades. She makes her living as a coach and consultant to software teams around the world. You can find Julie presenting on Entity Framework, Domain Driven Design, and other topics at user groups and conferences around the world. Julie is the author of the highly acclaimed "Programming Entity Framework" books, the MSDN Magazine Data Points column, and popular videos on Pluralsight.com.

## Revisiting the EF Core Vision

EF Core has always been about trying to hit a sweet spot between a rich object/database mapper experience while still trying to be simple, lightweight, down to the metal, extensible, and less tied to relational database concepts than other existing products. And it's cross-platform, too! EF Core follows several tenets:

- You shouldn't have to pay for complexity that you're not going to use.
- You should be able to target different data stores while still using some of the same basic concepts.
- Complex abstractions shouldn't get in the way of accessing the full capabilities of the underlying store.

EF6 is a very complex O/RM, at least in terms of the mappings that it supports and how the mapping is implemented. That had some consequences, for example, in terms of memory usage, startup time, and even in the difficulty to evolve the codebase and add new features. That's why EF Core started much simpler. EF Core's approach to inheritance is a good example of this, supporting only Table per Hierarchy (TPH). There are no Table per Type (TPT), no Table per Concrete Type) TPC, and no hybrid inheritance mappings.

As EF Core evolves, some of the features that were present in EF6 are starting to appear. You might interpret this as EF Core catching up with its big brother. But when you take a closer look at the features, you see that they have different inflections and personalities. And it's unlikely that EF Core will ever support everything that EF6 did. Doing so would incur the cost of becoming as complex as EF6.

You can read the list of "guiding principles" the EF team has used in the initial development of EF Core at https://github.com/aspnet/EntityFrameworkCore/wiki/guiding-principles. One of those principles that may be surprising to you is "EF Core prioritizes features based on their individual merit rather than EF6 parity."

Some examples:

- FromSql, introduced in EF Core 1, is a lot like SqlQuery in EF6, but is composable, meaning that additional LINQ operators are applied after it can be translated to SQL.
- Owned entities, which arrived in EF Core 2, are a lot like complex types in EF6, but they have stronger semantics for aggregates and, in the future, they'll support collections.
- Query types, one of EF Core 2.1's new features, enable projecting SQL queries into types that aren't mapped to a table, but they also enable many other scenarios, including mapping to tables without keys.
- Lazy loading returns in EF Core 2.1, but uses a different (non-monolithic) architecture, and tries to fix some of the problems of the past, for example: being unsure if your entities are capable of lazy loading or not, or forgetting to make the navigation properties virtual.
- GroupBy translation debuts in EF Core 2.1 and it works differently from EF6 in some cases. One example there is that it can generate HAVING clauses in the SQL translation.
- Data Seeding also returns to the fold in EF Core 2.1 but instead of the imperative model of EF6, it uses a declarative model in which seed data is part of the model.
- System.Transactions support, which was in EF since the very first version, appears in EF Core 2.1. That's finally possible because it was added in .NET Core 2 and is now supported in some of the most important ADO.NET providers.
- Change tracking events (Tracked and StateChanged) debut in EF Core 2.1.

There are also features that have received a completely new treatment and are far superior to their EF6 counterparts.

- The set of types you can use in properties in EF6 was fixed. In EF Core 1, the set of types was borrowed directly from the underlying database provider. EF Core 2.1 enables an even richer set of types through value conversions.
- Alongside lazy loading, the EF team has enabled parameter constructors in entities. Some of us are very happy to no longer need to justify adding private parameterless constructors into Domain-Driven Design models just to make EF happy!
- EF Core 2.1 includes code analyzers that detect possibly unsafe usage of FromSql and ExecuteSqlCommand.

Another convenient change is that the .NET Core Command Line Interface (CLI) commands (usually employed to manage migrations and to reverse engineering classes

from an existing database) have now been moved to the .NET Core SDK, so you no longer need to manually edit project files to have them included.

## Exploring Features with Code

Now let's take a closer look at some of the new features. I'll use a simple model of Team and TeamMember to demonstrate the features.

```
public class Team
{
    public int TeamId { get; set; }
    public string Name { get; set; }
    public string TwitterAlias { get; set; }
    public List<TeamMember> Members { get; set;}
}

public class TeamMember
{
    public int TeamMemberId { get; set; }
    public string Name { get; set; }
    public string Role { get; set; }
    public int TeamId { get; set; }
}
```

The samples presume that you're using the SQL Server database provider.

## EF6 Parity Features

Lazy loading and LINQ's GroupBy support were some of the most (loudly) requested features from Entity Framework to be enabled in EF Core 2.1.

### *Lazy Loading*

There are two ways to implement lazy loading in EF Core starting with 2.1. For those new to EF, lazy loading allows EF Core to retrieve related data, as needed, without you writing additional queries. The first uses proxies, as had always been done with Entity Framework. However, as the proxy logic isn't a core feature of EF Core, it's encapsulated in its own package, Microsoft.EntityFrameworkCore.Proxies, which you'll need to add to your project. You'll also need to signal to your DbContext that you want the proxies to be used, which you must do with the DbContext options. If this is part of an ASP.NET Core application, you can specify it in the AddDbContext method of the startup files Configure method.

```
services.AddDbContext<TeamsContext>(
    optionsBuilder => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString));
```

If you're configuring your DbContext in its OnConfiguring override, the syntax is:

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
}
```

With lazy loading enabled on the context, any virtual navigation property is overridden under the covers by the proxy at run time.

That means that in order for the Members property of Team to be lazy loaded along with a Team object, you'll need to change its declaration to virtual:

```
public virtual List<TeamMember> Members
    { get; set; }
```

Something that's different from earlier versions of EF is that it can't be selective about which navigation properties can be lazy loaded. You must mark every navigation property in your model as virtual. This is a side-effect of EF Core helping you remember to mark the properties as virtual. Perhaps it will be more lenient in the future.

Now, any reference to the Members property on an existing instance of a Team triggers EF Core to retrieve the members for that object. Even this example requesting the count of the members populates the members and then returns the count.

```
var team = context.Teams.Find(101);
var memberCount = team.Members.Count();
```

If you have enabled lazy loading but the model builder can't find any virtual navigation properties, it alerts you. The usual caveats that you've always had with lazy loading still apply:

- The context needs to be available and in scope in order for it to perform this task
- Lazy loading is an all-or-nothing means of retrieving related data. If the navigation is a collection, not a reference, it loads all of the related items in that collection from the database.
- EF Core uses a flag to determine whether related data has been loaded and won't reload even if the data in the database has changed since the first load.
- Roundtrips to the database are implicit and it's easy to end up executing N+1 queries.
- Lazy loading is a sync/blocking I/O and can take an arbitrary amount of time. If you want to keep things asynchronous, use eager or explicit loading.

Proxies can be messy at runtime and you may prefer not to use them. Or you may not be able to use proxies: There are .NET implementations that need to know all types at compile time, like UWP and Xamarin, and therefore don't allow emitting new types at runtime. The alternate way to use lazy loading in EF Core 2.1—which doesn't rely on proxies—is a completely new mechanism with two ways to achieve it. I'll explain the feature and those details that originally confused me, and let you look at the detailed code in the EF Core docs (https://docs.microsoft.com/ef/core/querying/related-data#lazy-loading) to see how to implement it.

The first way is to let EF Core inject one of the key objects from the new Microsoft.EntityFrameworkCore.Abstractions package, an implementation of the ILazyLoader service, into your entity. This requires setting up your entity constructor to accept the ILazyLoader type as a

### More on Defining Queries

See the blog post "Defining a Defining Query in EF Core 2.1" (http://thedatafarm.com/data-access/defining-a-defining-query-in-ef-core-2-1/) for an example of creating and using a defining query that incorporates the new query type feature along with the new ToQuery method.

parameter and then calling its Load method in the getter of the navigation property to be lazy loaded. There's an alternate way to allow EF Core to inject the ILazyLoader service without making any reference to its assemblies and reducing coupling with anything related to EF Core. That's by using a System.Action object as a parameter of the constructor with the parameter named lazyLoader, a current naming requirement. EF Core sees that and injects the instance.

```
private Team(Action<object, string> lazyLoader)
```

### *LINQ GroupBy Translation*

GroupBy is an important query operation. Up until EF Core 2.1, the LINQ GroupBy method was never translated to SQL and was always evaluated in memory on query results. EF Core 2.1 brings the first wave of translation into SQL, although there's still more to go.

Here's an example of grouping on a single column and projecting the group key along with an aggregate:

```
context.TeamMembers
  .GroupBy(m => m.TeamId)
  .Select(g =>
      new { TeamId = g.Key, Count = g.Count() })
  .ToList();
```

The SQL sent to the database reflects that the group by is translated

```
SELECT [m].[TeamId], COUNT(*) AS [Count]
FROM [TeamMembers] as [m]
GROUP BY [m].[TeamId]
```

The results are a pair of anonymous types with requested properties and of course, the correct values, as shown in **Figure 1**.

You can find detailed samples of each of each supported scenario in the GitHub issue "Relational: Support translating GroupBy() to SQL" (https://github.com/aspnet/EntityFrameworkCore/issues/2341). Here's an overview:

- Group by a single column or by multiple columns where those columns are expressed as an anonymous type. For example, if there were also a **Role** property in teamMember, you could group by TeamId and Role as GroupBy(m=>new{m.TeamId,m.Role}).
- GroupBy can be translated when it's been appended to a complex query.

| Name | Value |
|---|---|
| membersGroup | Count = 2 |
| [0] | { TeamId = 101, Count = 2 } |
| Count | 2 |
| TeamId | 101 |
| [1] | { TeamId = 102, Count = 2 } |
| Count | 2 |
| TeamId | 102 |

**Figure 1:** The results of the GroupBy query

- Group by a constant or a variable.
- Group by scalar properties of a related type.
- Select into a known type.
- Order or filter on a key or aggregate after GroupBy

Also from the GitHub issue, here's list of scenarios that aren't supported in EF Core 2.1:

- Grouping on an entity (e.g., a reference navigation property)
- Projecting non-aggregate scalar subqueries after grouping, e.g., FirstOrDefault()
- Making groups of multiple entityTypes using anonymous types
- Using Key/Aggregate values after GroupBy in joins

## EF Core Gets Out of the Way of Your Domain Design

Anything that you can do to avoid modifying your business classes in order to accommodate EF Core's mapping rules makes EF Core more appealing to many developers. To this end, there are two notable improvements in EF Core 2.1: value conversions and the ability for EF Core to materialize results without an available parameterless constructor.

### *Value Conversions*

The new value conversion feature allows you to have more control over mapping CLR types to database types. EF has always been limited by the fact that you could only map types if they were natively supported by the database provider. In EF Core 2.1, you can use the types you want in your model without being constrained by the database mapping support and you can define a converter to take care of identifying the proper database type as well as when saving and querying data. There are also a number of pre-defined converters built in for some common scenarios. Let's take a look at one of these: TimeSpanToTicksConverter.

TimeSpan maps by default to SQL Server's Time type, which represents the duration since midnight. But developers who wanted to truly store it as a duration, not a time, especially if that duration is more than 24 hours, had to come up with clever strategies. Now you can easily store a TimeSpan duration as a bigint in the database and retrieve that value back into the TimeSpan type.

Here's a new TimeSpan property, TypicalCommuteTime, in the TeamMember class. Its value is set by the method, CalculateCommuteTime.

```
public TimeSpan TypicalCommuteTime
    { get; private set; }

public void CalculateCommuteTime(
    DateTime start, DateTime end)
{
    TypicalCommuteTime = end.Subtract(start);
}
```

By default, the SQL Server provider creates this new column as a Time type. Ticks in .NET are represented as an Int64, which in SQL Server is a bigint. It's possible to use the HasColumnType("bigint") mapping with the HasConversion with the .NET type and not worry about what the database type is.

### EF6's Relationship to EF Core

Entity Framework was first released in 2008 and evolved to the fairly sophisticated EF6 by 2012. Even so, new innovation was limited by the old code base. With .NET and ASP.NET evolving to their cross-platform Core versions starting in 2015, EF needed to be part of this process and a big decision was made to rewrite EF from the ground up with new code, new modern coding practices and new capabilities in place. The first iterations of .NET Core, ASP.NET Core, and EF Core were targeted for developers and projects who were willing to work with new bleeding edge technology. When these tools reached the 2.0 version, they were all solidly positioned and stable to be production-ready for a much broader group of projects. Forward evolution of EF Core brings additional parity with critical features from EF6 as well as tightening up the APIs and innovating further its capabilities.

```
modelBuilder.Entity<TeamMember>()
    .Property(e => e.TypicalCommuteTime)
    .HasConversion<System.Int64>();
```

EF Core creates a bigint column for you when defining a migration as well as when creating SQL, so you can easily save and query TeamMember objects with the TimeSpan value without needing to be involved with the conversion.

Another built-in converter that will be of interest to many developers is EnumToStringConverter, providing the ability to finally store enums as strings.

Check out the Value Conversion details in the docs at https://docs.microsoft.com/core/modeling/value-conversions to learn more about the built-in converters and creating your own custom conversions. Also, pay attention to the Limitations section in the doc that discusses lack of support for nulls and other things.

### *Materialize Entities That Have Parameterized Constructors*

Another change in EF Core 2.1 should make anyone who's ever added a constructor to their entity classes happy. EF and EF Core have never been able to materialize results without the presence of a parameterless constructor. If you create a constructor with parameters, that overrides the default parameterless constructor inherited from System.Object.

The fix has always been to add a private parameterless constructor to your entities, an annoying concession for developers who don't want to design their entities to satisfy the data persistence layer. Now EF Core can materialize results if the only available constructors have parameters. If there are multiple constructors, EF Core uses the narrower one to materialize the object.

### *Data Annotation Attribute for Owned Types*

If you like to use Data Annotations, be aware that there's now a data annotation provided in the Abstractions package, Owned, to mark properties that are owned types. This aligns with the IsOwned fluent mapping introduced in EF Core 2.0.

## Data Seeding

Data seeding by way of migrations has returned in EF Core 2.1. This is a fresh take on the workflow. In this iteration, you define the seed data as part of the model in the DbContext's OnModelConfiguring method. For example:

```
modelBuilder.Entity<Team>().HasData(
    new Team
    {
        TeamId = 101,
        Name = "Entity Framework",
        TwitterAlias = "efmagicunicorns"
    });
```

You can also add multiple entities as an array:

```
modelBuilder.Entity<Team>().HasData(
    new Team[]
    {
        new Team
        {
            TeamId = 101,
            Name = "Entity Framework",
            TwitterAlias = "efmagicunicorns"
        },
        new Team
        {
            TeamId = 102,
            Name = ".NET",
            TwitterAlias = "dotnet"
        }
    });
```

There are some important points to note about how this works.

- You must specify the key value even if you're seeding a database that generates those keys. EF Core ensures that your values are inserted.
- In the HasData method, you can only specify data for a single type. If you want to also add TeamMembers, you must do that via **modelBuilder.Entity<TeamMember>.HasData()**.
- When you add a migration after defining HasData, new MigrationBuilder methods are used to insert that data. The migration's Down method removes that same data.
- The seed data is only inserted into the database when you call the migrations' update database command, rather than at runtime.
- If you're using the InMemory provider for testing, EnsureCreated reads the HasData method and seeds your InMemory database.

That last point about working with the InMemory provider is important. It's a nice benefit, so you don't have to always write extra code to populate the InMemory context in your tests. Pay attention, in case your tests require different data.

Check the documentation (https://docs.microsoft.com/ef/core/modeling/data-seeding) for other nuances about data seeding.

## Richer Data Models with Query Types

EF Core 2.1 brings you Query types, which enable some scenarios that are critical to many developers as well as a great deal of the software you're building. Query types are a completely new idea in EF Core and didn't exist in any version of Entity Framework. What's interesting is that with this feature, it's the first time the DbContext can do something useful with objects without requiring keys.

Remember that every entity in your model must have a key. A query type is a type that's recognized by DbContext for the sole purpose of executing queries and pushing the results into objects. Although the DbContext is aware of these types, there are three important traits for you to understand. Query types:

- Don't require a key property
- Aren't tracked by the ChangeTracker
- May or may not be mapped to tables in the database

Some of the most important scenarios that query types enable include mapping to database views and tables with no primary key, returning FromSql queries into non-entity types (something we were able to do in EF6), mapping to tables that don't have a primary key, and mapping to queries defined with the new ToQuery mapping (a.k.a. defining queries).

Database view mapping is probably the most requested capability. The documentation provides a nice walk through for this, so here are some highlights. To map a database view, you'll need:

- A view in the database
- A class that matches the schema
- A QueryType mapping in the DbContext that includes the new ToView method to specify the name of the view:

```
modelBuilder.Query<TeamMemberCount>()
    .ToView("View_TeamMemberCounts");
```

- Optionally, a DbQuery property, which is like a DbSet, in the DbContext class. This example assumes that the class is named TeamMemberCount:

```
public DbQuery<TeamMemberCount> TeamMemberCounts
    { get; set; }
```

If you choose not to define the DbQuery, you can use the dynamic syntax when writing your query. You'll see an example of this below with FromSql.

Now let's look at the FromSql support that query types enable. Here's an example of a stored procedure that takes a team ID as a parameter and returns the name and commute time of each team member. Surely that's easy enough to do with a LINQ projection, but given the simple model, I'll have to use this contrived example:

```
CREATE PROCEDURE [dbo].GetCommuteTimes
      @teamId int
AS
    SELECT Name, TypicalCommuteTime
    FROM TeamMembers
    WHERE TeamId = @teamId
```

You'll need a class that matches the results:

```
public class TeamCommute
{
    public string Name { get; set; }
    public TimeSpan TypicalCommuteTime
        { get; set; }
}
```

And you'll need to let the model be aware that this class can be used as a query type. In OnModelCreating, you only need to refer to the query type to achieve this:

```
modelBuilder.Query<TeamCommute>();
```

Notice that I'm using the TimeSpan in the TeamCommute class just as I did above in the TeamMember. You can use the same mapping to ensure that EF Core knows how to translate the System.Tick (an Int64) to the TimeSpan property. Append that to the Query method:

```
modelBuilder.Query<TeamCommute>()
    .Property(e => e.TypicalCommuteTime)
    .HasConversion<System.Int64>();
```

Now you can use the non-entity type with a FromSql query that calls the stored procedure. This query uses a dynamic context.Query<TeamCommute>() rather than defining the DbQuery in the TeamContext class.

```
var teamlist = context.Query<TeamCommute>()
    .FromSql("EXEC GetCommuteTimes {0}",101)
    .ToList();
```

Note that the documentation at https://docs.microsoft.com/ef/core/modeling/query-types says that you can return arbitrary types from the FromSql method. The term arbitrary doesn't mean any undefined class. You'll have to pre-define the class and configure it in the model, as described above.

## The EF Core You've Been Waiting For

There are even more excellent features in EF Core 2.1. Here's a quick list of some of the more notable changes.

- There are new pipeline hooks with the Tracked and StateChanged events on DbContext.ChangeTracker.
- System.Transactions is again supported so you can customize how transactions are handled.
- Developers have long complained about the inability to control how database columns are ordered; now migrations honor the order of properties in your entities when generating the code that creates the table for the first time.
- Derived types are now possible with the eager loading Include method.
- For apps where you're working with the .NET Core CLI and EF Core's command line tools, they've now moved to the .NET Core SDK so you won't have to add the DotNEtCliToolReference into your csproj files.

The attention being paid to EF Core in this version demonstrates that there's no question about EF Core being production ready. There may yet be a particular feature that you're still waiting for, but for the majority of projects, EF Core 2.1 has come to the point where you can start calling it feature-rich and ready for prime time.

Julie Lerman

CODE

### Expert Help

Tech reviewer for this article Diego Vega is a Program Manager for Entity Framework and data access on the .NET Team at Microsoft. Look for his blogs about Entity Framework on the .NET Blog at https://blogs.msdn.com/dotnet.

# Qualified, Professional Staffing When You Need It


CODE Staffing provides professional software developers to augment your development team, using your technological requirements and goals. Whether on-site or remote, we match our extensive network of developers to your specific requirements. With CODE Staffing, you not only get the resources you need, you also gain a direct pipeline to our entire team of CODE experts for questions and advice. Trust our proven vetting process, and let us put our CODE Developer Network to work, for you!

Contact CODE Staffing today for your free Needs Analysis.

*Helping Companies Build Better Software Since 1993*

**www.codemag.com/staffing**
832-717-4445 ext. 9 ● info@codemag.com

CODE STAFFING

# Learn Python with Visual Studio

In the Stack Overflow Developer Survey for 2018, Python sits comfortably in the first place as the most wanted programming language, with JavaScript and Go coming second and third. Python also ranks third in the "most loved" category. Despite the age (20 years and counting), Python's popularity keeps growing, and for good reasons. It's reliable, flexible, easy to learn,

**Nicola Iarocci**
@nicolaiarocci
www.nicolaiarocci.com

Nicola is a passionate Python and C# developer based in Ravenna, Italy. He's the co-founder of CIR2000 where he leads the development of Amica 10, an accounting software for Italian small businesses. He's the creator and maintainer of a number of Python and C# open source projects such as the Eve REST API Framework, Cerberus, and more. Awarded the Microsoft MVP Award for Development Technologies and the MongoDB Master Award, Nicola is also a trainer and speaker. In his free time, he runs the local CoderDojo, a coding club for kids, and DevRomagna, the leading development community in his area.

open-source, and cross-platform since the beginning. It also helps that, over time, it enjoys a robust and active developer community and incredibly rich eco-system of free libraries supporting all kind of usages: Web applications and services, desktop apps, scientific computing, scripting—you name it.

Surprisingly enough, not many .NET developers know that their favorite development tool, Visual Studio, offers superb support for Python. In this article, you'll see how you can leverage your hard-acquired Visual Studio skills to work immediately and efficiently with this fantastic language.

## On the Relationship Between Python and Visual Studio

Nowadays, the Visual Studio brand encompasses several different products. There is Visual Studio for Windows, Visual Studio for Mac, and then the cross-platform Visual Studio Code editor. As you know, despite their names, these are entirely different products, with different prerequisites and feature-sets. Python support is available in Visual Studio for Windows. On Mac and Linux, and of course in Windows, you can count on a grand Python experience in Visual Studio Code.

Presently, Visual Studio for Mac offers no support for Python, and frankly, I wouldn't bet on something like that happening any time soon. Anecdotal fact: One year ago, someone opened a feature request ticket on UserVoice. Since then, the ticket has received many votes (mine included) and comments, but no feedback from the team. At its core, VSMac is MonoDevelop with many new extensions added to support new workloads (.NET Core, Azure Deployment, Unity). According to Miguel De Icaza (founder of the GNOME, Mono, and Xamarin projects), the internals are being progressively replaced with Visual Studio code when applicable. So yes, it's a different beast. Interestingly though, if you look at the MonoDevelop feature matrix, you'll find that Python 2 bindings are available for Linux. Wait and see, I guess.

In this article, I'll cover the flagship product, Visual Studio for Windows. The story between Python and VS has been going on for a long time, and I can be very precise about that as the project is and always has been open source (https://github.com/Microsoft/PTVS), something that allowed me to clone the repository and play with it a little bit. As it turns out, the first commit date is March 8, 2011 (**Figure 1**).

```
[nicola:~/code/PTVS] master ± git log --oneline | tail -1
00d974b Python Tools for Visual Studio - initial commit
[nicola:~/code/PTVS] master ± git log 00d974b
commit 00d974b73bcf52cf08b0af61bac119b37fae9869
Author: Dino Viehland <dinov@microsoft.com>
Date:   Tue Mar 8 16:49:11 2011 -0800

    Python Tools for Visual Studio - initial commit
[nicola:~/code/PTVS] master ±
```

**Figure 1:** Exploring the open source repository, I discover that the first commit dates was in 2011.

In the dev team, there are Microsoft employees who are, or have been, Python core developers. These include Steve Dower, Eric Snow, Dino Viehland, and Brett Cannon, who is now leading the VSCode Python extension. Project maturity and team composition, I think, offer a clear view of Microsoft commitment to the language. Python in Visual Studio is real. So real, in fact, that today Python ships as an integral part of the product.

> In the dev team, there are Microsoft employees who are, or have been, Python core developers.

## Installing Python in Visual Studio

Both VS2017 and VS2015 installers allow you to add Python as an option, either on the first the install or later. There are small differences, however. VS2017 is capable of installing the language interpreter if needed, whereas with VS2015, you have to install the interpreter separately if it's missing in the target system. Also, in VS2017 (15.2 or later), Python comes as a standard workload. With the VS2015 installer, you need to enable the Python Tools for Visual Studio package from the list of available languages instead.

If, for some odd reason, you're stuck with VS2013, VS2012, or even VS2010, well, good news! You can still add Python manually by downloading and installing the appropriate version of the stand-alone installer (PTVS 2.2 for VS2013; PTVS2.1 for VS2012 and VS2010).

Long story short, you can have the best Python experience in Visual Studio 2017. You can still enjoy Python with previous VS versions, but seriously, you should upgrade as soon as possible, and of course not just because of Python. VS2017 is better (and much more performant) in so many ways. For the remainder of this article, I'll refer to Visual Studio 2017 for Windows.

I assume you already have Visual Studio installed. Run the Installer, then select the **Modify** option. In the Workloads tab, select the **Python development** workload (**Figure 2**).

When you select the workload, you can fine-tune what gets installed in the right-side pane. I suggest that you

go with the default. Just make sure that Web support is checked, as well as Python 3 64-bit. If you're just starting out, skip Python 2. It's going to end-of-life soon (check out the real-time clock at https://pythonclock.org), and Python 3 is much better in so many ways. Unless you have to deal with some grumpy old library, Python 2 just isn't worth it, and you can always come back and add it later if needed.


**Figure 2:** Make sure that the Python development workload is selected.

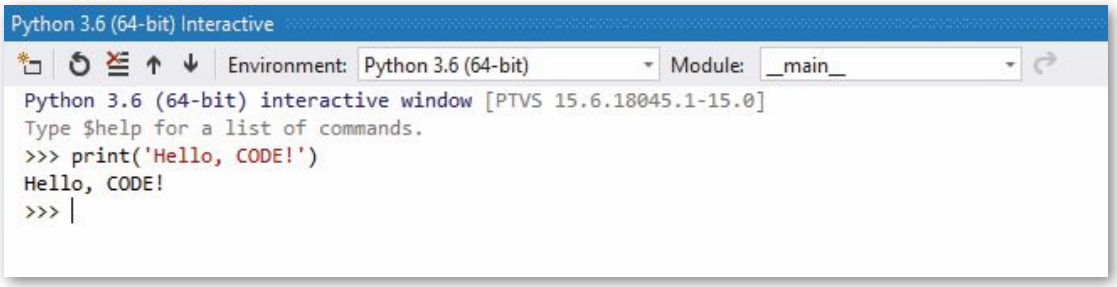



**Figure 3:** If it doesn't work, go back and check your steps.

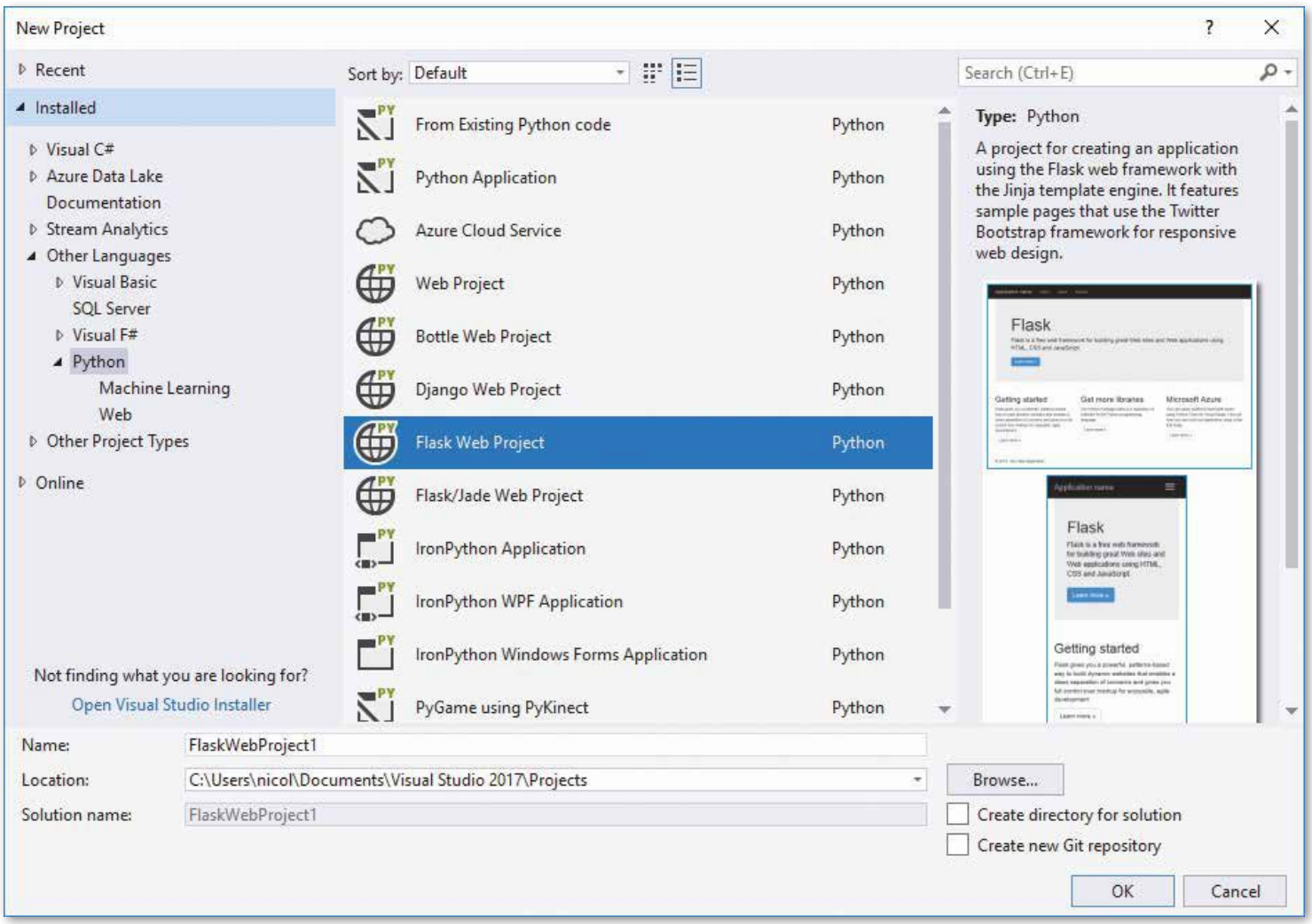



**Figure 4:** Python support in Visual Studio offers a number of project templates; Flask is my favorite.

Once the workload installation is complete, open Visual Studio and hit **Alt+I**, or click **Tools/Python/Python Interactive Window** to open, you guessed it, the Python interactive window. If it opens, success! Python is sitting right there, at your fingertips. To test it, enter something pythonic and very original, as I did in **Figure 3.**

## Working with Python in Visual Studio

The TL;DR (too long; didn't read) version of this article is: you work with Python as you would with any other .NET language. Everything works as you would expect. Allow me to elaborate a little bit.

### *Project Templates*

In the New Project window, pick **Installed/Other Languages**, then **Python.** Notice how Visual Studio proposes a list of project templates including (if you opted-in for Web support) Web applications built with well-known frameworks such as Flask and Django. These framework templates include a starter site with some pages and static files, just like the ASP.NET templates we all know. They provide all the code and assets needed to run and debug the server locally and, eventually, deploy to Azure. Support for Python virtual environments is also built-in (I'll get to virtual environments in a minute). In **Figure 4**, you can see that I also have an Azure Cloud Service template. That's because on install, I checked the Azure Cloud Services core tools option. For this first run, pick the most straightforward option: an empty Python Application.

> Visual Studio proposes a list of project templates, including Web applications built with well-known frameworks such as Flask and Django.

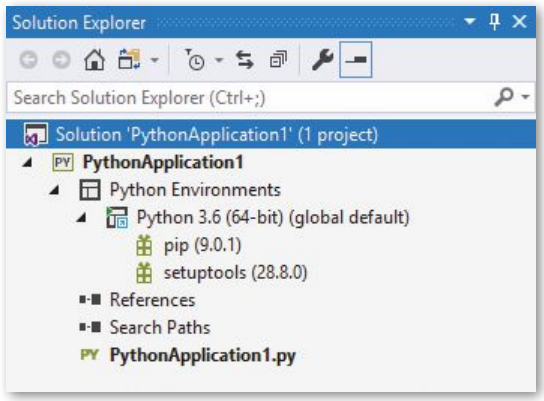



**Figure 5:** Solution Explorer looks familiar yet different. Python Environments node is new. The project file (.pyproj) is a Visual Studio artifact.

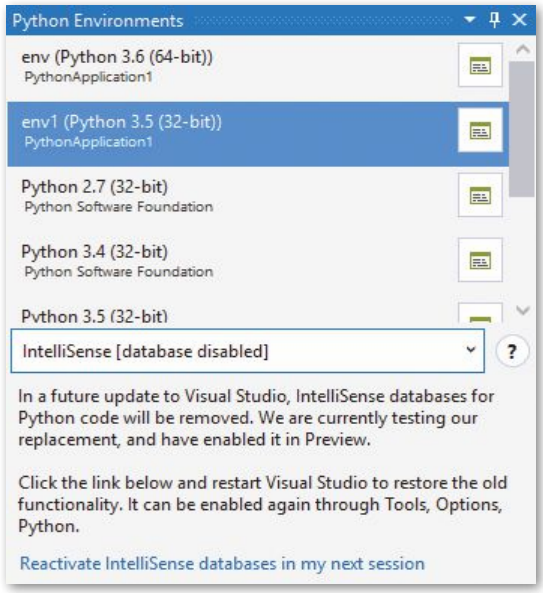



**Figure 6:** Starting with VS 15.6, database-based IntelliSense is disabled, but you can reactivate it if needed. Despite the caption, it's also effective in the standard release (non-Preview).

### *Solution Explorer*

As you would expect, a new solution appears in the Solution Explorer. It holds a single Python project (PythonApplication1.pyproj) with a single Python file (PythonApplication1.py—all Python files use the .py suffix). Notice that in addition to the well-known **References** node, you also have **Python Environments,** which, unless you are already familiar with Python, is new. When expanded, you can see the Python interpreters that are available to you. Expand an interpreter node to see the libraries installed into that environment (**Figure 5**).

I'll get to virtual environments in a moment. Let's write some code first.

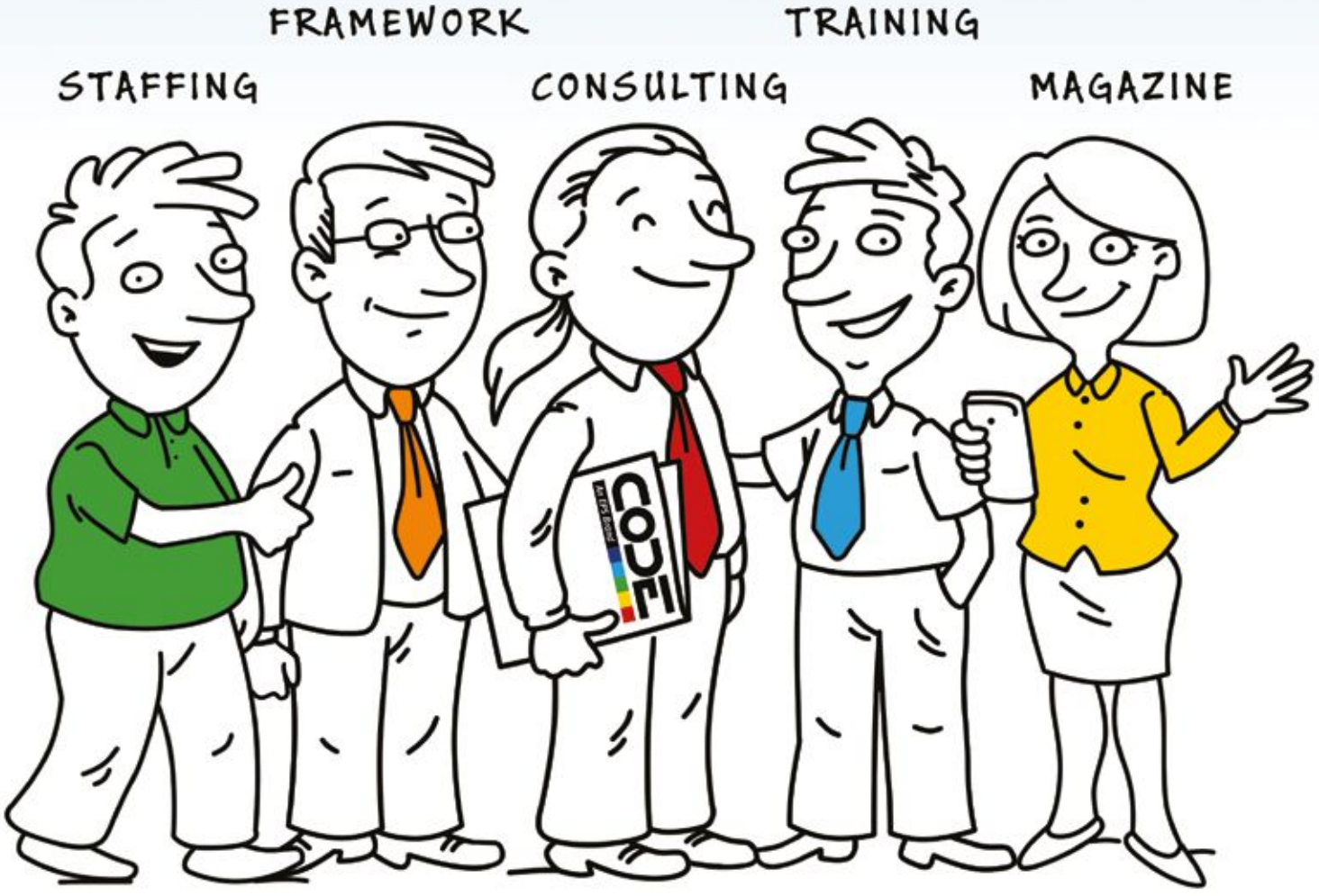
### *IntelliSense, Code Completions, Type Signatures, Etc.*

As you know, IntelliSense provides code completion, type signatures, quick identifier info, code coloring, and probably more cool features. To the .NET developer, the IntelliSense features set is a no-brainer, but the thing is, stuff like code completion and type signatures are somewhat of a challenge in Python, which is strongly typed but has dynamic semantics.

> For some packages, you get a better IntelliSense experience; for others, well, things are still a bit clumsy

Until not too long ago, every time you installed or updated a Python package, Visual Studio took its time to scan it and then update an internal cache, also known as "completion DB." Depending on the number of packages installed and their size, the process could become (and, in fact, was) slow and inefficient. Not exactly the best user experience. But since Visual Studio 15.6, things have changed. It now performs a lightweight analysis of Python modules as you import them into the code (and not at install time). The problem with this new technique is that not all packages provide the metadata required to

# CODE - More Than Just CODE Magazine




The CODE brand is widely-recognized for our ability to use modern technologies to help companies build better software. CODE is comprised of five divisions - CODE Consulting, CODE Staffing, CODE Framework, CODE Training, and CODE Magazine. With expert developers, a repeatable process, and a solid infrastructure, we will exceed your expectations. But don't just take our word for it - ask around the community and check our references. We know you'll be impressed.

Contact us for your free 1-hour consultation.

*Helping Companies Build Better Software Since 1993*

**www.codemag.com**
832-717-4445 ext. 9 • info@codemag.com

CODE
An EPS Brand

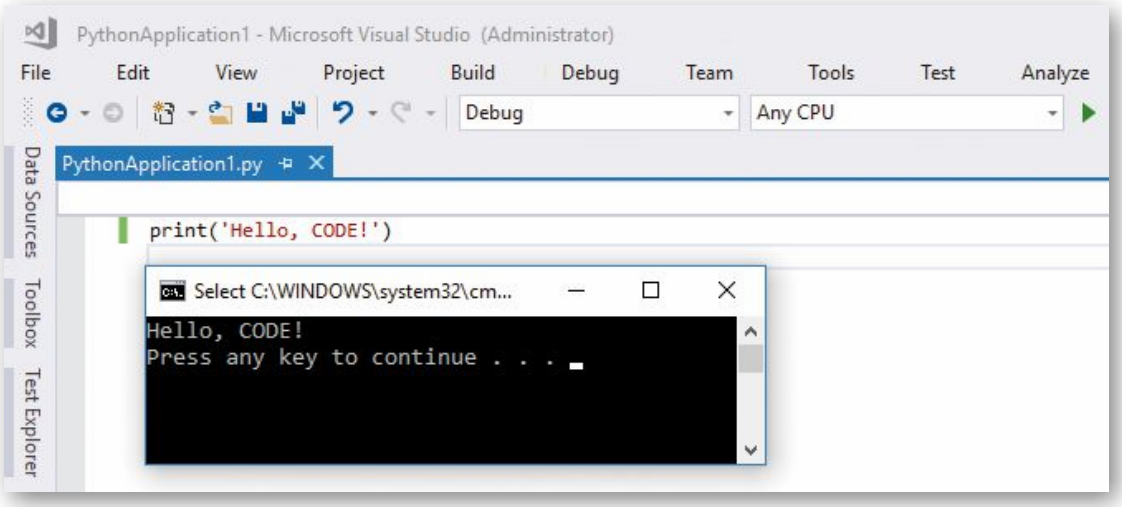



**Figure 7:** Type Python code and then run it, as you would do with any other .NET language. IntelliSense works too.

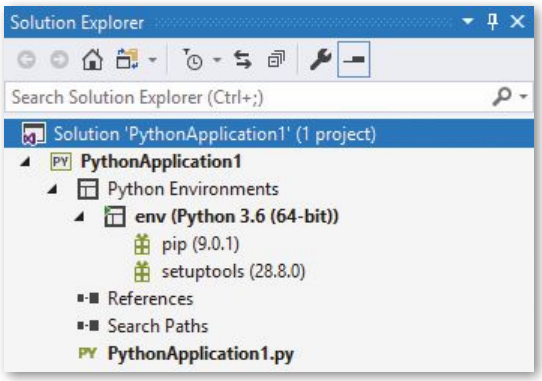



**Figure 8:** You created a virtual environment for the project. It uses the global interpreter (Python 3.6 64-bit) and comes with two pre-installed packages.

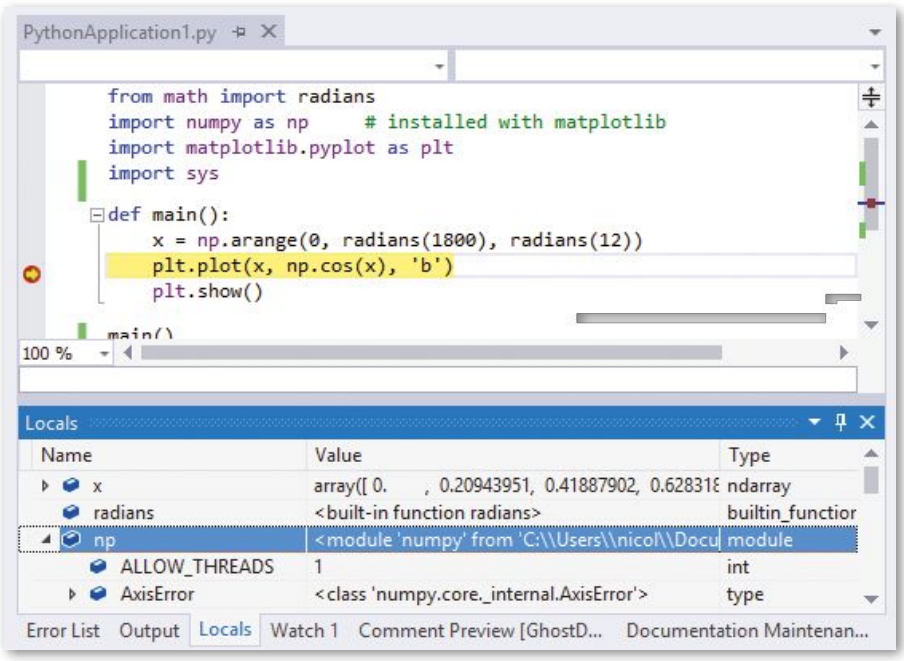



**Figure 9:** The breakpoint I set triggered a pause in code execution. I'm now having fun inspecting the locals.

perform an efficient, lightweight yet in-depth analysis. The result is that for some packages, you get a better IntelliSense experience; for others, well, things are still a bit clumsy. Currently, it's possible to fall back to the old method, although you have to do it explicitly. The plan is to abandon the completion DB approach sometime in the future (**Figure 6**).

My advice is to keep going with the default settings. As more packages include the necessary metadata (namely, .pyi type hint files), the IntelliSense experience improves. The dev team is actively working on this feature. With the next release (15.7), we're getting improved type hints, which are capable of issuing warnings for mismatched types, something not available at this time.

Armed with this knowledge, type **print('Hello, CODE!')** in the code editor (single and double quotes are both valid for strings, although the convention is single quotes). Notice how code completion and signature hints work fine. Running the program is a simple matter of hitting Ctrl+F5, as you would do with any other .NET language (**Figure 7**).

Now, before you replace the code with something new and more exciting, let's dig into Python virtual environments and package management.

## *Virtual Environments and Package Management*

"Dependency hell is a colloquial term for the frustration of some software users who have installed software packages that have dependencies on specific versions of other software packages." –Michael Jang, 2006, "*Linux Annoyances for Geeks.*"

We've all had our share, right? Python's solution to this problem is virtual environments. A virtual environment is an isolated environment, usually (but not necessarily) used by a single project. A project can comfortably sit, along with its dependencies, in its isolated environment, regardless of what dependencies every other project has. To reduce overhead, a virtual environment usually (but again, not necessarily) uses the global interpreter and standard library (think the BCL) but maintains its package store in a private folder.

Visual Studio offers full support for global and virtual environments through the Python Environments window. To open it, right-click on your project's **Python Environments** node in the Solution Explorer, and then click on **View All Python Environments**. On a new project, the first time you do this, you get a list of the global Python interpreters detected by Visual Studio in your system. The one in bold is the default interpreter used by new projects.

> A project can comfortably sit, along with its dependencies, in its isolated environment, regardless of what dependencies every other project has.

Of course, you don't want to work in a global environment. You first want to create an isolated environment for your project. So, go back to the Solution Explorer and right-click on **Add Virtual Environment**. A dialog pops up asking for two fundamental things: the **location of the environment** (by default a project sub-folder), and the **base interpreter**. Visual Studio is smart enough to figure out the most recent, suitable Python interpreter in the system and select it for you. Once you hit the **Create** button, the new environment appears in the Solution Explorer. Expanding it reveals a couple of pre-installed packages: pip and setuptools. These are used to install and upgrade project dependencies (**Figure 8**).

You could add more virtual environments to the project and then switch among them as needed, maybe to experiment with different versions of specific dependencies. The active environment is always in bold.

How do you install packages in the environment? The Python equivalent to NuGet is the Python Package Index (PyPI), "a repository of software for the Python programming language." It predates NuGet by a long shot. Over time, the community has developed and registered over 130K packages on PyPI (https://pypi.org). That's huge, yes. To install a package, right-click the desired environment in the Solution Explorer, then click on **Install Python Package**. In the search box, type *matplotlib*, then click on **pip install matplotlib from PyPI**. Once installed, the package appears in the Python Environments window. Matplotlib is a plotting package. It has a few dependencies that are also downloaded and installed for us.

Back to the code editor. Replace the previous print statement with this code:

```
from math import radians
import numpy as np  # installed with matplotlib
import matplotlib.pyplot as plt

def main():
    x = np.arange(0, radians(1800), radians(12))
    plt.plot(x, np.cos(x), 'b')
    plt.show()

main()
```

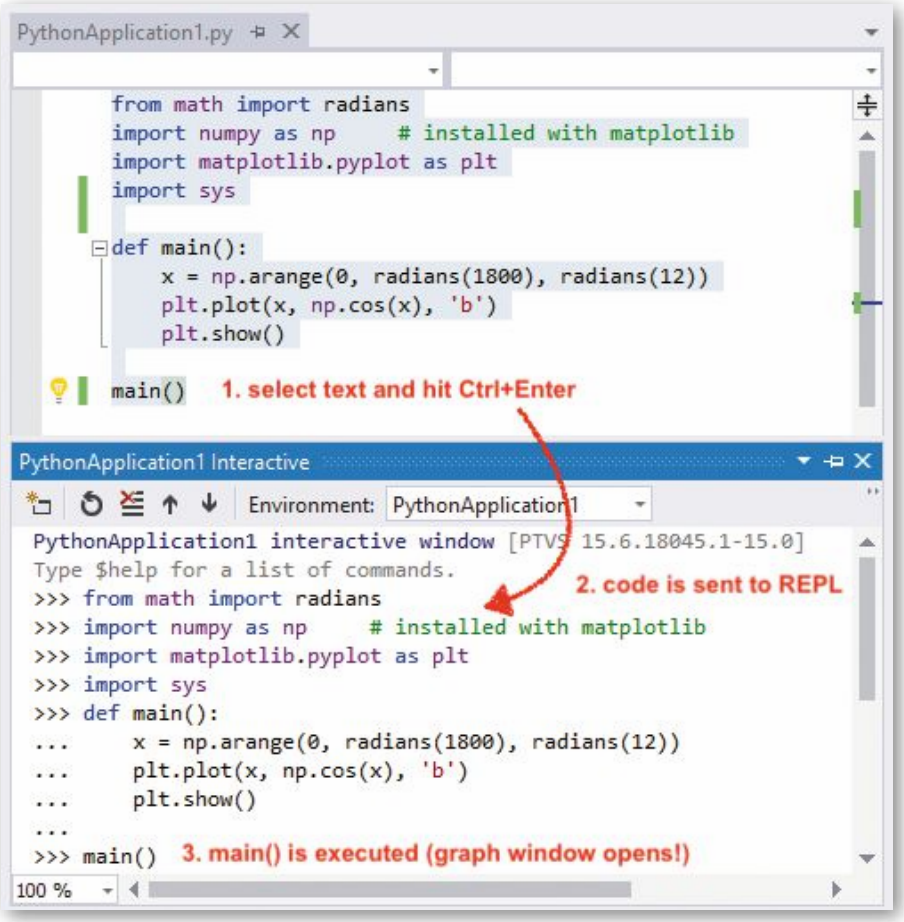



**Figure 10:** When you send a code block to the interactive window, lines are executed at once, sequentially, as they hit the REPL.

On the first line, you're importing the **radians** function from the math module, which is part of the standard library. On the second line, you import from **numpy**, a matplotlib dependency. Finally, you import the **pyplot** function from matplotlib and rename it as **plt**. You then define a plotting function and, on the last line, execute it.

The snippet above comes from the official documentation (https://docs.microsoft.com/visualstudio/python). The project recently got a dedicated technical writer, and it shows. The revamped documentation is top-notch and it's open to public contribution. I contributed a few irrelevant fixes myself, something that hopefully didn't lower the overall quality.

If you run the code, a window with a nice-looking graph should open up.

### *Interactive Debugger and the Python REPL*

Python developers traditionally spend a lot of their time at a terminal or command prompt, switching back and forth from their editor. I know because I do that all the time in my command line/Vim environment (one day we should talk about the advantages of using vim key-bindings in Visual Studio, but this is not this day).

One of the coolest features in Visual Studio is the interactive debugger. Wouldn't it be cool if you could leverage that

with Python too? Well, rejoice! The complete VS debugging experience you're used to when working with .NET is also available with Python. You can run code step-by-step or add breakpoints, even conditional ones. You can, of course, examine the entire program state and change the value of variables, or examine the call stack (**Figure 9).**

Speaking of debugging, keep in mind that Visual Studio can also launch and debug Python applications on a remote Windows computer. It can also debug on a different remote operating system, device, or Python implementation (the standard Python implementation is CPython). Remote debugging on a different operating system is possible thanks to ptvsd, or Visual Studio remote debugging server for Python, a Microsoft-maintained package currently in its alpha stage.

> The complete VS debugging experience you're used to when working with .NET is also available with Python.

One great feature of interpreted languages is the read-evaluate-print-loop (REPL) experience they offer. The VS interactive window provides all the capabilities of the Python REPL. You used the interactive window briefly before. It's handy when you want to try out some ideas or test some code before committing to it. Alternatively, you can send code from the editor to the interactive window, so that you can try it out right away; move the cursor over a line, then press **Ctrl+Enter** (or right-click and select **Send to interactive**).

When you do that, the line content goes straight to the interactive window and in the editor, the cursor moves to the next line, which allows you to continue sending lines to the interactive window as needed. Even better, you can select a block of code in the editor and then hit **Ctrl+Enter** to send it all in one move (**Figure 10**).

Now, this is a REPL, so keep in mind that every line is executed at once, even when you send a whole block of code. In **Figure 10,** I sent the whole little program over to the REPL. The result is that when the last line hits the REPL, the main() function executes and the graph window opens.

### *Profiling and Unit Testing*

Unit testing is critical, even more so with dynamic languages such as Python. Visual Studio can discover, execute, and debug unit tests as seamlessly as it does with .NET languages.

In Python, the convention is that test method names begin with "test." Test classes are subclasses of unittest.TestCase. Visual Studio follows the convention. Add a New Python Unit Test Item to your Python project (Ctrl+Shift+A). What you get is a test1.py file with the following default code:

```
import unittest

class Test_test1(unittest.TestCase):
    def test_A(self):
        self.fail("Not implemented")

if __name__ == '__main__':
    unittest.main()
```

A quick glance at the code reveals that a typical Python test is not very different from a C# test. You import the unittest module from the standard library and then declare a class that inherits from the base test class. The class only holds one test method: test_A.

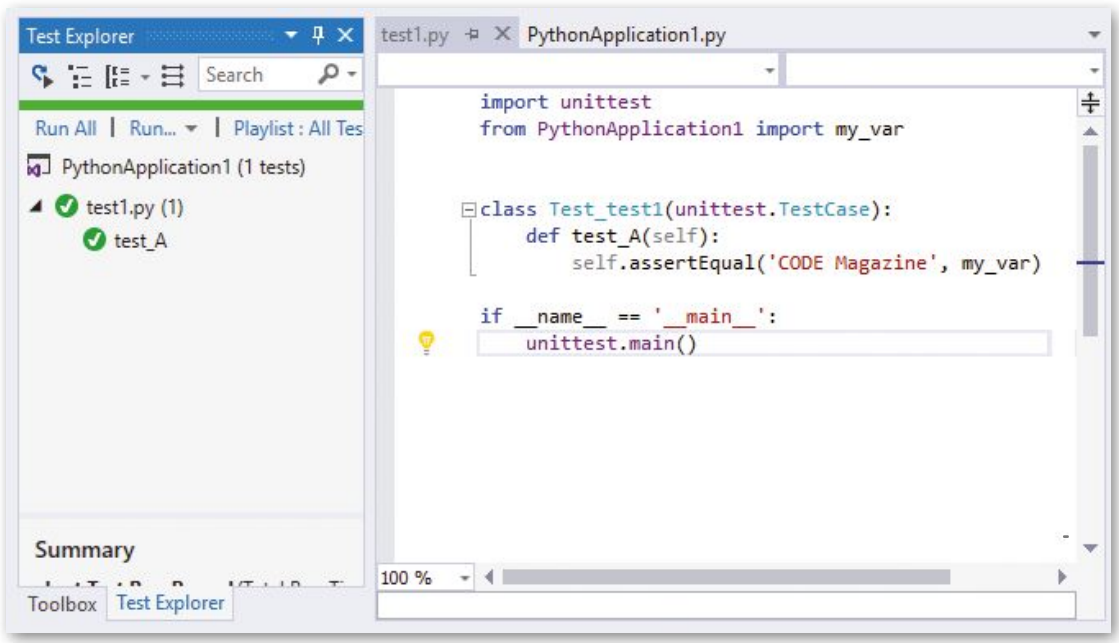



**Figure 11:** Visual Studio can discover, execute, and debug Python unit tests as it does with any other .NET language.

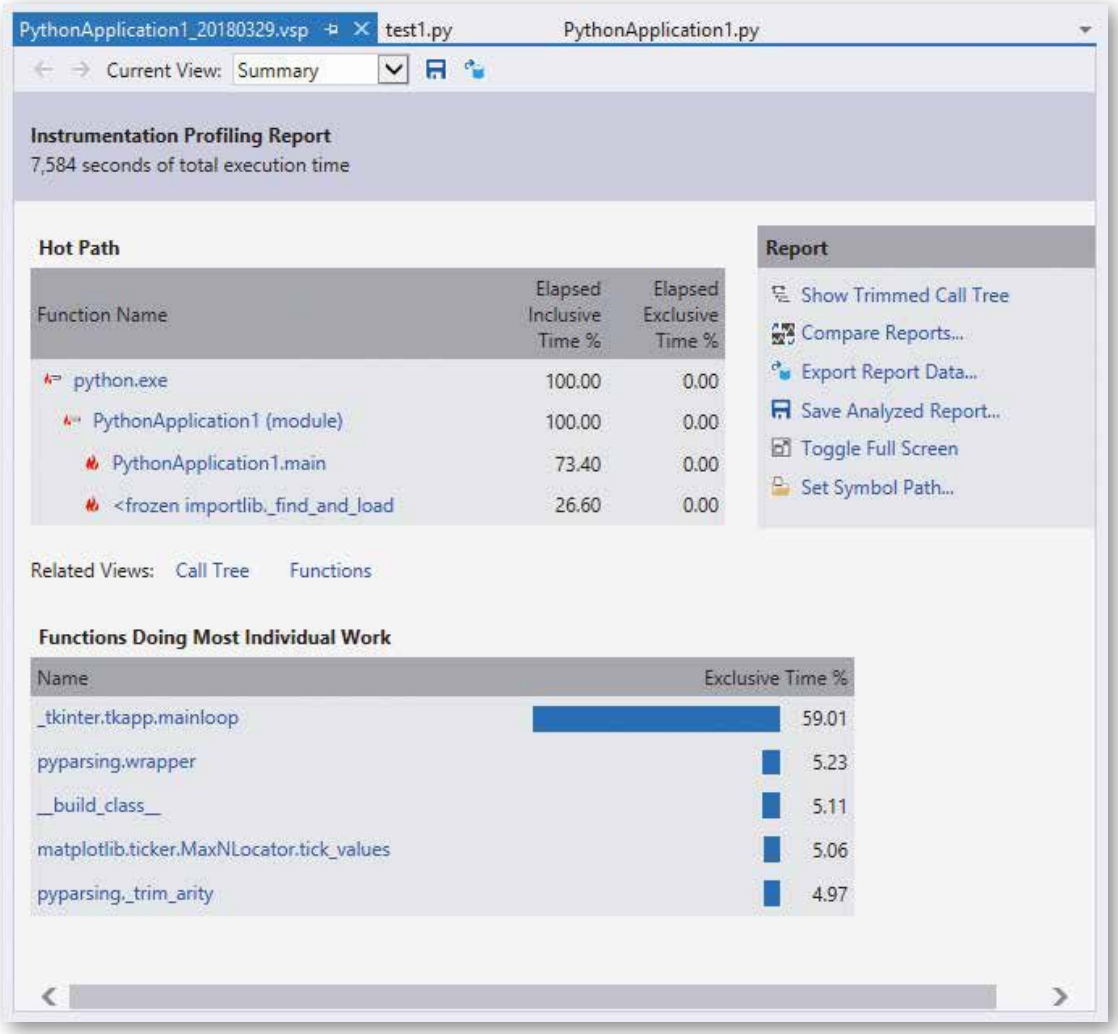



**Figure 12:** Visual Studio can profile your code. Obviously, in this case, most of the time is spent plotting and not in our code.

Now open the Test Explorer (**Test/Windows/Test Explorer**). Your project should already sit there as the root node, with test1.py as a child. Expand the child node to verify that the only test available in the code, test_A, has been discovered and is listed. If you run the test by double-clicking it in the Text Explorer, it fails, as expected given the current code.

Now go back to PythonApplication1.py and modify it to look like this:

```
from math import radians
import numpy as np  # installed with matplotlib
import matplotlib.pyplot as plt
import sys

my_var = 'CODE Magazine'  # we want to test this

def main():
    x = np.arange(0, radians(1800), radians(12))
    plt.plot(x, np.cos(x), 'b')
    plt.show()

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

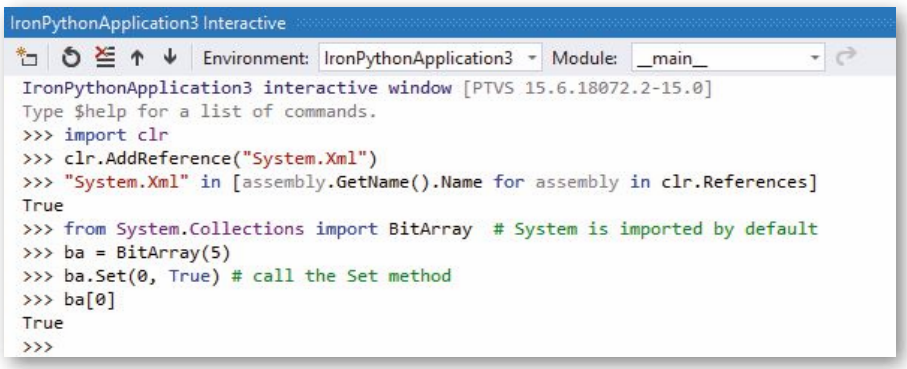



**Figure 13:** Calling .NET from Python, courtesy of IronPython

There are two additions. You added a new global variable my_var; then you added an If statement before the last line. The global variable my_var is going to be the target of your soon-to-be-updated test. That If statement does look weird though, doesn't it? When a Python script runs as the main program, all of the code that's at indentation level 0 gets executed. Functions and classes that are defined are, well, defined, but none of their code gets executed. Unlike other languages, there is no explicit main()

function that gets run automatically; implicitly, the main() function is all the code that sits at the top level.

After the edit, all the top-level code that's left is your newly added If statement and, of course, your new variable that's declared and set to a string value. The built-in variable **__name__** evaluates to the name of the current module. If a module runs as the main program, then __name__ is set to "__main__". Thus, you can test whether your script is being run directly or imported by another module by testing:

```
if __name__ == '__main__':
    # code executed only when the module
    # is run directly (not imported)
    ...
```

All that your If statement does is make sure that code in its block does not run when another module imports your script. So now when you launch test_A and test1.py imports PythonApplication1.py, you don't get the graph to show up. Notice that you have the same test in test1.py, and for the same reason. With that out of the way, let's edit the test to make it pass.

Modify test1.py to look like this:

```
import unittest
from PythonApplication1 import my_var


class Test_test1(unittest.TestCase):
    def test_A(self):
        self.assertEqual('CODE Magazine', my_var)

if __name__ == '__main__':
    unittest.main()
```

Now save all the changes (yes, you do have to save your changes manually), then run test_A and enjoy watching it go all green (**Figure 11**)! You can, of course, also debug unit tests by setting breakpoints, inspecting locals, and all the other debug features you know.

### *vFun with Visual Studio, Python, and C/C++*

Python and the C language have a strong relationship. For starters, Python core is written in C, hence its name, CPython. It's quite easy to add new C/C++ built-in modules to Python. You typically do that for performance. Remember, Python is an interpreted language, so it makes sense to write performance-critical pieces in C/C++. After all, the main reason why Python is so widely adopted in the scientific field is that it combines the excellent readability intrinsic to the language with the high performance offered by its scientific libraries, which are written in C and wrapped in Python.

In Visual Studio, you can write both Python and C++ code and, what is even more exciting, you can write extension modules for CPython. You need to have both the C++ and Python workloads installed, or you can select the **Python native development tools** option for the Python workload in the Visual Studio Installer. If you're interested in this topic, I suggest that you visit the official documentation site. Over there, you'll find an excellent tutorial on writing a C++ extension for Python.

### *Calling .NET from Python*

Besides CPython, .NET developers have another intriguing option at their disposal: IronPython. IronPython is a different implementation of Python. Like CPython, it's open source. It runs on the .NET DLR (Dynamic Language Runtime) and allows it to call .NET libraries from Python. That's right. With IronPython, you can happily consume the .NET eco-system from your Python code.

In **Figure 13**, you can see how Visual Studio offers direct support for IronPython. A few IronPython project templates are also available. You can create a new WPF or WinForms Python application. How weird is that?! Keep in mind that the official IronPython distribution is an implementation of Python 2.7. A Python 3 effort, conveniently called IronPython3, is ongoing. You can build and use it (instructions are available on GitHub), but you can't find a reference (or a download link, for that matter) to it on the official website, and for a good reason. At the time of this writing, the IronPython3 build fails on CI.

Being able to consume the .NET libraries from Python (and vice-versa!) opens for some interesting use-cases. I've seen IronPython used as a powerful, embedded script engine, for example. Got you interested? Head over to the IronPython website at http://ironpython.net for the full rundown.

## Summary

The story between Python and Windows has always been complicated. Python comes pre-installed with all Linux distributions and on MacOS, not on Windows. Until not too long ago, even installing Python on Windows was quite an uncomfortable experience. Traditionally, Windows developers prefer to work with rich, full-featured IDEs and aren't prone to leaving their comfort zone to work with a different language. These factors alone probably didn't help with making Python very popular in the Windows eco-systems, which is unfortunate considering how popular the language is on the other platforms.

Python for Visual Studio fills the gap. As .NET developers, we get a seamless installation and development experience right in our preferred development environment. On the other hand, Python folks get a fantastic development environment on Windows, even more so with the Community Edition of Visual Studio, which also supports the Python workload.

I'm a firm believer that the effort required to become a polyglot programmer is worth it. It pays big dividends in the long run. Embracing different technologies, stacks, and programming cultures made me a much better developer than I was. Like many others, to achieve that goal, I had to take the plunge and move far away from my comfort zone to become proficient with oh-so-many new things: Linux, bash, Python, open source. Today, thanks to the new workloads in Visual Studio, becoming proficient with new languages (did you know that there is a Node workload too?) is easier than ever.

You should take the chance.

Nicola Iarocci

CODE

### Learning a New Language

Python makes a great second language for .NET developers. I didn't go into language details, or differences and similarities between, say, C# and Python. If you're interested (you should be), there's a fantastic article by my friend Michael Kennedy in a previous issue of CODE Magazine (http://www.codemag.com/article/1407051).

# Prepare Visual Basic for Conversion to C#

It seems that more programmers are switching to C# from Visual Basic (VB) than ever before. As we all see when searching the Web, there are more samples in C# than there are in VB. Visual Basic is a perfectly good language for creating .NET applications, but it's just not the cool kid down the block anymore. If you currently have VB applications, and your current programmer(s) quit, you're going to be hard-pressed to replace them. Programmers don't want to keep working in what are viewed as legacy technologies. Right or wrong as that might be, you can't fight the trend.

**Paul D. Sheriff**
www.fairwaytech.com

Paul D. Sheriff is a Business Solutions Architect with Fairway Technologies, Inc. Fairway Technologies is a premier provider of expert technology consulting and software development services, helping leading firms convert requirements into top-quality results. Paul is also a Pluralsight author. Check out his videos at http://www.pluralsight.com/author/paul-sheriff.

How do you prepare for converting your VB applications to C#? I'm currently helping one company do that. They have a very large Windows Forms application that they've committed to converting to C#. In this article, I'll take you through the process my team and I have come up with to make the conversion much easier.

## Our First Steps

I started programming Visual Basic right when VB 3.0 came out. I kept programming in VB until .NET was released. I then programmed in both for a couple of years, but then switched completely to C#. Making the switch was a little painful at first because I realized how much sloppy coding Visual Basic lets you get away with. Yes, I was guilty of it back then, too. C# doesn't allow you get away with as many bad programming practices. Remembering my transition, I've come up with a list of things you can do in your VB programming to ease your transition to C# today.

Working with my current client, I took their complete VB Windows Forms application and ran it through a tool that purports to convert VB to C#. The tool, Instant C# (http://bit.ly/2rCwDL5), did a decent job on most of the VB code, but it can't convert everything if you don't write VB the correct (.NET) way. After the first run of the conversion tool, I encountered about 15,000 errors upon the first compile of the translated C# code.

As I worked my way through the errors, I compared the C# code to what was in the original VB code. From this list, I came up with the list of best practices you should employ in your VB programming to make the transition simpler. The rest of this article describes the things you should start doing immediately in your VB programs.

## The .NET CLR and Framework Class Libraries

Microsoft designed .NET on a Common Language Runtime (CLR) library. This CLR provides a set of services for all languages that wish to use .NET, such as Visual Basic, C#, F#, COBOL, etc. The services of the CLR give each language a common data-type system, automatic memory management, garbage collection, and many other features. Each language compiler takes the syntax of the language and converts the statements into the appropriate intermediate language (MSIL) that's eventually compiled into executable code.

There's a set of Framework classes that all languages have access to. For example, the Convert class has methods such as ToInt32(), ToBoolean(), etc. All languages can use these methods. VB has its own versions of some of these methods to be consistent with older versions of VB. For example, in Visual Basic you might write CInt(stringValue) to convert a string to an integer instead of using the Convert.ToInt32() method. Although these methods ultimately do the same thing, the CInt() method doesn't translate to the same MSIL because it does a lot of type checking at runtime to handle some of the quirks that VB lets you get away with.

## Visual Basic Differences

The BASIC language was developed in the 1960s, about 40 years before .NET. It has different ways of doing things compared to what is considered the "correct .NET way of doing things." It's not that these things are necessarily bad, it's just different from the way .NET, and modern compile languages, do things. Although many statements, such as CInt(), can be converted into MSIL, some statements in VB might take many more MSIL statements compared to a pure .NET method. This can cause performance issues, and introduce runtime errors, instead of catching errors during compile-time. As we all know, compile-time errors are preferable to runtime errors.

## Turn on Option Strict

One of the best things you can do right now is to turn Option Strict on in all your VB projects. Go to your project properties, click on the Compile tab, and you can turn this option on there. When you create a new VB project in Visual Studio, the default value for Option Strict is off. This is done for backward-compatibility reasons. It's recommended by Microsoft, and widely considered to be a best practice, to turn Option Strict on to increase performance and reduce the possibility of naming and runtime errors. For more information on this option in Visual Studio, read the Microsoft docs at http://bit.ly/2DwWOV8. Also, http://bit.ly/2GcT3Gd has a good reference regarding why turning on Option Strict is a good idea.

The primary reason for the large number of errors I got on the first conversion was because of Option Strict being turned off in their projects. The reason for this is that Visual Basic performs a lot of type conversions at runtime and is very forgiving when converting from one type to another. This isn't a good practice as it leads to slow performance and runtime errors. You need to eliminate all code that does automatic type conversion.

## Turn on Option Explicit

Set the Option Explicit to on in all your VB projects. The default value for this is on, but it might be good to check it. Go to your project's properties, click on the Compile tab, and you can check that this option is turned on. This option forces you to declare all variables prior to using

them. In the old days of BASIC programming, you didn't need to declare variables to use them. As we all know, compilers do much better when you declare variables with a specific data type.

## Implicit Data Type Conversions

The largest number of errors you might get from turning on Option Strict is implicit data type conversions. These implicit conversions allow the VB runtime engine to handle coercing one data type to another, instead of enforcing type-correctness at compile time. Consider the following valid code in VB.

```
Dim anyValue As Object
Dim anyString As String

anyValue = "Some value"
anyString = anyValue
```

The above code compiles just fine with **Option Strict Off**, but once you turn Option Strict On, the last line gives an error: "**Option Strict On** disallows implicit conversions from **Object** to **String**". This is because you're assigning an object to a string. If the value within the variable anyValue can't be translated into a string, you receive a runtime error. Therefore, **Option Strict On** is the best option. You don't want runtime errors, you want the compiler to warn you when you have one type being coerced into another. Consider the following code snippet.

```
Dim anyValue As Object
Dim anyString As String

anyValue = "Some value"
anyString = anyValue

anyValue = Convert.ToInt32(anyString)
```

The last line of this code causes a runtime error because the value in the variable **anyString** doesn't contain a valid integer value. Consider this next line as a replacement for the line that assigns the variable **anyValue** to "Some Value" in the last snippet. :

```
anyValue = "15"
```

If you assign a string with a numeric value like that, the code illustrated previously works perfectly. Now you understand why using implicit conversions is such a bad idea. Sometimes your code will work and sometimes it won't. These types of errors can be very difficult to debug, because the code may succeed when you input a correct value, but if your user enters some bad data, you get a runtime error.

### *Methods that Return an Object*

Many third-party Grid controls have methods that return an object data type because they don't know what you put into each cell of a grid. For example, the GetRowCellValue() method in the DevExpress Grid returns an object data type. Many VB developers just take that value and assign it directly to another variable. This means that the VB runtime has to perform the type conversion. If the value in the cell can't be converted into the target variable's type, you get a runtime error.

```
txtNotes.Text = grid1.GetRowCellValue(
     grid1.FocusedRowHandle,
     grid1.Columns("Code"))
```

Instead of the above, create a generic method that attempts to convert to the specified data type and returns a default value if the data in the grid cell can't be converted.

### *Test for Equality*

When you're testing for equality, both operands on each side of the equal sign must be the same data type. In the code below, the equal sign is testing an object against a string, which isn't valid.

```
Dim dt As DataTalbe

' Load Data Table here

For Each dr In dt.Rows
  If dr("Code") = "PP" Then ' BAD CODING
    dr.Delete()
  End If
Next
```

An easy way to fix the code above is to add the ToString() method after the dr("Code") object. This converts the object to a string, so the two string data types may be compared correctly. Like this:

```
If dr("Code").ToString() = "PP" Then
```

### *Assign SelectedValue or SelectedItem*

The SelectedValue and SelectedItem properties of a combo or list box in Windows Forms returns an object data type. Common code that I see in VB applications is where the SelectedValue from a combo or list box is assigned to a string or integer data type. The code below is what you should **not** do.

```
Dim strVal As String
Dim intVal As Integer

stringVal = cboState.SelectedValue
intVal = cboState.SelectedItem
```

> Ensure that the data within the SelectedItem object is indeed an integer or risk a runtime error.

Instead of the above code, you should use code like the following.

```
Dim strVal As String
Dim intVal As Integer

stringVal =
   cboState.SelectedValue.ToString()
intVal =
   Convert.ToInt32(cboState.SelectedItem)
```

Ideally, when converting to an integer from an object, you should ensure that the data within the SelectedItem

object is indeed an integer. Otherwise you receive a runtime error. Become familiar with the TryParse() methods on each data type to help you determine if the data can be converted.

### *Test SelectedValue and SelectedItem*

Another common practice in many VB programs is comparing SelectedValue or SelectedItem properties to some other data type. In the code below, the SelectedValue is compared directly to a string data type.

```
If combo1.SelectedValue = "GOOD" Then
  ...
End If
```

Instead of writing the above code, write the code like this:

```
If combo1.SelectedValue.ToString() = "GOOD" Then
  ...
End If
```

Remember, this doesn't necessarily protect you from a runtime error, but this code will convert to C# just fine. You should always do type-checking whenever you compare an object value to any other value.

### *Assign a Form to Specific Form Type*

It's easy to create a multiple document interface (MDI) in Windows Forms applications. When you have many child forms contained within the main form, you sometimes write code to get the currently active child form and cast it to a real form type.

```
frmCust = Me.ActiveMdiChild
```

Always perform type checking before doing this conversion, and use the CType() or DirectCast() functions to perform the conversion.

```
If Me.ActiveMdiChild Is frmCust Then
  frmCust = CType(Me.ActiveMdiChild, frmCust)
End If
```

### *Use of Split() Method*

The Split() method needs to you pass a character, or a character array, with the characters to use to split a string into substrings. The following is valid VB syntax, but some C# converters have problems converting the syntax New Char() {"*"}.

```
Dim files() As String = _
 fileNames.Substring(0, fileNames.Length - 1) _
   .Split(New Char() {","})
```

To help with C# conversion, modify the code above to create a single character variable. You can then use the following syntax instead of the above.

```
Dim delimiter As Char = ","c
Dim files() As String = fileNames.Substring(0,
   fileNames.Length - 1).Split(delimiter)
```

## Avoid Using the My Namespace

The **My** namespace exposes a set of objects to the VB programmer. Each of these objects is available through the normal .NET CLR objects. To use the My namespace in C#, you must include a VB DLL in your C# project. Avoid the usage of the My namespace so you can eliminate references to VB after you've converted to C#.

All objects within the My namespace have equivalents in the .NET Framework classes. It might take you a few more lines of code to get the same functionality, but you end up learning .NET better, and your conversion will be much easier.

It might take you a few more lines of code to get the same functionality, but you end up learning .NET better, and your conversion will be much easier.

## No Modules

A Module statement in VB defines a reference type in which you declare events, variables, properties, and procedures to be called. There's no equivalent of this in C#. The closest you could come to in C# is a class with all shared properties and methods. A module can't be instantiated as an object and can't use inheritance. This makes modules basically worthless and should be avoided at all costs. To summarize; just use classes.

## Case Sensitivity

When writing VB code, be consistent on the case of your method and variable names. Most of the time, the Visual Studio editor fixes these up, but not always. Remember, C# is a case-sensitive language, so if you don't have the case correct, a C# program won't compile. A call to a **Copy()** method is different than a call to a **copy()** method in C#, but works fine in VB. Notice the upper-case C, versus the lower-case c. These rules apply to method, property, variable, and enumeration names.

Visual Studio fixes most of the errors for you for all the .NET classes, but it won't necessarily do the same for any custom class, property, method, or enumeration names. This is especially true if you have a VB application that's very old and you've been using it for a long time. All that old VB code was probably written at a time when Visual Studio didn't automatically fix all casing of names.

As an example, the following names are invalid when translated to C#, but work perfectly fine in VB.

```
OracleDBType
Oracle.VarChar2
```

The above variable names should have been written as shown in the following bullets.

```
OracleDbType
Oracle.Varchar2
```

Notice the upper versus the lower-case b in the first name. Notice the use of the upper-case C instead of the

lower-case c used in the second name. Because these variable names came from a third-party library and were written a long time ago—or at least not fixed—and Visual Studio doesn't automatically fix them.

## Calling Methods

When calling methods such as Trim(), Copy(), ToUpper(), etc., always put the open and closing parentheses after the name of the method. Properties don't use parentheses but methods always do. No code translator out there can distinguish between what should be a method or a property if you don't follow this basic convention. For example, the following code is valid in VB, but not in C#.

```
Dim str As String = "  a lower case string  "

' Valid in VB, not C#
str = str.ToUpper.Trim  ' Missing parantheses
```

> Properties don't use parentheses but methods always do.

## Calling Shared Methods

Don't declare an instance of a class that contains Shared methods and invoke a method through that instance. Consider the following class definition and shared method.

```
Public Class CommonFunctions
  Public Shared Sub SayHello()
    MessageBox.Show("Hello")
  End Sub
End Class
```

Visual Basic allows you to create an instance of the CommonFunctions class and call the SayHello() method.

```
Dim cf As New CommonFunctions()

cf.SayHello()  ' THIS IS BAD!
```

The newer versions of Visual Studio should warn you about these types of calls, but they're just warnings. The code still compiles and works at runtime. Just call the method by referencing the name of the class CommonFunctions.SayHello().

## Reserved Word Usage

Be careful of using reserved words as variable names. This includes reserved words in both VB and in C#. In the conversion I performed for my client, there were many instances where I found declarations such as **Dim int as Integer**. In C#, "int" is a keyword and this causes issues when converting the code. The following links are a list of reserved words in both C# and VB that you should avoid.

- http://bit.ly/2mcjtQ6
- http://bit.ly/2m6WnJE

### *Eliminate Array Usage*

It's highly recommended to stop using arrays and switch to generic collections instead. Use **List<string>**, **List<integer>**, and any of the other generic collection classes. The **List<T>** collection can be used with any data type, even your own objects.

Don't use the **Redim** statement; this command doesn't exist in C#. If you switch to using collections, you won't need to use this command.

### *Avoid Exit Statements*

Don't use **Exit Try**, **Exit Sub**, and **Exit Function.** These don't exist in C# and the converters will create **goto** statements for these. Rewrite the logic in your VB method to just fall out the end of the sub or function when some condition is met.

- Instead of Exit Sub, use the Return statement with nothing after it.
- Instead of Exit Function, use the Return statement followed by some value.

The **Exit Do** and **Exit For** statements are ok to use as these statements both translate to a **break** statement in C#.

### *Hungarian Notation*

Hungarian notation has not been used in over 20 years. This naming standard was used before tools like Visual Studio allowed us to hover over a variable and see its data type. Today, just use descriptive variable names and you'll be fine.

### *Don't Use ByRef*

Ever since object-oriented programming was invented, it's rarely a good idea to pass in a parameter to a method by reference. This allows the method to modify that parameter to a different value, and the calling method now has a different value in it. These types of bugs are very hard to track down. Avoid the use of the **ByRef** statement. There are many ways to eliminate this statement's usage.

- Set properties in a class prior to calling the method and that method sets other properties.
- If you have a single **ByRef** and a **Sub**, make that **Sub** a **Function** and **Return** a value from the function.
- If you have multiple **ByRef** arguments to a method, return an object from the function with the appropriate properties set.

### *Eliminate Late Binding*

Late binding refers to having an object's type determined at runtime. There are many cases you might not think of as doing late binding but that can easily be seen once you turn on **Option Strict**. For example, if you bind a ComboBox control to a DataTable, each row in the combo box is now a DataRow object. If you iterate over the Items collection in the combo box to find where a specific column equals a certain code, you might write code like the following:

```
For i = 0 To .Items.Count - 1
  If CType(cbo.Items(i)("Code"), String) =
        CodeToFind Then
    .SelectedIndex = aIndex
    Exit For
  End If
Next
```

The Items() indexer is of the type **object** and thus using an index to reference a DataRow is considered late binding. You must cast that expression to a DataRow before you can access the field name **Code**. Replace this code with code like the following to eliminate late binding.

```
Dim dr As DataRow
For i = 0 To .Items.Count - 1
  dr = CType(.Items(i), DataRow)
  If CType(dr("Code"), String) = CodeToFind Then
    .SelectedIndex = aIndex
    Exit For
  End If
Next
```

### *String Handling*

Use a plus (+) sign for arithmetic and an ampersand (&) for string concatenation. Don't mix these two, as this can cause some unintended side effects if there are numbers contained in the string. Consider using String.Format() instead of string concatenation. This method is also more efficient than normal string concatenation.

If you are concatenating many strings together, use the **StringBuilder** class. The **StringBuilder** class is efficient because it allocates a large block of memory at one time. Strings in .NET are immutable, meaning that once you place a value in them, they are stuck at that memory location. If you add something to that string, a new area of memory must be allocated to store the old value, plus the extra characters you just added. This is an expensive operation. The **StringBuilder** pre-allocates a set of memory so there will be fewer of these operations.

### *Null Testing*

I've done many code reviews in my more than 32-year career, and I frequently see programmers checking values from a DataRow like the following:

```
Dim dr As DataRow

dr = dt.Rows(0)

If Not dr("Code") Is System.DBNull.Value
   AndAlso Not dr("Code")) Is Nothing
   AndAlso Not dr("Code") = "" Then
  TheCode = dr("Code")
Else
  TheCode = String.Empty
End If
```

There are so many problems with the above code, it's hard to know where to start. The dr("Code") object is queried four times! Accessing a value through an indexer like this is not very efficient. The last test compares an object against an empty string, so this is an implicit type conversion. The repeated use of the Not operator makes the logic very difficult to follow.

Instead of writing this type of code, create a new method in one of your classes that contains shared methods to check values against **DbNull**, **Nothing,** and empty strings.

```
Public Shared Function
  IsStringNullOrEmpty(value As Object)
      As Boolean
  Dim ret As Boolean = False

  If DBNull.Value.Equals(value) Then
    ret = True
  ElseIf String.IsNullOrEmpty(value) Then
    ret = True
  End If

  Return ret
End Function
```

You can now simplify the code you wrote earlier to compare the "Code" field value to read as follows.

```
Dim dr As DataRow
' Assign dr to a row in a data table

If Common.IsStringNullOrEmpty(dr("Code")) Then
          TheCode = String.Empty
Else
  TheCode = dr("Code")
End If
```

This is much more efficient than the previous code because the DataRow is only accessed once, it eliminates the Not operators, and the code is easier to understand.

### *Eliminate Old Visual Basic Functions*

VB has a Microsoft.VisualBasic namespace that's imported automatically in your projects. This namespace contains global functions you can use such as CLng(), CInt(), Asc(), etc. If you wish to move to C#, you must start eliminating usage of these built-in functions. Most VB to C# conversion tools translate these automatically for you, but you should get used to using the equivalent CLR objects and methods.

If a function you're calling isn't prefixed by an object, then it's a Visual Basic function and not a .NET method. In this next section, I'll take you through many of the common VB functions, and point out the equivalent .NET library methods to use instead.

## Asc

If you want to find out the ASCII number for a letter like A, the Asc() function allows you to do that. If you use this function, I suggest creating your own method to perform this conversion as there's no equivalent in the .NET Framework. Here's a method, named SafeAsc, that replaces the Asc() function.

```
Dim value As Char = "A"c

Debug.WriteLine(Asc(value))     ' Old Method
Debug.WriteLine(SafeAsc(value)) ' New Method

Private Shared Function
     SafeAsc(ByVal str As String) As Short
  Return Convert.ToInt16(
       Encoding.[Default].GetBytes(str)(0))
End Function
```

### *CBool*

The CBool() function converts a string or integer value into a Boolean true or false value. If the string contains **True** or **False**, that string is converted into the appropriate Boolean value. If it contains any other values, an exception is thrown.

If you convert an integer value into a Boolean value using CBool(), zero is converted to false, and any non-zero number is converted to true. The equivalent function in .NET is Convert.ToBoolean().

```
Dim str As String = "True"
Dim value As Integer = 33

Debug.WriteLine(CBool(str))

' The following throws an exception if 'str'
' contains a numeric value such as 0 or 1
Debug.WriteLine(Convert.ToBoolean(str))

' When using a numeric, any
' non-zero value is considered true
Debug.WriteLine(Convert.ToBoolean(value))
```

### CByte

The CByte() function takes a numeric value and converts it into a byte. The equivalent in .NET is the Convert.ToByte() method.

```
Dim value As Double = 41.8

Debug.WriteLine(CByte(value))
Debug.WriteLine(Convert.ToByte(value))
```

### CChar

The CChar() function takes a string value and converts it to a single character. The closest equivalent in .NET is the Convert.ToChar() method. However, be aware that if the string value contains more than one character, Convert.ToChar() throws an error; the CChar() does not.

```
Dim value As String = "BCD"

' The following line of code returns "B".
Debug.WriteLine(CChar(value))

' The following line will bomb
' if the string is > 1 character
' Debug.WriteLine(Convert.ToChar(aString))

value = "B"
Debug.WriteLine(Convert.ToChar(value))
```

### CDate

The CDate() function takes a string value and converts it to a Date data type. The equivalent in .NET is the Convert.ToDate() method.

```
Dim value As String = "1/1/2018"

Debug.WriteLine(CDate(value))

Debug.WriteLine(Convert.ToDateTime(value))
```

### CDbl

The CDbl() function takes a numeric value and converts it to a Double data type. The equivalent in .NET is the Convert.ToDouble() method.

```
Dim value As Integer = 42

Debug.WriteLine(CDbl(value))

Debug.WriteLine(Convert.ToDouble(value))
```

### CDec

The CDec() function takes a numeric value and converts it to a Decimal data type. The equivalent in .NET is the Convert.ToDecimal() method.

```
Dim value As Integer = 42

Debug.WriteLine(CDec(value))

Debug.WriteLine(Convert.ToDecimal(value))
```

### Chr

This function is the opposite of the Asc() function. It returns a letter for an integer value. There's no equivalent to this method in .NET, so you need to create your own function. Here's a method named SafeChr() that you can use instead of the Chr() function.

```
Private Shared Function SafeChr
  (ByVal CharCode As Integer) As String
  If CharCode > 255 Then
    Throw New ArgumentOutOfRangeException
       ("CharCode", CharCode,
        "CharCode must be between 0 and 255.")
  End If

  Return System.Text.Encoding.[Default]
            .GetString({CByte(CharCode)})
End Function
```

You can use this method as shown in the following code:

```
Dim value As Integer = 65

Debug.WriteLine(Chr(value))     ' Old Method
Debug.WriteLine(SafeChr(value)) ' New Method
```

### CInt

The CInt() function takes a numeric value and converts it to a Integer data type. The equivalent in .NET is the Convert.ToInt32() method.

```
Dim value As Double = 41.8

Debug.WriteLine(CInt(value))

Debug.WriteLine(Convert.ToInt32(value))
```

### CLng

The CLng() function takes a numeric value and converts it to a Long data type. The equivalent in .NET is the Convert.ToInt64() method.

```
Dim value As Double = 41.8

Debug.WriteLine(CLng(value))

Debug.WriteLine(Convert.ToInt64(value))
```

### CObj

The CObj() function takes a value and converts it to an Object data type. There's no equivalent in .NET except to use a cast operator. I recommend that you use the

DirectCast() function in VB because the conversion tools generally convert this to a cast in C#.

```
Dim value As Double = 41.8

Debug.WriteLine(CObj(value))

Debug.WriteLine(DirectCast(value, Object))
```

### CShort

The CShort() function takes a numeric value and converts it to a Short data type. The equivalent in .NET is the Convert.ToInt16() method.

```
Dim value As Double = 41.8

Debug.WriteLine(CShort(value))

Debug.WriteLine(Convert.ToInt16(value))
```

### CSng

The CSng() function takes a numeric value and converts it to a Single data type. The equivalent in .NET is the Convert.ToSingle() method.

```
Dim value As Double = 41.8

Debug.WriteLine(CSng(value))

Debug.WriteLine(Convert.ToSingle(value))
```

### CStr

The CStr() function takes a numeric value and converts it to a String data type. The equivalent in .NET is the Convert.ToString () method. You may also use the ToString() method on the numeric data type.

```
Dim value As Double = 41.8

Debug.WriteLine(CStr(value))

Debug.WriteLine(Convert.ToString(value))
Debug.WriteLine(value.ToString())
```

### Filter

The Filter() function in VB iterates over an array to locate a value you pass in. It then creates a subset of the original array with the items that match your value. For example, here, you create an array of three words and return a subset of that array that contains the word "is".

```
Dim values(2) As String
values(0) = "This"
values(1) = "Is"
values(2) = "It"

Dim subset() As String
' Returns ["This", "Is"].
subset = Filter(values, "is", True,
                CompareMethod.Text)
```

C# doesn't have a Filter() method, so use a LINQ query instead.

```
Dim values(2) As String
values(0) = "This"
values(1) = "Is"
values(2) = "It"

Dim subset() As String
' Returns ["This", "Is"].
subset = values.Where(Function(s)
           s.ToLower().Contains("is")).ToArray()
```

### Format

The Format() function helps you format numbers, dates, and times. The ToString() method on the Date and each of the number data types allow you to pass formatting characters to perform any formatting you want. Consider if you have the following date declared in a variable named "dt".

```
Dim dt As Date = #1/15/2018 2:04:23 PM#
```

Here, you see the Format() function and the equivalent .NET method on the Date data type to return a long time string.

```
Debug.WriteLine(Format(dt, "Long Time"))
Debug.WriteLine(dt.ToLongTimeString())
```

The next piece of code formats a date using a long date format.

```
Debug.WriteLine(Format(dt, "D"))
Debug.WriteLine(dt.ToLongDateString())
```

You can use your own formatting using both methods.

```
Debug.WriteLine(Format(dt, "dddd, MMM d yyyy"))
Debug.WriteLine(dt.ToString("dddd, MMM d yyyy"))
```

When you want to format numeric values, you have the same options.

```
Debug.WriteLine(Format(5459.4, "##,##0.00"))
Debug.WriteLine(5459.4.ToString("##,##0.00"))

Debug.WriteLine(Format(5, "0.00%"))
Debug.WriteLine(5.ToString("0.00%"))
```

### FormatCurrency

The FormatCurrency() function is used to format a numeric value as currency. Consider the following double value.

```
Dim value As Double = -4456.43
```

You can return a currency value using the FormatCurrency() or the ToString() method on the Double data type.

```
' Returns "($4,456.43)".
Debug.WriteLine(
   FormatCurrency(value, , , TriState.True,
                          TriState.True))
Debug.WriteLine(value.ToString("C"))
```

If you set the TriState enumeration to a false value, the negative sign prefixes the number instead of wrapping the number within parentheses.

```
' Returns "-$4,456.43".
Debug.WriteLine(
   FormatCurrency(value, , , TriState.False,
```

## Sample Code

Many of the code snippets in this article you can try out for yourself by downloading the sample code from the CODE Magazine page associated with this article. You may also download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select PDSA/Fairway Articles, and then select "CODE Magazine: Prepare Visual Basic for Conversion to C#" from the drop-down list.

```
                                TriState.True))
Debug.WriteLine(value.ToString("$#,##0.00"))
```

### InStr

The InStr() method returns an index number for where a string value exists in another string. The String data type has the IndexOf() method that performs this same operation. Be aware though, the InStr() function numbers its characters in the string starting with one, whereas IndexOf() numbers from zero.

```
Dim value As String = "Hello from VB"

' Returns 7
Debug.WriteLine(InStr(value, "from"))

' Returns 6
Debug.WriteLine(value.IndexOf("from"))
```

### InStrRev

The InStrRev() method returns an index number of where a string value exists in another string, but the index is from the end of the string. The String data type has the LastIndexOf() method that performs this same operation. Be aware that, the InStrRev() function numbers its characters in the string starting with one, whereas LastIndexOf() numbers from zero.

```
Dim value As String = "Hello from VB"

' Returns 7
Debug.WriteLine(InStrRev(value, "from"))

' Return 6
Debug.WriteLine(value.LastIndexOf("from"))
```

### Join

The Join() function gathers each array element and concatenates them into a single string value. By default, Join() places a space between each array element when concatenating into the string. You may specify which character you want to use by passing in a second parameter to the Join() function. The String class has a Join() method that performs the same functionality.

```
Dim values(2) As String
values(0) = "This"
values(1) = "is"
values(2) = "it."

Debug.WriteLine(Join(values))
Debug.WriteLine(String.Join(" ", values))
```

### LCase

The LCase() function converts all characters in a string to lower-case. The ToLower() method on the String data type performs the same operation.

```
Dim value As String = "HELLO WORLD"

Debug.WriteLine(LCase(value))
Debug.WriteLine(value.ToLower())
```

### Left

The Left() function extracts the specified amount of characters from the left part of a string value. The Substr() method on the String data type performs the same operation.

```
Dim value As String = "Hello World"

Debug.WriteLine(Left(value, 5))
Debug.WriteLine(value.Substring(0, 5))
```

### Len

The Len() function returns the number of characters in a string. The Length() method on the String data type performs the same operation.

```
Dim value As String = "Hello World"

Debug.WriteLine(Len(value))
Debug.WriteLine(value.Length)
```

### LTrim

The LTrim() function removes leading spaces from a string and returns a new string. The TrimStart() function on the String data type performs the same operation.

```
Dim value As String = "   Hello World"

Debug.WriteLine(LTrim(value))
Debug.WriteLine(value.TrimStart())
```

### Mid

The Mid() function extracts characters from a specific starting point in a string, and returns a new string from that point to the end of the string. Again, be aware that the VB functions are all one-based and the .NET methods are all zero-based.

```
Dim value As String = "Hello World"

Debug.WriteLine(Mid(value, 7))
Debug.WriteLine(value.Substring(6))
```

### MsgBox

The MsgBox() function displays a modal dialog the user must to respond to. The .NET equivalent in Windows Forms is the MessageBox.Show() method. Below are a few examples of MsgBox() and MessageBox.Show().

```
Dim value As String = "Hello World"

MsgBox(value)          ' VB
MessageBox.Show(value) ' .NET

' The next two lines are VB
Dim result As MsgBoxResult
Result = MsgBox(value, MsgBoxStyle.Exclamation)

' The next lines are the .NET equivalent
Dim mresult As MessageBoxResult
mresult = MessageBox.Show(value, "Caption",
                MessageBoxButton.OK,
                MessageBoxImage.Exclamation,
                MessageBoxResult.OK)
```

### Replace

The Replace() function replaces one string value to another within a string. The Replace() method on the String data type performs this same operation.

```
Dim value As String = "Hello World"

Debug.WriteLine(Replace(value, "World", "VB"))
Debug.WriteLine(value.Replace("World", "VB"))
```

### Right

The Right() function extracts the specified amount of characters from the end of a string value. The Substr() method on the String data type performs the same operation.

```
Dim value As String = "Hello World"

Debug.WriteLine(Right(value, 5))
Debug.WriteLine(value.Substring(value.Length-5))
```

### RTrim

The RTrim() function removes trailing spaces on a string and returns a new string value. The TrimEnd() function on the String data type performs the same operation.

```
Dim value As String = "Hello World   "

Debug.WriteLine(RTrim(value))
Debug.WriteLine(value.TrimEnd())
```

### Space

The Space() function creates a string variable with a specified amount of spaces in it. The PadLeft() method on the String data type performs the same operation.

```
Dim value As String

value = Space(50)

value = value.PadLeft(50, " "c)
```

### Split

The Split() function converts a string into an array by looking for spaces within the string. Each word within the string is converted into an element of the resulting array. You may optionally specify the delimiter to use to separate each word in the string. The .NET equivalent to use is the Split() method on the String data type.

```
Dim value As String = "This is it."
Dim values As String()

values = Split(value)
values = value.Split(" "c)
```

### StrDup

The StrDup() function takes a character value and duplicates that value a specified amount of times. You can also pass in a character and number to the String() constructor to perform this same operation.

```
' Returns "PPPPP"
Debug.WriteLine(StrDup(5, "P"))
Debug.WriteLine(New String("P"c, 5))
```

### StrReverse

The StrReverse() method reverses all of the characters in a string and returns a new string. There is no .NET equivalent for reversing a string, but you can write a function like the ReverseString() function shown below.

```
Public Shared Function
     ReverseString(value As String) As String
  Dim chrs() As Char

  chrs = value.ToCharArray()
  Array.Reverse(chrs)

  Return String.Join(" ", New String(chrs))
End Function
```

You can then use either of these to perform the same operation.

```
Dim value As String = "Hello World"

Debug.WriteLine(StrReverse(value))
Debug.WriteLine(ReverseString(value))
```

### Trim

The Trim() function removes both leading and trailing spaces from a string. The Trim() function on the String data type performs the same operation.

```
Dim value As String = "    Hello World   "

Debug.WriteLine(Trim(value))
Debug.WriteLine(value.Trim())
```

### UCase

The UCase() function converts all characters in a string to upper-case. The ToUpper () method on the String data type performs the same operation.

```
Dim value As String = "hello world"

Debug.WriteLine(UCase(value))
Debug.WriteLine(value.ToUpper())
```

### vbCrLf and vbNewLine

Avoid using the vbCrLf and vbNewLine and use Environment.NewLine instead. The vbCrLf and vbNewLine are not part of .NET, but, are VB-specific. If you perform a conversion to C#, these will need to be rewritten.

```
Dim value As String =
     "Hello World" & vbNewLine
     & "Hello VB" & vbCrLf
     & "Hello To You"

Dim newValue As String =
   "Hello World" & Environment.NewLine
   & "Hello VB" & Environment.NewLine
   & "Hello To You"

Debug.WriteLine(value)
Debug.WriteLine(" ")
Debug.WriteLine(newValue)
```

## Approaching Your Conversion Process

Going through your whole application and finding all instances of the items pointed out in this article could take a lot of time. Time that you probably don't have. There are some tools that may be able to help you.

### *Turn Option Strict On*

The first thing you can do is simply turn on **Option Strict** in each of your projects. You should get a series of error and warning messages in the Error List window. Work your way through these errors one by one. If you only have one hour a day to devote to this endeavor, you can always turn the **Option Strict** option on, work on some errors, then turn it back off so you can check everything back into your source control. The project will still compile. This also avoids errors happening for the other developers working on this project.

> Even if you only commit an hour a day to working through errors, it's worth the trouble.

### *Upgrade to Visual Studio 2017*

Each version of Visual Studio gets better at helping detect errors and providing hints on how to refactor your code. Visual Studio 2017 is the latest version at the time of the writing of this article, and provides tools built right into the editor. If you bring up your old VB code in the VS 2017 editor, you might see some squiggles under some code that needs to be fixed. The compiler has improved dramatically over the previous versions so more errors and warnings are reported.

### *Code Analysis*

Select **Build** > **Run Code Analysis on Solution** to see any warnings or messages that appear in the Error List window in VS 2015 and later. This tool, like the compiler, helps you identify areas to fix that to make your conversion to C# an easier process.

### *FxCop*

This tool helps you maintain consistency in your code. The default rules in FxCop check for potential design flaws, incorrect casing, cross-language keyword collisions, and many other problems related to naming. In addition, FxCop tells you about performance, usage, and security problems. This tool is available free from Microsoft and I highly recommend that you integrate it into your build process.

## VB to C# Conversion

After you have made all the changes you can to your VB application, start trying to convert to C#. If you have a solution comprised of many different projects, start with a single project at a time. You can perform the conversion, replace the old VB project with the new C# project, and ensure that it works as expected. You can then continue this process until all your projects are converted. If there's no hurry to do this conversion process, you might tackle one project, test it out over a week, a month, or two months before you move on to the next project. If time is of the essence, by following the guidance I've provided in this article, the conversion should go quickly.

There are a few online code converters that are free to use, such as the Telerik Code Converter located at http://converter.telerik.com. This tool converts a single class or method at a time. The code converter at Developer Fusion, http://www.developerfusion.com/tools/convert/vb-to-csharp also converts a single class or method at a time.

If you're looking for a converter that tackles a complete project and/or solution at one time, check out the Instant C# tool by Tangible Software Solutions. You can see this tool at http://bit.ly/2rCwDL5. For a very reasonable price, it converts your complete Windows Forms or Web Forms application to C#. It was by running this code conversion tool that I was able to identify the problem areas I pointed out in this article. To be fair, the other converters I mentioned found the same problems.

The solution I was converting had about one million lines of code spread out over 23 projects. Instant C# took approximately 35 minutes to perform the conversion of these projects. After the first conversion and receiving the 15,000 errors, I decided it would be easier for my client to fix their VB code first. They are more familiar with VB and can make the changes following this article's guidelines easier in VB than in C#. In addition, they get more familiar with the .NET methods they're going to be using in C#.

## Summary

Converting from one language to another is not an easy process. Realize that it is a long journey, and approach it as a long-term strategy rather than a short-term process. Even with good conversion tools, like the ones I pointed out in this article, there's no perfect conversion. You're going to have lines of code that won't convert correctly. However, that number can be greatly reduced by following the techniques outlined in this article. Good luck on your conversion!

Paul D. Sheriff

(Continued from 74)

with that, pun intended), but to account for it in both our pre-decision analysis and our post-decision analysis.

It begins with the realization of several things that readers of this column have heard me talk about before: Understanding that our brains are wired to believe in confirmation bias (viewing outcomes in a light favorable to ourselves). There's the classic "causation vs. correlation" argument that comes into play here, as well: merely because a particular decision was made and a particular result emerged doesn't mean that the one caused the other, only that both happened. (The old joke about a man in Florida selling Genuine Polar Bear Repellent comes to mind. When challenged by a passer-by, the salesman replies, "I've been using it for twenty years and never once has a Polar Bear been seen anywhere in my vicinity." Although, given current climate change rates, that joke may not be funny in about fifty years, either.)

> What makes a decision great is not that it has a great outcome. A great decision is the result of a good process, and that process must include an attempt to accurately represent our own state of knowledge. That state of knowledge, in turn, is some variation of "I'm not sure." … [A]cknowledging uncertainty is the first step in executing on our goal to get closer to what is objectively true. To do this, we need to stop treating "I don't know" and "I'm not sure" like strings of dirty words.

## Embracing Incomplete Knowledge

Unless you happen to be a deity of some sort, it's only a matter of time before you're presented with the need to provide an answer without having complete knowledge—the decision to purchase a warranty on a new appliance, for example, or the decision around which new programming language to learn to better your chances of obtaining a better job. This is a bet, no different than being dealt five cards at a poker table. Information will be hidden from you, and no amount of study or analysis will ever yield that information. At the poker table, you might be able to track the cards that have already been used—the card-counting technique—but most poker tournaments use many decks together in order to minimize the benefit that knowledge of history might provide. At the appliance counter, you might look at reliability reports for this brand of appliance, but that still in no way means you won't be the one-in-a-million that gets the one that breaks all the time. (Fortunately, you should be safe—that fate seems reserved for my wife and me.)

All of which means that if you're to be comfortable with taking bets, you need to embrace the idea that you simply don't know. As Ms Duke puts it:

"Because now, once we know that we don't have perfect information, and once we accept, deep down, that we will never have perfect information—no matter how many prototypes we build or how many benchmarks we run—then we can begin to perform the necessary analysis of figuring out how unsure we really are."

## Betting on the Team

Recently, my management at Smartsheet asked me to build out an Engineering team that would stake out some new ground within the company. We'll be using a new platform (React and Node, instead of Java), and we'll be looking to hire younger, less-senior developers so that we can staff up the team quickly. Within two months, I had a team of seven people, five of whom were relatively brand-new to the industry. One of my management peers looked at me with a high degree of skepticism one day, and said, "How do you know that these people are all going to work out?"

To which I responded, "I don't know. I'm just betting on it." The team lead, the recruiting team, and I, we did what research we could, we looked to establish what knowledge we had, and then, we placed the bet.

Ted Neward



CODE

Jul/Aug 2018
Volume 19 Issue 4

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers In This Issue
Anthony Chu
Ahson A. Khan
Sahil Malik
John V. Petersen
Paul D. Sheriff
Nicola Iarocci
Julia Lerman
Ted Neward
Daniel Roth

Technical Reviewers
Markus Egger
Rod Paddock

Production
Franz Wimmer
King Laurin GmbH
39057 St. Michael/Eppan, Italy

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

Circulation & Distribution
General Circulation: EPS Software Corp.
International Bonded Couriers (IBC)
Newsstand: Ingram Periodicals, Inc.
Media Solutions

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 300, Spring, Texas 77379
Phone: 832-717-4445
Fax: 832-717-4460

# On Bets

For an industry that prides itself on its analytical ability and abstract mental processing, we often don't do a great job applying that mental skill to the most important element of the programmer's tool chest—that is, ourselves.

"So, boss, you've heard his idea and you've heard mine, and we've gone through why we each think our idea is the better approach to take. What do you want to do?"

Although it seems to be self-evident, it still comes as a surprise to many new managers that they will be called upon to make a decision. Sure, it's often true that the team can figure out a number of things on their own, but sometimes, the team can't come to a unified decision. Sometimes they require some sort of guidance to deal with a thorny situation. Sometimes it's the new manager's boss who demands a decision—which candidate to hire, or (far worse) which employee to fire.

It's a frightening position to be in: upon your words, your decision, the company will undertake some action, and there's no "going back" after that. A fired employee will not come back if that turns out to be the wrong choice, and you can never recover the time, money, or energy invested into a project that turns out to be an economic dead-end or technical quagmire. As they say in the movies, "It's all on you." Your decision, your fault. Sure, you're more than happy to claim the credit for the decision when it turns out to be the right one, but the fear associated with making the wrong choice? It can be crippling.

That's part of the reason that so many managers prefer to pass the decision up the management chain, and have their boss make the call. And it's why so many middle managers appear to have no power or authority whatsoever.

## Management and Poker

Annie Duke is a pretty good poker player. She's won a number of televised tournaments and as a result, she's made quite a reputation for herself in her ability to judge a good bet from a bad one. But much of that reputation doesn't derive from her success at the poker table. Instead, she's taken up providing advice and consulting, much of it in the form of seminars and books, to financiers, traders, strategic planners, human resources groups, entrepreneurs, and more. Annie has, through exposure at the poker table, figured out that many, if not most, decisions are actually bets: decisions about outcomes in the face of an uncertain future.

Each time managers are presented with a decision to make, they essentially do the same mental exercise that the poker player does when presented with a set of cards and a pot in the center of the table: They're trying to peer through the fog of the future by examining the present and make a choice today that will hopefully yield a positive outcome tomorrow (or in a few minutes, when the cards are all on the table).

Poker players have to make these decisions hundreds, if not thousands, of times in a given setting, and usually have nothing more than a few minutes' time in which to make them. You don't succeed at poker if you're bad at making those decisions, and you don't succeed at management, either.

But let's begin the analysis of betting-as-decision-making by examining what many people (particularly my neighbors here in Seattle) consider the worst play call of all time: Pete Carroll's decision to throw a pass in the waning seconds of Super Bowl XLIX at the New England Patriots' goal line. For those sports fans who missed that one, Carroll's choice to throw, instead of hand the ball off to Marshawn Lynch, one of the best running backs in the league that year, turned out to be a bad call, as the pass was intercepted by the Patriots and the Seahawks lost their chance at a second championship.

After the game was over, the popular media roasted Carroll, using headlines like "Worst Play-Call in History." Certainly, it was the least-desirable outcome, to be sure. But Carroll found a few outlying supporters in people who examined the decision on its own merits, rather than by the outcome: FiveThirtyEight.com, for example, noted that the pass, had it fallen incomplete, would have stopped the clock and allowed for an additional play, whereas the run could have burned up all the time remaining. Across the history of the league that year, that play—a pass at the two-yard-line—had been attempted sixty-six times and never once intercepted. In other words, by looking only at the data that Carroll had at the time, the pass was actually a very solid bet. So why all the criticism and heat?

As Ms Duke puts it, "We can sum it up in four words: The play didn't work."

When a bet is made, we often judge the decision to bet one way or another by reviewing the outcome after the bet's resolution has been discovered, rather than looking at the data at the time of the bet. Imagine this: Had the play succeeded, the Seahawks would've been heralded as "amazing" and Carroll's play-calling as "brilliant." But this would be an equally unfair assessment of the bet, for the same reason—to judge the quality of the decision by only examining its outcome is to essentially relegate decision-making to be an entirely backward-looking exercise and leaves us with no ability to determine how to make better bets.

"But isn't that the point?" some will argue. "Isn't the whole point of a decision process to figure out which is the right decision to make, and then make that decision?" Sure, if life were a game of chess, where all of the factors involved in making a decision are right there, on the board, easily visible to anyone familiar with the game. Chess is a game of perfect information—neither player has any more information about the state of the game than the other. Alas, life gives us no such clues or hints. Much of the information we might want or need to make a decision is hidden from us, just as in poker.

Every decision will be affected by two factors: skill and luck. (Some will argue that there's no such thing as luck, that all of the factors could be discovered, identified, traced, and accounted for, but usually we don't have that kind of time, so let's leave all of those un-accounted factors under the general umbrella of "luck".) In the Super Bowl, certainly, skill was involved when the defender made the catch, and when the defender was coached to recognize the play call, and when the Patriots coach studied the film of previous Seahawks games so that he could identify this play as one that the Seahawks might run. But luck takes a hand here, as well—the defender could've missed the cue that this was the upcoming play because his attention was on a different part of the formation, or even a stray arm could have deflected the ball away from its intended flight path. Luck is always present in these sorts of decisions, and our goal as decision-makers is not to try to eliminate luck entirely (good luck

*(Continued on page 73)*

**VENDORS: ADD A REVENUE STREAM BY OFFERING ESCROW TO YOUR CUSTOMERS!**

# CAN YOU TAKE THE RISK ?

Less than $1 per day!

## Affordable High-Tech Digital Escrow

Tower 48 is the most advanced and affordable digital escrow solution available. Designed and built specifically for software and other digital assets, Tower 48 makes escrow inexpensive and hassle free. Better yet, as a vendor, you can turn escrow into a service you offer to your customers and create a new revenue stream for yourself.

Regardless of whether you are a vendor who wants to offer this service to their customers, or whether you are a customer looking for extra protection, visit our web site to start a free and hassle-free trial account or to learn more about our services and digital escrow in general!

Visit **www.Tower48.com** for more information!

TOWER 48