Text, Text, Text

CODE

MAY
JUN
2024

CODE 30 YEARS

Title

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

Cover AI generated - Markus Egger

Subtitle

Subtitle

Subtitle

CODE CONSULTING

ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN BENEFIT YOU TODAY?

©shutterstock

EXECUTIVE BRIEFINGS

Are you wondering how AI can help your business? Do you worry about privacy or regulatory issues stopping you from using AI to its fullest? We have the answers! Our Executive Briefings provide guidance and concrete advise that help decision makers move forward in this rapidly changing Age of Artificial Intelligence and Copilots!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

AI-SEARCHABLE KNOWLEDGEBASE AND DOCUMENTS

A great first step into the world of Generative Artificial Intelligence, Large Language Models (LLMs), and GPT is to create an AI that provides your staff or clients access to your institutional knowledge, documentation, and data through an AI-searchable knowledgebase. We can help you implement a first system in a matter of days in a fashion that is secure and individualized to each user. Your data remains yours! Answers provided by the AI are grounded in your own information and is thus correct and applicable.

COPILOTS FOR YOUR OWN APPS

Applications without Copilots are now legacy!

But fear not! We can help you build Copilot features into your applications in a secure and integrated fashion.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/ai-services

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

Features

Departments

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

LEAD Tools

Rod Paddock
CODE

CODE: 20 Years Ago

The end of 2023 was the start of our "30 years of CODE" celebration year, which will continue throughout all of 2024. To look back at those 30 years, I wrote articles in the last two issues of CODE Magazine, looking at what happened 25 and 30 years ago. This time, I'll look back 20 years and explore the latest and greatest of the early-to-mid 2000s. I remember it as a somewhat turbulent time. The dotcom bubble had burst, and the glory days of technology seemed to be over. Was the internet really all it was supposed to be or was it just a passing fad? It was hard to say.

The Aftermath of the Dotcom Bubble

One of the main catalysts for the dot-com-bubble bursting was the overvaluation of many internet-based companies that had little or no profits but huge expectations. Investors poured money into these ventures, hoping to cash in on the next big thing, but many of them turned out to be unsustainable or unprofitable. Some of the most notorious examples of dotcom failures were Pets.com, Webvan, eToys, and Boo.com, which burned through millions of dollars in a matter of months before going bankrupt. The collapse of these and other companies sent shockwaves through the stock market, wiping out billions of dollars in value and causing many investors to lose confidence in the sector.

The world of software development wasn't immune to the effects of the dotcom bubble bursting. Many software developers who'd been hired by dotcom startups found themselves out of work when their employers went under. Some of them had to accept lower salaries or switch careers, and others tried to start their own businesses or join more established companies. The demand for web development skills decreased, as many companies scaled back or canceled their online projects. The failure of many dotcoms also raised questions about the viability and quality of some of the emerging web technologies and standards, such as Java, XML, and HTML. Some critics argued that these technologies were overhyped and underdelivered, and that they weren't suitable for building complex and reliable applications. Others defended these technologies and claimed that they were still evolving and improving, and that they would eventually prove their worth.

Despite the challenges and setbacks that the dotcom bubble bursting posed for the software industry, it also had some positive effects. It forced many companies to rethink their business models and strategies, and to focus more on customer needs and satisfaction, rather than on growth and hype. It encouraged more innovation and experimentation as some developers sought to create new and better solutions for the web. It also paved the way for the emergence of new players and platforms, such as Google, Amazon, eBay, and PayPal, which took advantage of the opportunities and gaps in the market that the dotcom crash had left behind. These companies would go on to become some of the most successful and influential in the history of the internet and to shape the future of software development.

The Impact for CODE

Luckily for us at the CODE Group, the dotcom turbulences were less severe than for other companies. Most of the projects we were working on in the consulting and custom app dev side of the business weren't classic dotcom companies. Also, we'd started CODE Magazine in the Spring of 2000 and focused primarily on the new world of software development that Microsoft was generating. The Java programming language was of interest to a lot of people but had some issues that were, as of then, unaddressed, and one way to fix it was Microsoft's approach of re-inventing the language in a top-secret project headed up by language-guru Anders Hejlsberg, codenamed "Cool." (This became C#, and yes, C# is still cool. You may have seen the T-shirt).

C# became a key component of the then nascent .NET ecosystem, which did away with the concept of the programming language driving everything and instead created a development framework that could be used equally from various languages. This was a concept that jived very much with what we believed a modern software development magazine should be talking about, and thus CODE Magazine found itself in a sweet spot of sorts. Other magazines, like Visual Basic Programmer's Journal, FoxPro Advisor, and many more, suddenly didn't look so hot anymore. A lot of this wasn't a coincidence. After all, we'd long been partnering very closely with Microsoft—I worked for the Visual Studio team as a contractor on various projects—and we were strong believers in these new concepts.

All these goings-on meant that we were somewhat protected from the dotcom mess. Yes, we also lost some customers, and the pool of potential new customers shrank. We had to tighten our belts a bit, but overall, we came through it all reasonably well. I remember it as a time that was painful for us, but not to an existential level. And despite all the internet disillusionment, we remained stout believers that it wasn't the internet that was the problem but rather the problem was the idea that the internet made economic fundamentals obsolete. In other words, we considered it crucially important to push forward with internet-related technologies. As a Microsoft-focused organization (and a Microsoft partner), this meant mainly focusing on ASP.NET as the backbone of almost all web applications that we wrote. We had largely ignored earlier versions of ASP, but then there was this young kid of a program manager straight out of college with a vision of a better web development environment. I was very impressed with his early demos. He was a funny and rather likable kid, and he always wore red shirts. His name was Scott someone or another. I think he still works at Microsoft today. <grin>

And before you ask, most of the web applications we wrote in those days were mainly built for Internet Explorer, the *de facto* standard browser of the time. Netscape had faded in importance as they lost the "great browser wars" against Microsoft, and Firefox wasn't a thing yet.

Markus Egger
megger@codemag.com

Markus, the dynamic founder of CODE Group and CODE Magazine's publisher, is a celebrated Microsoft RD (Regional Director) and multi-award-winning MVP (Most Valuable Professional). A prolific coder, he's influenced and advised top Fortune 500 companies and has been a Microsoft contractor, including on the Visual Studio team. Globally recognized as a speaker and author, Markus's expertise spans Artificial Intelligence (AI), .NET, client-side web, and cloud development, focusing on user interfaces, productivity, and maintainable systems. Away from work, he's an enthusiastic windsurfer, scuba diver, ice hockey player, golfer, and globetrotter, who loves gaming on PC or Xbox during rainy days.

Internet Explorer had taken the web and HTML from being a simple mechanism to displaying hyperlinked text and simple documents that supported only laughable levels of visual design, to a far more functional user interface technology. I vividly remember sitting in some internet meetings at Microsoft (this was pre-open-source and being part of these secret closed-door meetings was a big thing for us geeks) where the idea of a DOM (Document Object Model) was first discussed. How awesome would it be to be able to interact with elements on a page using scripting technologies? Mind-boggling! (JavaScript was already a thing back then, of course, but VBScript was considered equally viable by many). In hindsight, it's now easy to blame Microsoft for having created many non-standard-compliant HTML problems, but back in the day, there were no standards and Microsoft had taken it upon themselves to push things forward as fast as possible. Although I had to suffer many of the later problems this caused (and I assume, so did you), I still think it was the right thing to do at the time, and I give them credit for it.

Other Important Tech

.NET, and Visual Studio were super important technologies for all professional developers. Yes, Linux was also important, but when you worked with enterprise customers, Microsoft was where the lucrative projects were. (Some Linux enthusiasts may disagree, and I don't want to take anything away from them, but we weren't successful in making money with Linux in those days.) Microsoft's anti-trust lawsuit had come to an end, and although it had an impact on how Microsoft had to operate for quite some time to come, it did solidify Microsoft's position and the company represented a very solid and steady bet for most enterprises around the world. When Microsoft pushed out technology in the early 2000s, you could assume it was going to be not just important and successful in the market, but also a steady horse to bet on for a while. I remember consulting around quite a few Microsoft technologies back then, and they were all solid investments. I never worried that the time we spent becoming experts in one Microsoft technology or another would be wasted. (In future installments of this series, I will take a much different look at this aspect of Microsoft.)

Operating systems were a big thing 20 years ago. Microsoft's Windows XP is still one of the most liked versions of Windows (**Figure 1**). It was Microsoft's first departure from the "battleship gray" user interface design and into a more colorful world. Most of us will fondly remember the default background image of some very green pastures, overlaid with visual elements that featured far more colors than in the past. I remember that some of our customers thought it looked "like a candy store" and it took them a while to get used to it. Some even used Windows NT as client operating systems because of it. Ultimately, Windows XP did become a fan favorite and a very good operating system for its time.

It may have seemed like that to a lot of us back then, but it wasn't just a Microsoft world when it came to operating systems. Linux was always around and important in certain scenarios. But there also was this niche operating system called **Mac OS X**. 20 years ago, I thought it was neat. After all, it had been born out of the very geeky-cool **NeXT Step** operating system, developed by the NeXT company founded by Steve Jobs, and later integrated into Apple with Steve Job's return. It didn't yet play a big role overall, and most people would never have considered buying a Mac. We used Macs in our magazine department, but generally, it seemed like something that wasn't very important to the business world, and it was almost non-existent in our software development considerations. (Another topic I'll have to revisit in the next articles in this series.)

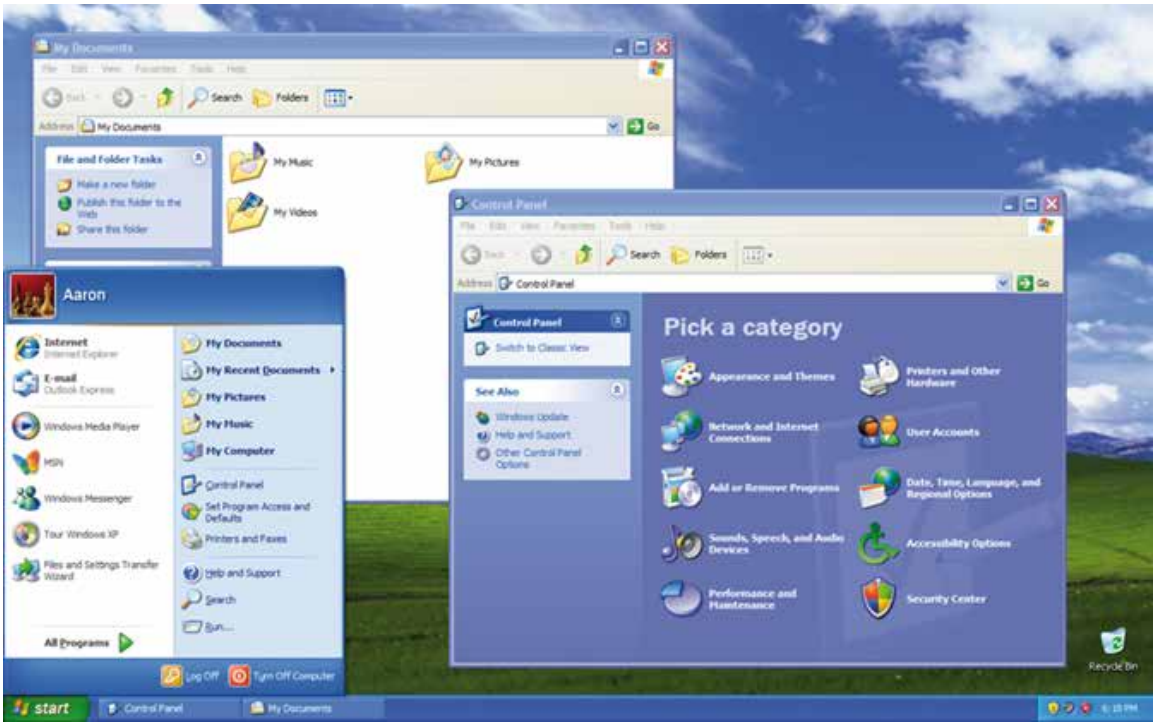

Figure 1: A typical Windows XP screen shot

Although we built a lot of web-based applications even 20 years ago, it should also be mentioned that, for most businesses, it was a "thick client" world. Many business applications were built as WinForms applications, because cross-platform deployment wasn't a big consideration for business applications. After all, why worry about users on a Mac when nobody in business used Macs? Therefore, WinForms applications, and later, WPF applications, were a very important market segment. (This isn't talked about very much anymore, but people are often surprised when I tell them how many companies are still building Rich Windows Client Apps even today.)

Apple was also not a real player in mobile computing yet. Yes, Apple had the Newton years earlier, but that was ahead of its time and was soon discontinued. Palm Pilots were also a thing of the past. But RIM's (Research-In-Motion) BlackBerry was all the rage for mobile enthusiasts (**Figure 2**). It may have later gotten Hilary Clinton into trouble as her email device of choice, but it was the state-of-the-art mobile business solution for quite some time. It seems quaint today, but it was considered unthinkable that a device without a physical keyboard could be feasible in business scenarios. This was an idea that Microsoft CEO Steve Balmer held onto way too long, in the process killing Microsoft's phone business. Today, most people don't even remember that Microsoft had a strong position in that market segment, with Windows Mobile and Windows CE.

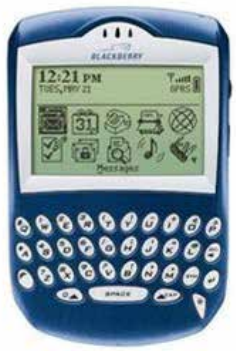

Figure 2: The BlackBerry 6000, released in 2003

Figure 3: The first Xbox was released for the Christmas 2001 season.

The Fun Stuff

The early 2000s were also a fun time. After all, Microsoft released the first Xbox (**Figure 3**). What a machine that was! And the games it had! It may not have immediately been a strong competitor for the already established PlayStation system, but it sure put out some classics. *Halo, Combat Evolved* (**Figure 4**) was groundbreaking. It finally made first-person shooters truly work on consoles and brought such experiences into living rooms. But there were also others. For me, The *Elder Scrolls III: Morrowind* and *Star Wars: Knights of the Old Republic* stand out as particular time sinks.

It's hard to believe that many of today's (and yesterday's) biggest brands in computer gaming date back to this era, whether you do your gaming on the Xbox, PC, PlayStation, or elsewhere. I very fondly remember classics such as *Grand Theft Auto III*, *Max Payne*, *Warcraft III*, *Deus Ex*, *Half-Life 2*, *The Sims*, *God of War*, and *Metal Gear Solid II*, to name just a few. Would you have guessed that *World of Warcraft* is now 20 years old? It seems that many of the games from back then are still on my "to be played one of these days soon" list, and not because they are classics, but just because I haven't quite gotten to them yet.

Although gaming went through a great period, I wasn't as excited about music back then. Chart toppers from 2003 and 2004 really don't resonate with me as true classics. I guess 50 Cent would disagree and rate *In Da Club* as one of the all-time great songs, but I'm not rushing out to buy a remix. The whole music industry took a blow from the Napster era and the Apple iPod, released in 2001, hadn't reinvigorated things yet. Or maybe I just wasn't that into that kind of music.

I did enjoy going to the movies back then though. There were some great ones. *Lord of the Rings* is at the top of my favorites from 2001 to 2003. Ah... they just don't make them like that anymore. Or actually, they do: This was also the start of the superhero movies, an era in which we are still stuck, it seems. Additionally, *Pirates of the Caribbean* resonated with people. *The Passion of the Christ* was released back then, and so was *Finding Nemo*. *Star Wars Episodes I* through *III* also fall into this time period. Yeah. I know. We shall not talk about that.

Figure 4: Halo: Combat Evolved

Markus Egger
CODE

Async Programming in JavaScript

Some of us like to brag about how old we are because we started working on languages like Assembler or Fortran or Basic. This was a while ago, when computers were very simple, and although those languages were extremely cryptic and they made us productive, it's also a reality that we were doing a lot less with them. We were building much shorter buildings. As time moved on, computers became a lot more powerful, and our customers demanded that we build skyscrapers. You might have heard of something called Moore's law, which is the observation that the number of transistors on an integrated circuit will double every two years with a minimal rise in cost. Yes, computers have become very powerful, but just due to basic physics and energy density issues, we've also hit a wall in absolute computing power that we can pack within a single processor. As a result, the world started moving toward multiple cores, and multiple processors; even your phone—or maybe even your watch—now has multiple cores or multiple processors inside it.

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.

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

When these multiple cores or multiple processes try to work together, adding two processors doesn't always equal 2X the performance. Sometimes it can even be less than 1X because the competing processors might be working against each other. For sure, the benefit you get will be less than 2X because some overhead is spent on coordination. Now imagine if you have a 64-core processor, how would that look? And how would you write code for it? There will always be that one really smart guy on your team that understands the difference between mutexes and semaphores, and that smart guy will act like the cow that gives one can of milk and tips over two. His smarts will make rest of the team unproductive because, let's be honest, these concepts can be hard to understand, harder to write, and very hard to debug.

Although it's no surprise that as we're building more complex software, our platforms and languages have also evolved to help us deal with this complexity, so the entire team of mere mortals is productive. Languages have also evolved to support more complex paradigms, and JavaScript is no exception.

In this article, I'm going to explore a back-to-basics approach by explaining asynchronous programming in JavaScript. Let's get started.

A Little Pretext

Before I get started, there's a little challenge I must deal with. Demonstrating asynchronous concepts through text and images as they appear in this article can be difficult. So I'm going to describe the various concepts, but you should also grab the associated code for this article at the following URL: https://github.com/maliksahil/asyncjavascript. I recommend running the code side-by-side as you read this article as that will help cement the concepts.

Let me first start by describing the code you've cloned here.

Code Structure

The code you've cloned is a simple nodejs project. It uses ExpressJS to serve a static website from the **public** folder, as can be seen in **Figure 1**.

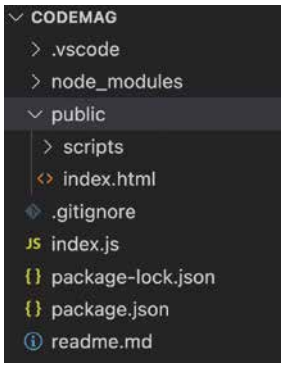

Figure 1: The project structure

To run it, just follow the instructions in readme.md. At a high level, it's a matter of running npm install, and hitting F5 in VSCode. Additionally, you'll see that in index.js as seen in **Listing 1**, in addition to serving the public folder as a static website, I'm also exposing an API at the "/random" URL. This API is quite simple; it waits for five seconds and returns a random number. I have a wait here to demonstrate what effect blocking processes, such as this wait, can have your browser's UI. The reason I've written this code in NodeJS is because I could use identical code for the wait on both client and server, although this isn't a hard requirement.

Let's also briefly examine the client side code. The index.html file is quite simple. It references jQuery to help simplify some of the JavaScript I'll write. It has a button called btnRandom that calls a JavaScript. It has a div called "output" where the JavaScript can show messages to communicate to the user. The idea is that I'll call a function that blocks for five seconds, and I'll show a "start" and the random number output message when the function starts and then when it's done.

Additionally, I've placed a text area where users can type freely. The function takes five seconds to complete, so what I'd like you to try is, within those five seconds, try to type in that text area. If you can type in that text area while the function is executing, that's a non-blocking UI, which is a good user interface. But if the UI is frozen and you cannot type in that textbox while your function runs, that's a bad user experience.

The user interface of my simple HTML file looks like **Figure 2**. The index.html file can be seen in **Listing 2**.

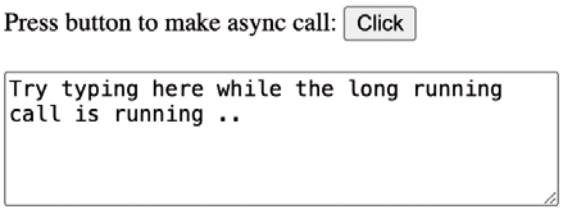

Figure 2: The user interface

A Synchronous Call

Let's first start by writing a simple JavaScript function that takes five seconds to execute. At the end of five seconds, it simply returns a random number. This function is basically the same function you see in index.js called "waitForMilliSeconds", except that for now, I'll just run it client side, and the function itself will return a random number.

The function is called on the click event of the button you see in **Figure 2**. As soon as the user clicks on the button,

it shows a "Start" message in the output div. Then the function runs for five seconds and five seconds later, you should see a random number shown in the output div. The code for the synchronous call is refenced in index.html as **scripts/1.sync.js** and can be seen in **Listing 3**.

Go ahead and run npm start (or F5 in VSCode) and visit the browser at http://localhost:3000. Click the "Click" button from **Figure 2**, and immediately try typing in the text area below. What do you see?

You'll notice that until the call completes and the random number is shown in the output div, the page is essentially frozen. It accepts no input from the user. In fact, the page is dead: It accepts or responds to no events. This is certainly a bad user experience but may also lead to inexplicable bugs.

Callbacks

Now let's explore a technique in JavaScript called callbacks. If you remember what you first did, this line stands out:

```
randomNum = waitForMilliSeconds(5000);
```

This means that the returnvalue of waitForMilliSeconds is what gets populated in randomNum.

We've learned from other languages that you could pass in a function pointer to waitForMilliSeconds. Wouldn't it be nice if waitForMilliSeconds could call that function pointer when it's done with its five seconds of blocking work?

To facilitate that, modify your waitForMilliSeconds function, as shown below in the next snippet. The login has been trimmed for brevity and the only change is that instead of sending back a return value, you're now accepting a parameter called callbackFunc. When you're done with your work, you simply call this callback function and pass in the result.

```
function waitForMilliSeconds(
   milliSeconds, callbackFunc) {
    var date = new Date();
    ..
    random = Math.floor(Math.random() * 100);
    callbackFunc(random);
}
```

Accordingly, how you call this method also changes. This can be seen below.

```
waitForMilliSeconds(5000, (random) => {
    $("#output").text(random);
});
```

As you can see, you're now calling waitForMilliSeconds with two input parameters. The first parameter instructs the function to wait for five seconds and the second is an anonymous function parameter. This function gets called once waitForMilliSeconds is done and it calls the callbackFunc variable function.

Before you run this, what do you expect the behavior will be? Will it block the UI or not? Let's find out. Go ahead and run this. You'll notice that although the code seems to have a different structure, the callback seemed to have no effect on the single-threaded nature of the code. The UI still blocks.

Bummer!

Listing 1: The index.js server side file

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.static('public'));
app.get("/random", (request, response) => {
    waitForMilliSeconds(5000);
    const random = {
        "random": Math.floor(Math.random() * 100)
    };
    response.send(random);
});

app.listen(PORT, () =>
  console.log(`Server listening on port: ${PORT}`));

function waitForMilliSeconds(milliSeconds) {
    var date = new Date();
    var curDate = null;
    do { curDate = new Date(); }
    while (curDate - date < milliSeconds);
}
```

Listing 2: index.html

```
<html>

<head>
  <script
    src="https://code.jquery.com/jquery-3.7.1.min.js"
    integrity=".."
    crossorigin="anonymous"></script>
</head>

<body>
  Press button to make async call:
  <button type="button" id="btnRandom">Click</button>
  <br />
  <div id="output"></div>
  <script src="scripts/1.sync.js"></script>

  <br/>
  <textarea rows="5" cols="40">Try typing here
   while the long running call is running ..</textarea>
</body>

</html>
```

Listing 3: 1.sync.js client side synchronous JS

```
$("#btnRandom").on("click", function () {
    $("#output").text("Start");
    randomNum = waitForMilliSeconds(5000);
    $("#output").text(randomNum);
});

function waitForMilliSeconds(milliSeconds) {
    var date = new Date();
    var curDate = null;
    do { curDate = new Date(); }
    while (curDate - date < milliSeconds);
    return Math.floor(Math.random() * 100);
}
```

Listing 4: Using Promises

```
function waitForMilliSeconds(milliSeconds) {
    const myPromise = new Promise((resolve, reject) => {
        var date = new Date();
          ...
        random = Math.floor(Math.random() * 100);
        resolve(random);
      });
    return myPromise;
}
```

Listing 5: Simple XHR request

```
$("#btnRandom").on("click", function () {
    $("#output").text("Start");
    const xhr = new XMLHttpRequest();

    xhr.addEventListener("loadend", () => {
        $("#output").text(xhr.responseText);
    });

    xhr.open("GET", "/random");

    xhr.send();
    $("#output").text("Sent xhr request");
});
```

Listing 6: XHR and Promises

```
function makeXhrCall() {
    const myPromise = new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.addEventListener("loadend", () => {
            resolve(xhr.responseText);
        });
        xhr.open("GET", "/random");
        xhr.send();
      });
    return myPromise;
}
```

Well, at least you learned a new concept here, and that such callbacks have no effect on the single-threaded nature of execution.

Promise

JavaScript has yet another way of structuring your code, which is Promises. You may have seen them when writing AJAX code, where your code can make an HTTP call to the server without refreshing the whole page. This is how complex apps such as Google Maps were born. Before Google Maps, navigating a map required you to refresh the whole page. It was a horrible user experience, until someone showed us a better way. Technically speaking, Outlook for the web was leveraging this technique already, but hey, this isn't a race.

There's also pure client-side code. What if you were to use Promises instead of callbacks. Will the code not be single threaded then? Let's find out.

The idea behind a JavaScript Promise is that the function doesn't return a value, but instead returns a Promise. The Promise will either resolve (succeed) or reject (fail). When it resolves, it can send back a success output. If it fails, it can send back an error.

Your caller then uses standard paradigms around Promises to handle success with a Then method.

Let's modify the waitForMilliSeconds method to now return a Pomise and resolve it on success. You can see this method in **Listing 4**.

This allows you to write calling code:

```
waitForMilliSeconds(5000).then( (random) => {
    $("#output").text(random);
});
```

Let's run this code again and hit the click button. What do you see?

Ah! Yet again, although the code functionally is accurate, it still blocks the UI thread. The code is still single threaded. It responds to no input while the function is running and suddenly reacts to key strokes queued up in those five seconds.

Awful! Promises and lies. I guess that didn't solve the problem either.

XHR

At this point, you must be thinking that you've written so much AJAX code, and that code leveraged Promises, callbacks, and other paradigms. For sure you didn't see that blocking behavior there. What is so special about AJAX that it doesn't block?

There are many ways to write AJAX code. I've referenced jQuery and certainly jQuery has abstractions that help write AJAX code. The most basic way to call AJAX is by using XHR.

The way XHR works is that you instantiate a new instance of XMLHttpRequest. You subscribe to the **loadend** event and fire your HTTP request. Now whenever the call returns, the loadend event gets called and you get the results. You can process accordingly whether it's an error or success.

Let's start by instantiating the XMLHttpRequest.

```
const xhr = new XMLHttpRequest();
```

Before you send the request, let's subscribe to the loadend event. You're going to call an anonymous method when the event gets called, which shows whatever response the server sent.

```
xhr.addEventListener("loadend", () => {
    $("#output").text(xhr.responseText);
});
```

Next, let's send the request.

```
xhr.open("GET", "/random");
xhr.send();
```

You can see the final code that puts all this together in **Listing 5**.

Remember from **Listing 1**, the server-side code is basically the same code you've been using except now instead of running on the client, it's running on the server. It waits five seconds and sends back a random number.

Now go ahead and run this by referencing this script, pressing F5 to refresh the browser, and clicking the button.

Very interestingly, now the UI doesn't block. How odd is that?

Although this is great, wouldn't it be nice if complex client-side code could be afforded the luxury of being multi-threaded? This XHR-based code feels so complicated. My example was simple, but imagine how this could look with multiple dependencies, inputs dependent on other XHR calls succeeded, timing issues, etc. Ugh!

Promises and XHR

Let's start by cleaning this code up a bit. You've already seen Promises in action. Can you combine XHR and Promises together to help write code that's simpler? Sure! All you'd have to do is abstract out all this XHR code into its own method that returns a Promise. When you do an xhr.send(), just return the Promise. When XHR's loadend event is called, either resolve or reject the Promise as per the return results. This can be seen in **Listing 6**.

By doing so, the calling code becomes a lot simpler, as can be seen below.

```
makeXhrCall().then((output) => {
    $("#output").text(output);
});
```

Now go ahead and run this code. It runs just like before and it doesn't block the UI thread. Is this because you're using a Promise or that you're using an XHR? Well, you did use a Promise on a loop that was entirely on the client side and that did block the UI. This non-UI blocking magic is built into XHR.

Async Await

Recently, JavaScript introduced support for async await keywords. The Promise code looks cleaner than pure XHR code, but it still feels a bit inside out. Imagine if you had three Promises you needed to wait for and those inputs go into two more Promises, which finally go into another AJAX call? Luckily, Promises do have concepts such as resolveAll etc., which do help. They're an improvement over pure XHR code. However, the code becomes severely indented and you're stuck in a hell hole of brace matching and keeping your code under 80 characters width.

Async await helps you tackle that problem. Look at the sync code example from **Listing 3**. To convert that from sync to async, all you have to do is add the **async** keyword in front of the function. In other words, change this line of code:

```
function waitForMilliSeconds(milliSeconds) {
```

into this line of code:

```
async function waitForMilliSeconds(milliSeconds) {
```

That's it! Okay, I lied. Your calling pattern changes slightly too, but for the better. Your calling code changes like this:

dtSearch®

Instantly Search Terabytes

dtSearch's **document filters** support:
- popular file types
- emails with multilevel attachments
- a wide variety of databases
- web data

Over 25 search options including:
- efficient multithreaded search
- easy multicolor hit-highlighting
- forensics options like credit card search

Developers:
- SDKs for Windows, Linux, macOS
- Cross-platform APIs cover C++, Java and current .NET
- FAQs on faceted search, granular data classification, Azure, AWS and more

Visit dtSearch.com for
- hundreds of reviews and case studies
- fully-functional enterprise and developer evaluations

The Smart Choice for Text Retrieval® since 1991

dtSearch.com 1-800-IT-FINDS

Listing 7: Async XHR calls

```
let makeXhrCall = async () => {
    const myPromise = new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.addEventListener("loadend", () => {
            resolve(xhr.responseText);
        });
        xhr.open("GET", "/random");
        xhr.send();
      });
    return myPromise;
}
```

```
$("#btnRandom").on("click", async () => {
    $("#output").text("Start");
    random =
      await waitForMilliSeconds(5000);
    $("#output").text(random);
});
```

The changes aren't severe at all. All you had to do was follow two rules:

- Put "await" in front of a method call that is intended to be async.
- You cannot use await in any method that isn't async itself.

Now you can finally start writing async code that doesn't look like a severely indented case of brace-matching disease. You can extrapolate this to an XHR example, as can be seen in **Listing 7**.

Now when you run this code, although it's quite simplified, unless it's an XHR call, it still seems to block the UI.

Async Await with Workers

What's so peculiar about XHR that it doesn't block the UI thread? It seems to work on an Eventing model. Luckily for you, you can leverage that same capability using workers in JavaScript. Workers in JavaScript is a topic in itself, but for my purposes here, you need to separate out the blocking client-side code in its own worker, which, in this case, means its own JavaScript file.

This worker will now listen for, and respond to, messages after the work is done. So go ahead and create a new file under the Scripts folder called **hardwork.js**. Here's where you intend to run your hard-working loop that takes five seconds to complete.

Inside this hardwork.js, you first need to add an event listener for "message". This looks like:

Listing 8: Async Await with Workers

```
const worker = new Worker('./scripts/hardwork.js');

$("#btnRandom").on("click", async () => {
    $("#output").text("Start");
    random = await doHardWork();
    $("#output").text(random);
});

async function doHardWork() {
    return new Promise((resolve) => {
        worker.postMessage({
        command: "Start", "milliSeconds": 5000 });
        worker.addEventListener("message", (output) => {
            resolve(output.data);
        });
    });
}
```

```
addEventListener("message", (message) => {
    if (message.data.command === "Start") {
        waitForMilliSeconds(
         message.data.milliSeconds)
    }
});
```

As you can see, you're adding an event listener for "message". Whenever this is called, you can assume that some input parameters are sent to you, in this case, via "message.data.command", which tells you what action to take. The code is pretty simple, so you just look for "Start". Additionally, the data object has another property called milliSeconds that tells you how long to wait before returning a random number.

As you will see shortly, you have full control on the "data" property and you can define your own structure. Let's get to that in a moment. First let's focus on what "waitForMilliSeconds" looks like in this worker world.

Below is an abbreviated version of waitForMilliSeconds that communicates back to the caller via the postMessage method.

```
function waitForMilliSeconds(milliSeconds){
    ...
    random = Math.floor(Math.random() * 100);
    postMessage(random);
}
```

I removed the actual logic for brevity, but it's the same waitForMilliSeconds function I've used numerous times in this article already. That's it. This is what the worker looks like.

Now let's see how calling this worker looks.

As can be seen in **Listing 8**, you first start by creating a worker object using the "hardwork.js" file. Then, in the doHardWork method, post a message with a custom object structure, which is made available as the message. data property to the worker. I think "add a listener" for "message" and whenever a message is available, which is after the five second loop, I grab the random number output and resolve it. I think to use the async await pattern to set the output div's value.

Wait a minute. This looks quite similar to the XHR object, doesn't it? Over there also, you had an addEventListener, only the actual event was different.

Go ahead and run this code. Remember that this is a client-side loop. Go ahead and click the button. The code behaves as before, but now the UI thread no longer locks. The text area remains responsive while the loop is running without XHR.

Great. You've finally achieved the panacea of clean code, that doesn't block the UI thread.

Summary

As computers become increasingly powerful and complex, it's reasonable to assume that we're going to have to write increasingly complicated code. To write that complicated code, we're going to have to learn new patterns, such as the asynchronous patterns available in JavaScript.

In this article, I discussed many such patterns, and I built a story bit by bit to show you how you can use the various paradigms in JavaScript to create non-blocking code that's easy to maintain.

There's a lot more to learn, of course, and I'm sure, as time moves forward, greater demands and better patterns will emerge.

This is the beauty of our industry. Never a boring day.

Until next time, happy coding

Sahil Malik
CODE

Adding Copilots to Your Apps

The future is here now and you don't want to get left behind. Unlock the **true potential** of your software applications by adding Copilots.

CODE Consulting can assess your applications and provide you with a roadmap for adding Copilot features and optionally assist you in adding them to your applications.

Reach out to us today to get your application assessment scheduled. www.codemag.com/ai

Manipulating JSON Documents in .NET 8

JavaScript Object Notation (JSON) is a great way of storing configuration settings for a .NET application. JSON is also an efficient method to transfer data from one machine to another. JSON is easy to create, is human readable, and is easy for programming languages to parse and generate. This text-based format for representing data is language agnostic and thus easy to use in C#, JavaScript, Python, and almost any programming language existing today. In this article, you're going to learn multiple methods of creating and manipulating JSON documents in .NET 8. In addition, you'll learn how to serialize and deserialize C# objects to and from JSON.

Paul D. Sheriff
http://www.pdsa.com

Paul has been working in the IT industry since 1985. In that time, he has successfully assisted hundreds of companies' architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has multiple courses in the www.pluralsight.com library (https://bit.ly/3gvXgvj) and on Udemy.com (https://bit.ly/3WOK8kX) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

JSON Structure

As shown in **Figure 1**, a JSON object is made up of a collection of name/value pairs. You may hear these also expressed as key/value pairs, or, in C# terms, this is a property and a value. In C#, a JSON object is the equivalent of an object, record, or a struct with property names, and the values you assign to those properties. JSON can also be a collection of one or more objects (**Figure 2** and **Figure 3**). In C#, this would be the equivalent of a dictionary, hash table, keyed list, or associative array. Although I'm going to use C# in this article, the constructs mentioned are universal across all modern programming languages. Because of this, any language can manipulate these JSON objects easily.

A JSON object begins with a left brace and ends with a right brace (see **Figure 1**). Each name is followed by a colon and the name/value pairs are separated by a comma. Each name must be wrapped within double quotes. All string values must be wrapped within double quotes.

A JSON array is an ordered collection of values that begins with a left bracket and ends with a right bracket. Each value within the array is separated by a comma. The values within the array may be a single item, such as a string or a number (**Figure 2**), or each value within the array may be a JSON object (**Figure 3**).

Nested Objects

Each value after the name can be one of the various data types shown in **Table 1**. Although this isn't a large list of data types, all data can be expressed with just this set of types.

Look at **Figure 4** to see an example of a JSON object that has a value of each of the data types for each name. The "name" property has a string value "John Smith" enclosed within double quotes. The "age" property is a numeric with a value of 31. The "ssn" property is empty as represented by the keyword null. The "address" property is another JSON object and is thus enclosed by curly braces. The "phoneNumbers" property is a JSON array, with each value of the array another JSON object. As you see, JSON is very flexible and can represent almost any type of structure you can possibly need in your programming tasks.

JSON Manipulation Classes

There are several classes within a couple of namespaces in .NET 8 that you use to work with JSON in your C# applications. The first namespace is **System.Text.Json** and the second namespace is **System.Text.Json.Nodes**. You have most likely already used the **JsonSerializer** class to serialize C# objects to JSON, or to deserialize a JSON string into a C# object. However, there are other classes you can use to add nodes to a JSON object, set and retrieve values, and create new JSON objects.

The System.Text.Json Namespace

Within this namespace are classes and structures to help you manipulate JSON documents including the JsonSerializer class. **Table 2** provides a description of each of the classes within this namespace. Each of these classes is illustrated throughout this article. All the classes and structures within this namespace are immutable. This

Figure 1: JSON objects are made up of name/value pairs.

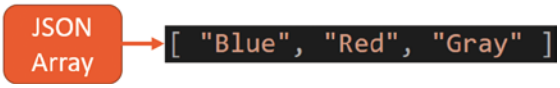

Figure 2: JSON arrays can have just simple values as their elements.

Figure 3: Each element in a JSON array can be a JSON object.

Data Type	Description
Number	A signed number value. The number may be a whole number (integer) or a decimal with a fractional part.
String	A set of zero or more Unicode characters enclosed within double quotes. Strings support a backslash escaping syntax just as you find in C#.
Boolean	Either a true or a false value.
Object	A JSON object using the JSON object syntax previously explained.
Array	A set of zero or more values using the JSON array syntax previously explained. Each element within the array may be of any of the types shown in this table.
Null	Use the keyword null to signify an empty value for this name.

Table 1: JSON has a limited set of data types available for a value.

Class / Structure	Description
JsonDocument	A class that represents an immutable (read-only) document object model (DOM) of a JSON object. Use this class when you don't have a C# class to deserialize the JSON into and you need to access the name/value pairs programmatically.
JsonProperty	This structure represents a single JSON property within a JSON object. For example, **"colorId": 1** is an example of a JsonProperty.
JsonElement	This structure represents a single value within a JSON property. For example, the number one (1) within the property **"colorId": 1** is the JsonElement.
JsonSerializer	A class used to serialize a JSON string into a C# object, or to deserialize a C# object into a JSON string.
Utf8JsonWriter	A class that can be used to emit a JSON document one property at a time. This class is a high-performance, forward-only, non-cached method of writing UTF-8 encoded JSON text.

Table 2: The System.Text.Json namespace contains classes and structures for manipulating and serializing JSON objects

Class	Description
JsonObject	This class represents a mutable (read/write) JSON document. This class is like the JsonDocument class from the System.Text.Json namespace.
JsonNode	This class represents a mutable (read/write) node within a JSON document. This class is like the JsonProperty class from the System.Text.Json namespace.
JsonValue	This class represents a mutable (read/write) JSON value. This class is like the JsonElement class from the System.Text.Json namespace.
JsonArray	This class represents a mutable (read/write) JSON array.

Table 3: The System.Text.Json.Nodes namespace contains classes for manipulating in-memory JSON objects as a document object model

means that once they've been instantiated with data, they cannot be modified in any way.

The System.Text.Json.Nodes Namespace

The classes in this namespace (**Table 3**) are for creating and manipulating in-memory JSON documents using a DOM. These classes provide random access to JSON elements, allow adding, editing, and deleting elements, and can convert dictionary and key value pairs into JSON documents.

Build In-Memory JSON Documents

I recommend following along with this article to ensure that you have a solid foundation for manipulating JSON documents. The tools needed to follow the step-by-step instructions in this article are .NET 6, 7, or 8, and either Visual Studio 2022 or VS Code. Create a console application named **JsonSamples**. The goal of this first example is to create the following JSON object.

```
{
  "name": "John Smith",
  "age": 31
}
```

Of course, there are many ways to create this JSON object using C# and the JSON classes. For this first example, open the **Program.cs** file, delete any lines of code in this file, and add the following Using statement as the first line.

```
using System.Text.Json.Nodes;
```

Below the Using statement, create a variable named **jo** that is an instance of a JsonObject. The JsonObject provides the ability to add, edit, and delete nodes within the JSON document.

```
JsonObject jo = new();
```

Figure 4: Each element in a JSON object can be another object, or an array.

Add two name/value pairs to the JsonObject using the C# **KeyValuePair** class, as shown in the following code.

```
jo.Add(new KeyValuePair<string,
  JsonNode?>("name", "John Smith"));

jo.Add(new KeyValuePair<string,
  JsonNode?>("age", 31));
```

Write out the JSON document using the ToString() method on the JsonObject.

```
Console.WriteLine(jo.ToString());
```

The output from this statement is the JSON object shown earlier. Notice how the JSON is nicely formatted. If you wish to remove all of the carriage returns, line feeds, and whitespace between all the characters, change the call from the ToString() method to the ToJsonString() method instead. You should then see the following JSON appear in the console window.

```
{"name":"John Smith","age":31}
```

Use a New C# 12 Feature

In C# 12 (.NET 8) you can create the JsonObject using the following syntax. Note that the square brackets used in the code are a new C# 12 feature that allows you to initialize the new JsonObject object without using the new keyword.

```
JsonObject jo =
[
  new KeyValuePair<string,
    JsonNode?>("name", "John Smith"),
  new KeyValuePair<string,
    JsonNode?>("age", 31)
];
```

Using a Dictionary Class

You can pass an instance of a **Dictionary<string, JsonNode?>** to the constructor of the JsonObject to create your JSON document. Replace the code in the **Program.cs** file with the following:

```
using System.Text.Json.Nodes;

Dictionary<string, JsonNode?> dict = new() {
  ["name"] = "John Smith",
  ["age"] = 31
};

JsonObject jo = new(dict);

Console.WriteLine(jo.ToString());
```

Using a JsonValue Object

The Add() method on the JsonObject class also allows you to pass in the name and a JsonValue object. Pass in the value to static method Create() on the JsonValue class to create a new JsonValue object.

```
using System.Text.Json.Nodes;

JsonObject jo = new() {
  { "name", JsonValue.Create("John Smith") },
  { "age", JsonValue.Create(31) }
};

Console.WriteLine(jo.ToString());
```

Because the JsonObject can be initialized with a Dictionary object, you may use the same syntax you used to create a Dictionary object in the constructor of the JsonObject, as shown in the following code:

```
using System.Text.Json.Nodes;

JsonObject jo = new() {
  ["name"] = "John Smith",
  ["age"] = 1
};

Console.WriteLine(jo.ToString());
```

Create Nested JSON Objects

Not all JSON objects are simple name/value pairs. Sometimes you need one of the properties to be another JSON object. The "address" property is another JSON object that has its own set of name/value pairs, as shown in the following snippet:

```
{
  "name": "John Smith",
  "age": "31",
  "ssn": null,
  "isActive": true,
  "address": {
    "street": "1 Main Street",
    "city": "Nashville",
    "stateProvince": "TN",
    "postalCode": "37011"
  }
}
```

To create the above JSON object, create a new instance of a JsonObject and, using a Dictionary object, build the structure you need, as shown in the following code snippet:

```
using System.Text.Json.Nodes;

JsonObject jo = new() {
  ["customer"] = "Acme",
  ["IsActive"] = true,
  ["address"] = new JsonObject() {
    ["street"] = "123 Main Street",
    ["city"] = "Walla Walla",
    ["stateProvince"] = "WA",
    ["postalCode"] = "99362",
    ["country"] = "USA"
  }
};

Console.WriteLine(jo.ToString());
```

Parse JSON Strings into Objects

JSON documents are commonly stored as strings in a file or in memory. Instead of attempting to read specific values in the JSON using File IO or string parsing, you can parse the string into a JsonNode object. Once in this object, it's very easy to retrieve single values, or entire nodes.

Listing 1: To save typing, I have created a class with some JSON documents

```
namespace JsonSamples;

public class JsonStrings
{
  public const string PERSON =
    @"{
      ""name"": ""John Smith"",
      ""age"": 31,
      ""ssn"": null,
      ""isActive"": true
    }";

  public const string PHONE_NUMBERS =
    @"[
        {
          ""type"": ""Home"",
          ""number"": ""615.123.4567""
        },
        {
          ""type"": ""Mobile"",
          ""number"": ""615.345.6789""
        },
        {
          ""type"": ""Work"",
          ""number"": ""615.987.6543""
        }
    ]";

  public const string PERSON_ADDRESS =
    @"{
      ""name"": ""John Smith"",
      ""age"": 31,
      ""ssn"": null,
      ""isActive"": true,
      ""address"": {
        ""street"": ""1 Main Street"",
        ""city"": ""Nashville"",
        ""state"": ""TN"",
        ""postalCode"": ""37011""
      }
    }";


  public const string PERSON_ADDRESS_PHONES =
    @"{
      ""name"": ""John Smith"",
      ""age"": 31,
      ""ssn"": null,
      ""isActive"": true,
      ""address"": {
        ""street"": ""1 Main Street"",
        ""city"": ""Nashville"",
        ""state"": ""TN"",
        ""postalCode"": ""37011""
      },
      ""phoneNumbers"": [
        {
          ""type"": ""Home"",
          ""number"": ""615.123.4567""
        },
        {
          ""type"": ""Mobile"",
          ""number"": ""615.345.6789""
        },
        {
          ""type"": ""Work"",
          ""number"": ""615.987.6543""
        }
      ]
    }";
}
```

Instead of repeating the same set of JSON strings throughout this article, I'm creating a single class (**Listing 1**) with some string constants to represent different JSON documents. The PERSON constant is a JSON document that contains four properties to represent a single person. The PHONE_NUMBERS constant is a JSON array of a few phone numbers where each number has a **type** property with a value such as Home, Mobile, or Work. The PERSON_ADDRESS constant contains a nested **address** property that is another object with a **street**, **city**, **state**, and **postalCode** properties. The PERSON_ADDRESS_PHONES constant contains person information, address information, and an array of phone numbers all in one JSON object.

Create a JsonNode Object

The JsonNode class is probably the class you will use most often. It's very flexible and contains most of the functionality you need when manipulating JSON documents. For example, to parse the PERSON constant into a JsonNode object, place the following code into the **Program.cs** file:

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(JsonStrings.PERSON);

Console.WriteLine(jn!.ToString());
Console.WriteLine(jn!.GetValueKind());
```

Run the console application and you should see the following displayed in the console window:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "isActive": true
}
Object
```

The first Console.WriteLine() statement emits the JSON object, and the second Console.WriteLine() statement reports the kind of object contained in the JsonNode object that's the type of **Object**. Of course, you may pass to the Parse() method any of the other constant strings. Write the following code in the **Program.cs** file to parse the PHONE_NUMBERS constant into a JsonNode object:

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(
  JsonStrings.PHONE_NUMBERS);

Console.WriteLine(jn?.ToString());
Console.WriteLine(jn!.GetValueKind());
```

Run the console application and you should see the following displayed in the console window:

```
[
  {
    "type": "Home",
```

```
    "number": "615.123.4567"
  },
  {
    "type": "Mobile",
    "number": "615.345.6789"
  },
  {
    "type": "Work",
    "number": "615.987.6543"
  }
]
Array
```

Notice that in this case, the GetValueKind() method reports this as an **Array** instead of an **Object**. When the JSON string that is read in starts with a square bracket, it's a JSON array instead of a JSON object.

Create a JsonDocument Object

Earlier in this article, you learned that you could add items to a JsonObject in its constructor. There's no constructor for the JsonDocument object, so you must use the Parse() method to get valid JSON data into this object. The JsonDocument object is a very efficient object to use when all you need to do is to read data from a JSON document. Once the data is parsed into the JsonDocument, access the **RootElement** property to retrieve the JSON. Write the following code into the **Program.cs** file:

```
using JsonSamples;
using System.Text.Json;

// Parse string into a JsonDocument object
using JsonDocument jd = JsonDocument.Parse(JsonStrings.PERSON);

// Get Root JsonElement structure
JsonElement je = jd.RootElement;

Console.WriteLine(je.ToString());
Console.WriteLine(je.ValueKind);
```

In the code above, you retrieve the **RootElement** property and place it into a new variable of the type JsonElement. It's this class that you're going to use to read the data from JSON document, as you shall soon learn. Run the application and you should see the following displayed in the console window:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "isActive": true
}
Object
```

Read Data from JSON Documents

There are a few different ways you can read individual name/value pairs from a JSON document. Both the JsonElement and the JsonNode classes allow you to get at the data within the JSON.

Once you've parsed some JSON into a JsonDocument object, you must always use the **RootElement** property to retrieve specific values within the JSON document. You can either place the RootElement property into a JsonElement object, or you can use the full path of the **JsonDocument.RootElement** property. Write the code shown in **Listing 2** into the **Program.cs** file.

In **Listing 2**, you parse the PERSON string into a JsonDocument. You then get the property called "name" from the JSON document and place this into a JsonElement object named **je**. Because the "name" property is a string value, call the GetString() method on the **je** variable to extract the value from the "name" property. If you don't wish to use a separate variable, you may access the jd.RootElement property directly by calling the GetProperty("age") to get the "age" element. Call the GetInt32() method on this element to extract the "age" value and display it on the console. Run the application and you should see the following output from **Listing 2** displayed in the console window:

```
Name=John Smith
Age=31
```

Retrieve Data in a Nested Object

Look back at **Listing 1** and notice that the PERSON_ADDRESS constant is the one with the nested "address" object. To access the "city" value within the "address", replace the code in the **Program.cs** file with the following:

```
using JsonSamples;
using System.Text.Json;

// Parse string into a JsonDocument object
using JsonDocument jd =
  JsonDocument.Parse(JsonStrings.PERSON_ADDRESS);

// Get a specific property from JSON
JsonElement je = jd.RootElement
  .GetProperty("address").GetProperty("city");

// Get the string value from the JsonElement
Console.WriteLine($"City={je!.GetString()}");
```

After parsing the string into the JsonDocument object, access the RootElement property and call the GetProperty("address") to get to the "address" property, and then call GetProperty("city") to get to the "city" property. Once you have this element in a JsonElement object, call the GetString() method to retrieve the value for the "city" property.

Listing 2: Retrieve values from the JSON using the RootElement property

```
using JsonSamples;
using System.Text.Json;

// Parse string into a JsonDocument object
using JsonDocument jd =
  JsonDocument.Parse(JsonStrings.PERSON);

// Get a specific property from JSON
JsonElement je =
  jd.RootElement!.GetProperty("name");

// Get the numeric value
// from the JsonElement
Console.WriteLine(
  $"Name={je!.GetString()}");
Console.WriteLine(
  $"Age={jd.RootElement!
    .GetProperty("age")!.GetInt32()}");
```

Run the application and you should see the following displayed in the console window:

```
City=Nashville
```

Reading Data Using JsonNode

Unlike the JsonDocument object, the JsonNode object has an indexer that allows you to specify the name in double brackets to retrieve that specific node, as shown in the following code:

```
// Parse string into a JsonNode object
JsonNode? jn =
  JsonNode.Parse(JsonStrings.PERSON);

// Get the age node
JsonNode? node = jn!["age"];
```

With this new JsonNode object, **node**, retrieve the value as a JsonValue using the AsValue() method. With the JsonValue object, you can report the path of where this value came from, the type (string, number, Boolean, etc.), and get the value itself as shown in the following code:

```
// Get the value as a JsonValue
JsonValue value = node!.AsValue();
Console.WriteLine($"Path={value.GetPath()}");
Console.WriteLine($"Type={value.GetValueKind()}");
Console.WriteLine($"Age={value}");
```

Another option is to retrieve the value using the GetValue<T>() method, as shown in the following code:

```
// Get the value as an integer
int age = node!.GetValue<int>();
Console.WriteLine($"Age={age}");
```

If you type in the above code snippets into the **Program.cs** file and run the application, the following should be displayed in the console window:

```
Path=$.age
Type=Number
Age=31
Age=31
```

Retrieve Data in a Nested Object

Look back at **Listing 1** and look at the PERSON_ADDRESS constant. This JSON string is the one with the nested "address" object. To access the "city" value within the "address", you replace the code in the **Program.cs** file with the following:

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(
  JsonStrings.PERSON_ADDRESS);

// Get the address.city node
JsonNode? node =
  jn!["address"]!["city"];

// Display string value from the JsonNode
Console.WriteLine(
  $"City={node!.AsValue()}");
```

The above code parses the PERSON_ADDRESS string into a JsonNode object. It then uses an indexer on the **jn** variable to drill down to the **address.city** node. This node is placed into a new JsonNode object named **node**. The value of the "city" property is retrieved using the AsValue() method and displayed on the console window when you run this application.

Add, Edit, and Delete Nodes

To add a new name/value pair to a JSON document, create a JsonObject object out of the PERSON JSON string constant and convert it to a JsonObject using the AsObject() method. Once you have a JsonObject, use the Add() method to create a new name/value pair, in this case "hairColor": "Brown".

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonObject
JsonObject? jo = JsonNode.Parse(
  JsonStrings.PERSON)?.AsObject();

jo?.Add("hairColor", JsonValue.Create("Brown"));

Console.WriteLine(jo?.ToString());
```

Replace the code in the **Program.cs** file with the code listed above and run the application to see the following displayed in the console window:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "isActive": true,
  "hairColor": "Brown"
}
```

Updating a Node

Use the JsonNode object to update a value in a name/value pair. Parse the JSON into a JsonNode object, then access the name using an indexer. Set the value using the equal sign just as you would any normal assignment in .NET.

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jo = JsonNode.Parse(
  JsonStrings.PERSON);

jo!["age"] = 42;

Console.WriteLine(jo?.ToString());
```

Replace the code in the **Program.cs** file with the code listed above and run the application to see the following code displayed in the console window. Notice that the age value has changed from 31 to 42.

```
{
  "name": "John Smith",
```

```
  "age": 42,
  "ssn": null,
  "isActive": true
}
```

Deleting a Node

The Remove() method on a JsonObject is used to delete a name/value pair from a JSON document. Create a JsonObject object out of the JSON PERSON string constant. Once you have a JsonObject, use the Remove() method, passing in the name you wish to remove from the JSON. In the code below, you remove the "age" name/value pair:

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonObject
JsonObject? jo = JsonNode.Parse(
  JsonStrings.PERSON)?.AsObject();

jo?.Remove("age");

Console.WriteLine(jo?.ToString());
```

Replace the code in the **Program.cs** file with the code listed above and run the application to see the following displayed in the console window. The **"age": 31** name/value pair has been removed from the JSON document.

```
{
  "name": "John Smith",
  "ssn": null,
  "isActive": true
}
```

Working with Arrays

In addition to a simple object, JSON can contain arrays of strings, numbers, Booleans, and JSON objects. Instead of using the JsonObject to represent a JSON document, use the **JsonArray** class to represent a list of items. For example, to create an array of string values, replace the code in the **Program.cs** file with the following:

```
using System.Text.Json.Nodes;

JsonArray ja = [ "John", "Sally", "Charlie"];

Console.WriteLine(ja.ToString());
Console.WriteLine(ja.GetValueKind());
```

Run the application and you should see the following code displayed in the console window. Notice that after the JSON array is displayed, the type reported from the call to the GetValueKind() method is "Array".

```
[
  "John",
  "Sally",
  "Charlie"
]
Array
```

To create an array of JSON objects, use the same syntax with the square brackets, but create a new JsonObject object separated by commas for each element you wish to create in the array. Write the following code into the **Program.cs** file:

```
using System.Text.Json.Nodes;

JsonArray ja = [
  new JsonObject() {
    ["name"] = "John Smith",
    ["age"] = 31
  },
  new JsonObject() {
    ["name"] = "Sally Jones",
    ["age"] = 33
  }
];

Console.WriteLine(ja.ToString());
Console.WriteLine(ja.GetValueKind());
```

Run the application and you should see the following displayed in the console window:

```
[
  {
    "name": "John Smith",
    "age": 31
  },
  {
    "name": "Sally Jones",
    "age": 33
  }
]
Array
```

Manipulate an Array

Like most arrays in .NET, you can easily add and remove elements within the array. Given the previous JsonArray object declaration, you can insert a new entry into the array by adding the following code after the declaration. The Insert() method lets you specify where in the array you wish to add the new object. In this case, you are adding a new element into the first position of the array.

```
ja.Insert(0, new JsonObject() {
  ["name"] = "Charlie Chaplin",
  ["age"] = "50"
});
```

You can always create a new JsonObject first, initialize it with some data, then add that new JsonObject to the JsonArray using the Add() method. The Add() method adds the JsonObject to the end of the array.

```
JsonObject jo = new() {
  ["name"] = "Buster Keaton",
  ["age"] = 55
};
ja.Add(jo);
```

Array elements may be removed by either a reference to the actual object, or by using an index number, as shown in the following two lines of code:

```
ja.Remove(jo);
ja.RemoveAt(2);
```

Extract a JSON Array from a JSON Node

Look back at **Listing 1** to view the PERSON_ADDRESS_PHONES constant string. Within this JSON object, there's an object named "phoneNumbers" that contains an array of phone number objects. To extract the phone number objects from this string, you first need to parse the string into a JsonNode object. You then retrieve the value from "phoneNumbers" object and convert it into a JsonArray object using the AsArray() method, as shown in the following code:

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(
  JsonStrings.PERSON_ADDRESS_PHONES);

// Get the Phone Numbers Array
JsonArray? ja = jn!["phoneNumbers"]!.AsArray();

Console.WriteLine(ja.ToString());
Console.WriteLine(ja.GetValueKind());
```

Place this code into the **Program.cs** file and run the application to display the following in the console window:

```
[
  {
    "type": "Home",
    "number": "615.123.4567"
  },
  {
    "type": "Mobile",
    "number": "615.345.6789"
  },
  {
    "type": "Work",
    "number": "615.987.6543"
  }
]
Array
```

Iterate Over Array Values Using JsonNode

Once you have a JsonArray object, you may iterate over each value in the array to extract the different property values. In the code shown below, you parse the PHONE_NUMBERS string constant into a JsonNode object. Next, convert this JsonNode object into a JsonArray using the AsArray() method. Use a foreach loop to iterate over each element in the array and emit the "type" and "number" properties onto the console window.

```
using JsonSamples;
using System.Text.Json.Nodes;

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(
  JsonStrings.PHONE_NUMBERS);

JsonArray? nodes = jn!.AsArray();
foreach (JsonNode? node in nodes) {
  Console.WriteLine($"Type={node!["type"]},
    Phone Number={node!["number"]}");
}
```

Type the above code into the **Program.cs** file and run the application to see the following values displayed in the console window:

```
Type=Home, Phone Number=615.123.4567
Type=Mobile, Phone Number=615.345.6789
Type=Work, Phone Number=615.987.6543
```

Iterate Over Array Values Using JsonDocument

If you wish to use the JsonDocument class instead of a JsonNode class, the following code illustrates the differences between the two classes. After parsing the PHONE_NUMBERS string constant into a JsonDocument, convert the **RootElement** property, which is an array, into an ArrayEnumerator using the EnumerateArray() method. You may now iterate over the array of JsonElement objects and display the phone number type and the phone number itself.

```
using JsonSamples;
using System.Text.Json;

// Parse string into a JsonDocument object
using JsonDocument jd = JsonDocument.Parse(
  JsonStrings.PHONE_NUMBERS);

JsonElement.ArrayEnumerator elements =
  jd.RootElement.EnumerateArray();
foreach (JsonElement elem in elements) {
  Console.WriteLine(
    $"Type={elem.GetProperty("type")},
    Phone Number={elem.GetProperty("number")}");
}
```

Listing 3: Retrieve a single item from an array

```
using JsonSamples;
using System.Text.Json.Nodes;

string? value = string.Empty;

// Create JsonNode object from Phone Numbers
JsonNode? jn = JsonNode.Parse(
  JsonStrings.PHONE_NUMBERS);

// Cast phone numbers as an array
JsonArray ja = jn!.AsArray();

// Search for Home number
JsonNode? tmp =
  ja.FirstOrDefault(row =>
    (string)(row!["type"]!
      .GetValue<string>()) == "Home");

// Extract the home number value
value = tmp![["number"]!.GetValue<string>();

Console.WriteLine($"Home Number={value}");
```

Listing 4: A sample runtime configuration file

```
{
  "runtimeOptions": {
    "tfm": "net8.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "8.0.0"
    },
    "configProperties": {
      "System.Runtime...": false
    }
  }
}
```

The output is the same as the output shown above when using the JsonNode object to iterate over the array values.

Get a Single Phone Number from Array

Looking back at **Listing 1,** you see the PHONE_NUMBERS string constant, which is a JSON array. After you parse this data into a JsonNode, you might wish to retrieve just the home phone number from this array. After converting the phone numbers to a JsonArray object, use the FirstOrDefault() method to search where the "type" value is equal to "Home". If this node is found, extract the number value to display on the console window, as shown in **Listing 3**. If you type the code shown in **Listing 3** into the **Program.cs** file and run the application, the code displays "Home Number=615.123.4567" in the console window.

Listing 5: Use File I/O to read a value from the runtime configuration file

```
using System.Text.Json.Nodes;

string? value = string.Empty;
string fileName =
  $"{AppDomain.CurrentDomain.FriendlyName}
    .runtimeconfig.json";

if (File.Exists(fileName)) {
  JsonNode? jn = JsonNode.Parse(
    File.ReadAllText(fileName));
  value = jn!["runtimeOptions"]
    ![“framework”]
    ![“version”]?.GetValue<string>();
}

Console.WriteLine(json);
```

Listing 6: Create an application settings file in the console application

```
{
  "ConnectionStrings": {
    "DefaultConnection":
      "Server=Localhost;
       Database=AdventureWorks;
       Trusted_Connection=Yes;
       MultipleActiveResultSets=true;
       TrustServerCertificate=True;"
  },
  "AdvWorksAppSettings": {
    "ApplicationName": "Adventure Works",
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

Listing 7: Read the appsettings.json file using .NET File I/O classes

```
using System.Text.Json.Nodes;

string connectString = string.Empty;
string fileName = "appsettings.json";

if (File.Exists(fileName)) {
  // Read settings from file
  JsonNode? jd = JsonNode.Parse(
    File.ReadAllText(fileName));

  // Extract the default connection string
  connectString = jd![“ConnectionStrings”]
    ![“DefaultConnection”]?
    .GetValue<string>() ?? string.Empty;
}

Console.WriteLine(connectString);
```

Parsing JSON from a File

There are a couple of different methods you may use to extract JSON from a file. You can use .NET File I/O classes, or you can use the IConfiguration interface. Let's start by looking at how you can read JSON values using the .NET File I/O classes.

Read Runtime Configuration Settings

When you run a .NET application, there's a **runtimeconfig.json** file (**Listing 4**) created with information about the application. The JSON object shown below is an example of what's generated from a console application. When you run an ASP.NET web application, there will be additional information in this file.

If you wish to read the .NET Framework version from this file, you need to first open the file and parse the text into a JsonNode or JsonDocument object, as shown in **Listing 5**. You then access the **runtimeOptions.framework.version** property to retrieve the value "8.0.0". Replace all the code in the **Program.cs** file with the code shown in **Listing 5** and run the application to display the runtime version.

Create an appsettings.json File

In most .NET applications you write, you most likely will need a file to store global settings such as a connection string, logging levels, and other application-specific settings. This information is generally stored in a file named **appsettings.json**. Add this file to the console application and put the settings shown in **Listing 6** into this file. Once the file is created, click on the file and bring up the **Properties** window. Set the **Copy to Output Directory** property to **Copy always**. You should put the connection string all on one line. I had to break it across several lines due to the formatting constraints of the print magazine.

Read appsettings.json File Using File I/O

Because the appsettings.json file only contains text, you can read in the JSON contained in this file using the .NET File I/O classes, as shown in **Listing 7**. In this code, you set the **fileName** variable to point to the location of the appsettings.json file. Because the appsettings.json file is in the same folder as the executable that's running, you don't need to specify a path to the file. If the file exists, read all of the text using the ReadAllText() method of the File class and pass that text to the Parse() method of the JsonNode class. Once you have the JSON in the JsonNode object, you can read the value from the **ConnectionStrings.DefaultConnection** property. Type the code shown in **Listing 7** into the **Program.cs** file, run the application, and you should see the connection string in the appsettings.json file displayed in the console window.

Writing to a File

If you're running a WPF application, or a console application, it's perfectly acceptable to write data back to the appsettings.json file. Of course, you wouldn't want to do this when running an ASP.NET web application. The code shown in **Listing 8** reads in the appsettings.json file, adds a new name/value pair, then writes the new JSON back to the appsettings.json file. Type the code in **Listing 8** into the **Program.cs** file and run the application to produce the following results:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=...;"
```

```
  },
  "AdvWorksAppSettings": {
    "ApplicationName": "Adventure Works",
    "LastDateUsed": "12/27/2023"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

Using the IConfiguration Interface

Instead of using the .NET File I/O classes, you can take advantage of the IConfiguration interface and the ConfigurationBuilder class to read in a JSON file. To use the ConfigurationBuilder class, you must add two packages to your project.

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Json

After adding these two packages to your project, you can write the code shown in **Listing 9**. In this code, you pass in the runtime configuration file name (see **Listing 4**) to the AddJsonFile() method on the ConfigurationBuilder. The Build() method is called to create the configuration builder object, which reads the JSON file into memory and converts the text into a JSON document. Use the GetSection() method to retrieve a specific section within the JSON file. In this case, you're asking for the **runtimeOptions** section. From the **section** variable, you can now retrieve the framework version number. Type in the code in **Listing 9** into the **Program.cs** file, run the application, and you should see the version number appear in the console window.

Read the appsettings.json File

You previously read the appsettings.json file using the .NET File I/O classes. In **Listing 10** you're now going to use the ConfigurationBuilder to read the same file. Because the appsettings.json file is in the same folder as the executable that's running, you don't need to specify a path to the file. Type the code in **Listing 10** into the **Program.cs** file, run the application and you should see the connection string appear in the console window.

Bind Settings to a Class

Instead of reading values one at a time from a configuration file, you can bind a section within a configuration file to a class with just one line of code. Create a class named AppSettings and add a property that maps to each name in the configuration file. In the following code, there's a sole property named **ApplicationName** that maps to the "ApplicationName" property in the appsettings.json file shown in **Listing 6**.

```
namespace JsonSamples;

public class AppSettings
{
  public string ApplicationName { get; set; }
    = string.Empty;
}
```

Listing 8: Write a new value to the appsettings.json file

```
string fileName = "appsettings.json";
JsonObject? jo = null;

if (File.Exists(fileName)) {
  // Read settings from file
  jo = JsonNode.Parse(
    File.ReadAllText(fileName))?.AsObject();

  if (jo != null) {
    // Locate node to add to
    JsonObject? node =
      jo!["AdvWorksAppSettings"]?.AsObject();

    // Add new node
    node?.Add("LastDateUsed",
      JsonValue.Create(
        DateTime.Now.ToShortDateString()));

    // Write back to file
    File.WriteAllText(fileName, jo?.ToString());
  }
}

Console.WriteLine(jo?.ToString());
```

Listing 9: Use the ConfigurationBuilder class to read in a JSON file

```
using Microsoft.Extensions.Configuration;

string? value = string.Empty;
string fileName =
  $"{AppDomain.CurrentDomain.FriendlyName}
     .runtimeconfig.json";

IConfiguration config =
  new ConfigurationBuilder()
  .AddJsonFile(fileName)
  .Build();

IConfigurationSection section =
  config.GetSection("runtimeOptions");
value = section["framework:version"]
  ?? string.Empty;

Console.WriteLine(value);
```

Listing 10: Read the appsettings.json file using the ConfigurationBuilder class

```
using Microsoft.Extensions.Configuration;

string connectString;

IConfiguration config =
  new ConfigurationBuilder()
    .AddJsonFile("appsettings.json")
    .Build();

IConfigurationSection section =
  config.GetSection("ConnectionStrings");
connectString =
  section["DefaultConnection"]
    ?? string.Empty;

Console.WriteLine(connectString);
```

To perform the binding operation, add the package **Microsoft.Extensions.Configuration.Binder** to your project using the NuGet Package Manager. Add the following code to the **Program.cs** file and run the application to see the application name displayed in the console window:

```
using JsonSamples;
using Microsoft.Extensions.Configuration;

AppSettings entity = new();
```

```
IConfiguration config =
  new ConfigurationBuilder()
  .AddJsonFile("appsettings.json")
  .Build();

config.Bind("AdvWorksAppSettings", entity);

Console.WriteLine(entity.ApplicationName);
```

Serialize C# Object to JSON

So far, everything you've done manipulates JSON using C# code. Another excellent feature of .NET is that you may serialize your C# objects into JSON using just a few lines of code. This is very handy for sending C# objects over the internet via Web API calls. In fact, the code you're going to learn about now is exactly how ASP.NET sends your data across the internet when writing Web API calls. To illustrate this concept, right mouse-click on the project and add a new class named **Person**, as shown in the following code snippet:

```
namespace JsonSamples;
```

Listing 11: Add a JsonSerializerOptions object to control how the JSON is formatted

```
using JsonSamples;
using System.Text.Json;

Person entity = new() {
  Name = "John Smith",
  Age = 31,
  SSN = null,
  IsActive = true
};

JsonSerializerOptions options = new() {
  PropertyNamingPolicy =
    JsonNamingPolicy.CamelCase,
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(entity, options));
```

Listing 12: Add JSON attributes to your C# class properties to control serialization

```
using System.Text.Json.Serialization;

namespace JsonSamples;

public class PersonWithAttributes
{
  [JsonPropertyName("personName")]
  [JsonPropertyOrder(1)]
  public string? Name { get; set; }
  [JsonPropertyName("personAge")]
  [JsonPropertyOrder(2)]
  public int Age { get; set; }
  public string? SSN { get; set; }
  public bool IsActive { get; set; }

  [JsonIgnore]
  public DateTime? CreateDate { get; set; }
  [JsonIgnore(Condition =
    JsonIgnoreCondition.WhenWritingNull)]
  public DateTime? ModifiedDate { get; set; }

  public override string ToString()
  {
    return $"{Name}, Age={Age},
      SSN={SSN}, IsActive={IsActive}";
  }
}
```

```
public class Person
{
  public string? Name { get; set; }
  public int Age { get; set; }
  public string? SSN { get; set; }
  public bool IsActive { get; set; }

  public override string ToString()
  {
    return $"{Name}, Age={Age},
      SSN={SSN}, IsActive={IsActive}";
  }
}
```

After creating the Person class, replace all the code in the **Program.cs** file with the following:

```
using JsonSamples;
using System.Text.Json;

Person entity = new() {
  Name = "John Smith",
  Age = 31,
  SSN = null,
  IsActive = true
};

Console.WriteLine(
  JsonSerializer.Serialize(entity));
```

This code uses the Serialize() method of the JsonSerializer class from the System.Text.Json namespace. Pass in the instance of your C# object to the Serialize() method and a string of JSON is returned. Run the application and you should see the following string of JSON appear in your console window:

```
{"Name":"John Smith","Age":31,
 "SSN":null,"IsActive":true}
```

Notice that there's no indentation or spaces between the values. Also notice that the names are the exact same case as your C# class property names. JSON usually uses camel case for names (first letter is lower-case), whereas C# uses Pascal case (first letter is upper-case).

Change Casing of Property Names

If you wish to change the casing of the property names, create an instance of a JsonSerializerOptions object and set the **PropertyNamingPolicy** to the enumeration value of **CamelCase** (as seen in **Listing 11**). Change the formatting of the JSON to become indented by setting the **WriteIndented** property to **true**.

Type the code in **Listing 11** into the **Program.cs** file and run the application to display the following JSON in the console window:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "isActive": true
}
```

Go back to the **Program.cs** file and change the **PropertyNamingPolicy** property to **JsonNamingPolicy.Snake-**

CaseUpper to produce each property as upper-case with each word in the property name separated by an underscore, as shown in the following output:

```
{
  "NAME": "John Smith",
  "AGE": 31,
  "SSN": null,
  "IS_ACTIVE": true
}
```

Go back to the **Program.cs** file and change the **PropertyNamingPolicy** property to **JsonNamingPolicy.SnakeCaseLower** to produce each property as lower-case with each word in the property name separated by an underscore, as shown in the following output:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "is_active": true
}
```

The other enumeration values you may set the PropertyNamingPolicy to are **JsonNamingPolicy.KebabCaseUpper** or **JsonNamingPolicy.KebabCaseLower**. This policy separates each word in the property name with a dash instead of an underscore, as shown below:

```
{
  "name": "John Smith",
  "age": 31,
  "ssn": null,
  "is-active": true
}
```

ontrol Serialization Using JSON Attributes

In the System.Text.Json.Serialization namespace are some attributes you may use to decorate C# class properties to help you control how each property is serialized. There are several attributes you may use, but the most used are **JsonPropertyName**, **JsonIgnore**, and **JsonPropertyOrder**. **Listing 12** shows a **PersonWithAttributes** class with these attributes applied to different properties.

When you decorate a C# property with the **JsonPropertyName** attribute, you pass the JSON name to use to the attribute when this property is serialized. When a C# class is serialized, the order of the properties is in the order they appear in the class. To change the order in which the properties are serialized, add a **JsonPropertyOrder** attribute to each property and set the order in which you want them to appear. If the **JsonPropertyOrder** attribute is not applied to a property, the default number is zero (0). To never have a property serialized into JSON, apply the **JsonIgnore** attribute with no parameters. You may also set the **Condition** property of the **JsonIgnore** attribute to not serialize the data when the value is null.

After creating the **PersonWithAttributes** class, write the code shown in **Listing 13** in the **Program.cs** file. In this listing, notice that the **CreateDate** and **ModifiedDate** properties are both set to the current date and time. If you run the code shown in **Listing 13**, the following output is displayed in the console window:

Listing 13: Serialize the C# object with the JSON attributes applied

```
using JsonSamples;
using System.Text.Json;

PersonWithAttributes entity = new() {
  Name = "John Smith",
  Age = 31,
  SSN = null,
  IsActive = true,
  // This property is never serialized
  CreateDate = DateTime.Now,
  // Comment this property to
  // remove from serialization
  ModifiedDate = DateTime.Now
};

JsonSerializerOptions options = new() {
  PropertyNamingPolicy =
    JsonNamingPolicy.CamelCase,
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(entity, options));
```

```
{
  "ssn": null,
  "isActive": true,
  "modifiedDate": "2024-01-28T12:48:53",
  "personName": "John Smith",
  "personAge": 31
}
```

There are a few things to notice about this output. The modifiedDate appears in the output because the value is not null. The **Name** and **Age** C# properties are emitted as **personName** and **personAge** in JSON. These two properties also appear at the end of the object because their order was set to one (1) and two (2) respectively. If you go back to the code in the **Program.cs** file, comment the **ModifiedDate** property, and rerun the application, the **ModifiedDate** value will not display in the output.

Serializing Objects with Enumerations

Another common scenario is that you have a class with an enumeration as one of the property types. When you serialize that object, you can either emit the numeric value of the enumeration or use the string representation of the enumeration itself. Create a new enumeration named **PersonTypeEnum**.

```
namespace JsonSamples;

public enum PersonTypeEnum
{
  Employee = 1,
  Customer = 2,
  Supervisor = 3
}
```

Create a class named **PersonWithEnum** that has a **PersonType** property that is of the type PersonTypeEnum.

```
namespace JsonSamples;

public class PersonWithEnum
{
  public string? Name { get; set; }
  public PersonTypeEnum PersonType { get; set; }
```

Getting the Sample Code

You can download the sample code for this article by visiting **www.CODEMag.com** under the issue and article, or by visiting **www.pdsa.com/downloads**. Select "Articles" from the Category drop-down. Then select "XML Serialization and Validation in .NET 6/7" from the Item drop-down.

Listing 14: Create a JwtSettings class to be nested within an AppSettings class

```
namespace JsonSamples;

public class JwtSettings
{
  public string Key { get; set; }
    = string.Empty;
  public string Issuer { get; set; }
    = string.Empty;
  public string Audience { get; set; }
    = string.Empty;
  public int MinutesToExpiration
    { get; set; }
  public string[] AllowedIPAddresses
    { get; set; } = [];

  #region ToString Override
  public override string ToString()
  {
    return $"{Key} - {Issuer} -
      {Audience} - {MinutesToExpiration}";
  }
  #endregion
}
```

Listing 15: Write code to serialize a nested object to view the JSON output

```
using JsonSamples;
using System.Text.Json;

AppSettingsNested entity = new() {
  ApplicationName = "JSON Samples",
  JWTSettings = new() {
    Key = "ALongKeyForASymmetricAlgorithm",
    Issuer = "JsonSamplesAPI",
    Audience = "PDSCJsonSamples",
    MinutesToExpiration = 60
  }
};

JsonSerializerOptions options = new() {
  PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(entity, options));
```

```
  public override string ToString()
  {
    return $"{Name}, Type={PersonType}";
  }
}
```

Open the **Program.cs** file and write the code shown in the code snippet below:

```
using JsonSamples;
using System.Text.Json;

PersonWithEnum entity = new() {
  Name = "John Smith",
  PersonType = PersonTypeEnum.Supervisor
};

JsonSerializerOptions options = new() {
  PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(entity, options));
```

When you run the application, the following output is displayed in the console window.

```
{
  "name": "John Smith",
  "personType": 3
}
```

Notice that the personType property has a value of 3, which equates to the **Supervisor** enumeration value. Go back to the **Program.cs** file and add the following using statement at the top of the file:

```
using System.Text.Json.Serialization;
```

Set the **Converters** property in the JsonSerializerOptions object to use an instance of the **JsonStringEnumConverter** class. This class works with the serializer and instead of emitting the numeric value of enumeration properties, it emits the string representation of the enumeration.

```
JsonSerializerOptions options = new() {
  PropertyNamingPolicy =
    JsonNamingPolicy.CamelCase,
  WriteIndented = true,
  Converters =
  {
    new JsonStringEnumConverter()
  }
};
```

After adding the JsonStringEnumConverter object, run the application and the following should now display in the console window:

```
{
  "name": "John Smith",
  "personType": "Supervisor"
}
```

Serialize a Nested Object

Often you have a property in a class that is itself another class. Don't worry, the JSON serialization process handles this situation just fine. To illustrate, create a class called **JwtSettings,** as shown in **Listing 14**. Next, create a class named **AppSettingsNested** that has two properties: **ApplicationName** and **JWTSettings**. The data type for the **JWTSettings** property is the JwtSettings class you just created.

```
namespace JsonSamples;

public class AppSettingsNested
{
  public string ApplicationName
    { get; set; } = string.Empty;

  public JwtSettings JWTSettings
    { get; set; } = new();
}
```

Now that you have a nested class, write the code shown in **Listing 15** into the **Program.cs** file. In this code, you fill in the **ApplicationName** property, create a new instance of a JwtSettings class for the **JWTSettings** property, and then fill in each property in that class too. Run the application and you should see the following displayed in your console window:

```
{
  "applicationName": "JSON Samples",
  "jwtSettings": {
    "key": "ALongKeyForASymmetricAlgorithm",
    "issuer": "JsonSamplesAPI",
    "audience": "PDSCJsonSamples",
    "minutesToExpiration": 60,
    "allowedIPAddresses": []
  }
}
```

Serialize a Dictionary

If you have a generic Dictionary<TKey, TValue> object, or a KeyValuePair object filled with data, the JSON serializer emits those as a single object. Open the **Program.cs** file and add the code shown below.

```
using System.Text.Json;

Dictionary<string, object?> dict = new()
  {
    {"name", "John Smith"},
    {"age", 31},
    {"isActive", true}
  };

JsonSerializerOptions options = new() {
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(dict, options));
```

When you run the application, the console window displays the following JSON object:

```
{
  "name": "John Smith",
  "age": 31,
  "isActive": true
}
```

Serialize a List

To write a JSON array, you can use any of the IEnumerable objects in .NET such as an array or a List<T>. To illustrate, use the PersonWithEnum class and create a generic list of two PersonWithEnum objects, as shown in **Listing 16**. Type the code shown in **Listing 16** into the **Program.cs** file and run the application to display the following output in the console window.

```
[
  {
    "name": "John Smith",
    "personType": 3
  },
  {
    "name": "Sally Jones",
    "personType": 1
  }
]
```

Listing 16: Create a generic List<T> and serialize it to create a JSON array

```
using JsonSamples;
using System.Text.Json;

List<PersonWithEnum> list = new()
{
  new PersonWithEnum {
    Name = "John Smith",
    PersonType = PersonTypeEnum.Supervisor
  },
  new PersonWithEnum {
    Name = "Sally Jones",
    PersonType = PersonTypeEnum.Employee
  }
};

JsonSerializerOptions options = new() {
  PropertyNamingPolicy =
    JsonNamingPolicy.CamelCase,
  WriteIndented = true
};

Console.WriteLine(
  JsonSerializer.Serialize(list, options));
```

Listing 17: Use the Deserialize() method to convert a JSON string into a C# object

```
using JsonSamples;
using System.Text.Json;

// JSON names must match
// C# properties by default
string json = @"{
  ""Name"": ""John Smith"",
  ""Age"": 31,
  ""SSN"": null,
  ""IsActive"": true
}";

// Deserialize JSON string into Person
Person? entity = JsonSerializer
  .Deserialize<Person>(json);

Console.WriteLine(entity);
```

Listing 18: Pass in options to control the deserialization process

```
using JsonSamples;
using System.Text.Json;

string json = @"{
  ""name"": ""John Smith"",
  ""age"": 31,
  ""ssn"": null,
  ""isActive"": true
}";

// Override case matching
JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true
};

// Deserialize JSON string into Person
Person? entity =
  JsonSerializer.Deserialize<Person>(json, options);

Console.WriteLine(entity);
```

Deserialize JSON into a C# Object

Now that you've seen how to serialize a C# object into JSON, let's look at reversing the process. To illustrate, create a JSON string with the JSON property name that exactly matches the C# property name in the class you wish to deserialize this JSON into. You use the JsonSerializer class (**List-**

Listing 19: Use the JsonNode object to deserialize a JSON string into a C# object

```
using JsonSamples;
using System.Text.Json.Nodes;
using System.Text.Json;

string json = @"{
  ""name"": ""John Smith"",
  ""age"": 31,
  ""ssn"": null,
  ""isActive"": true
}";

JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true,
};

// Parse string into a JsonNode object
JsonNode? jn = JsonNode.Parse(json);

// Deserialize JSON string into Person
// using the JsonNode object
Person? entity =
  jn.Deserialize<Person>(options);

Console.WriteLine(entity);
```

ing 17) like you did for serializing, but call the Deserialize() method passing in the data type you wish to deserialize the JSON string into and the string itself. Type the code in **Listing 17** into the Program.cs file and run the application to see the following results displayed in the console window:

```
John Smith, Age=31, SSN=, IsActive=True
```

Use Serialization Options

Just like you did when serializing, if the case of the JSON property names doesn't match the C# property names, you may set the **PropertyNameCaseInsensitive** property to true in the options and pass those options to the Deserialize() method, as shown in **Listing 18**. Notice that in this listing the JSON property names start with lowercase. If you forget to use the options and the property names have different casing, then no data is mapped from the JSON to the C# object, so an empty object is returned from the Deserialize() method.

Deserialize Using the JsonNode Object

Another option for deserializing JSON into a C# object is to use either the JsonNode or the JsonDocument classes. The code shown in **Listing 19** uses the JsonNode object to illustrate. The JsonDocument class looks very similar to that of the JsonNode. Type this code into the **Program.cs** file and run the application to get the same output as you saw in the last example.

Deserializing Enumeration Values

If you know that the JSON object is going to have the string representation of a C# enumeration, set the **Converters** property to a new instance of the JsonStringEnumConverter class in the JsonSerializerOptions object, as shown in **Listing 20**. If you forget to include the Converters property on the JsonSerializerOptions, a JsonException is thrown. Type the code shown in **Listing 20** into the **Program.cs** file and run the application to see the following output appear in the console window:

```
John Smith, Type=Supervisor
```

Convert JSON File to a Person Object

Right mouse-click on the JsonSamples console application and create a new folder named **JsonSampleFiles**. Right mouse-click on the JsonSampleFiles folder and add a new file named **person.json**. Place the following JSON object into this file:

```
{
  "name": "Sally Jones",
  "age": 39,
  "ssn": "555-55-5555",
  "isActive": true
}
```

Open the **Program.cs** file and replace the contents of the file with the code shown in **Listing 21**. In this sample, you're using a FileStream object to open the file and stream it to the Deserialize() method. Thus, you should use the Using statement in front of the FileStream declaration so it will be disposed of properly. Run the application and you should see the following output displayed in the console window:

```
Sally Jones, Age=39,
  SSN=555-55-5555, IsActive=True
```

ADVERTISERS INDEX

Advertising Sales:
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

Advertisers Index

1&1 Internet, Inc.	www.1and1.com	7
CODE Consulting	www.codemag.com/techhelp	2, 57
CODE Divisions	www.codemag.com	75
CODE Framework	www.codemag.com/framework	49
CODE Magazine	www.codemag.com/magazine	37, 65
CODE Staffing	www.codemag.com/staffing	25
dtSearch	www.dtSearch.com	69
LEAD Technologies	www.leadtools.com	5
SPTechCon	www.sptechcon.com	45

This listing is provided as a courtesy to our readers and advertisers. The publisher assumes no responsibility for errors or omissions.

Listing 20: If you use enumerations, be sure to include the JsonStringEnumConverter in the serialization options

```
using JsonSamples;
using System.Text.Json.Serialization;
using System.Text.Json;

string json = @"{
  ""name"": ""John Smith"",
  ""personType"": ""Supervisor""
}";

JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true,
  // Avoid an exception being
  // thrown on Deserialize()
  Converters =
  {
    new JsonStringEnumConverter()
  }
};

// Deserialize JSON string
PersonWithEnum? entity = JsonSerializer
  .Deserialize<PersonWithEnum>(json, options);

Console.WriteLine(entity);
```

Convert a JSON Array in a File to a List of Person Objects

Right mouse-click on the JsonSampleFiles folder and add a new file named **persons.json**. Place the following JSON array into this file:

```
[
  {
    "name": "John Smith",
    "age": 31,
    "ssn": null,
    "isActive": true
  },
  {
    "name": "Sally Jones",
    "age": 39,
    "ssn": "555-55-5555",
    "isActive": true
  }
]
```

Open the **Program.cs file** and type in the code shown in **Listing 22**. This code is almost the same as the code you wrote to deserialize a single person object; the only difference is that you pass the data type List<Person> to the Deserialize() method. Once you have the collection of Person objects, iterate over the collection and display each person on the console window. Run the application and you should see the following output displayed in the console window:

```
John Smith, Age=31,
  SSN=, IsActive=True
Sally Jones, Age=39,
  SSN=555-55-5555, IsActive=True
```

Get Maximum Age from List of Person Objects

After reading in a list of objects, you may now use LINQ operations or any Enumerable methods such as Min, Max, Sum, and Average on that list. In the code shown in **Listing 23**, the Max() method is applied to the list and the maximum value found in the Age property is displayed on the console window. When you run this application, the value reported back should be thirty-nine (39).

Using the Utf8JsonWriter Class

The Utf8JsonWriter class is a high-performance, forward-only, non-cached method of writing JSON documents. Just like with serialization, you can control the output of the JSON to include white space, and indentation. **Listing 24** shows how to write a single JSON object into a MemoryStream object. Note that both the MemoryStream and the Utf8JsonWriter objects implement the IDisposable interface, so you need to prefix them with the Using statement. You must start each JSON document by calling the WriteStartObject() or the WriteStartArray() method. You then call the appropriate method to write a string, a number, a Boolean, a null, or a comment. Finally, call the WriteEndObject() or the WriteEndArray() method to close the JSON document. Type the code shown in **Listing 24** into the Program.cs file and run the application to display the output shown below in the console window:

Listing 21: Read a JSON file and convert the JSON object in the file into a C# object

```
using JsonSamples;
using System.Text.Json;

string fileName =
  $"{AppDomain.CurrentDomain.BaseDirectory}
  JsonSampleFiles\\person.json";

using FileStream stream = File.OpenRead(fileName);

JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true,
};

// Deserialize JSON string into Person
Person? entity = JsonSerializer
  .Deserialize<Person>(stream, options);

Console.WriteLine(entity);
```

Listing 22: Read an array of JSON objects from a file and convert to a list of person objects

```
using JsonSamples;
using System.Text.Json;

string fileName =
  $"{AppDomain.CurrentDomain.BaseDirectory}
     JsonSampleFiles\\persons.json";

using FileStream stream = File.OpenRead(fileName);

JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true,
};

// Deserialize JSON string into List<Person>
List<Person>? list = JsonSerializer
  .Deserialize<List<Person>>(stream, options);

if (list != null) {
  foreach (var item in list) {
    Console.WriteLine(item);
  }
}
```

```
{
  "name": "John Smith",
```

Listing 23: After deserializing a list of person objects, apply the Max() method to calculate the largest numeric value

```
using JsonSamples;
using System.Text.Json;

string fileName =
  $"{AppDomain.CurrentDomain.BaseDirectory}
     JsonSampleFiles\\persons.json";

using FileStream stream = File.OpenRead(fileName);

JsonSerializerOptions options = new() {
  PropertyNameCaseInsensitive = true,
};

// Deserialize JSON string
List<Person>? list = JsonSerializer
  .Deserialize<List<Person>>(stream, options);

// Calculate maximum age
int maxAge = list?.Max(row => row.Age) ?? 0;

Console.WriteLine(maxAge);
```

Listing 24: The Utf8JsonWriter object is a forward-only cursor for emitting JSON quickly

```
using System.Text.Json;
using System.Text;

JsonWriterOptions options = new() {
  Indented = true
};

using MemoryStream ms = new();
using Utf8JsonWriter writer =
  new(ms, options);

writer.WriteStartObject();
writer.WriteString("name", "John Smith");
writer.WriteNumber("age", 31);
writer.WriteBoolean("isActive", true);
writer.WriteEndObject();
writer.Flush();

string json = Encoding.UTF8
  .GetString(ms.ToArray());

Console.WriteLine(json);
```

Listing 25: The Utf8JsonWriter object can write arrays as well as single objects

```
using System.Text.Json;
using System.Text;

JsonWriterOptions options = new() {
  Indented = true
};

using MemoryStream ms = new();
using Utf8JsonWriter writer =
  new(ms, options);

writer.WriteStartArray();
writer.WriteStartObject();
writer.WriteString("name", "John Smith");
writer.WriteNumber("age", 31);
writer.WriteBoolean("isActive", true);
writer.WriteEndObject();
writer.WriteStartObject();
writer.WriteString("name", "Sally Jones");
writer.WriteNumber("age", 39);
writer.WriteBoolean("isActive", true);
writer.WriteEndObject();
writer.WriteEndArray();
writer.Flush();

string json = Encoding.UTF8
  .GetString(ms.ToArray());

Console.WriteLine(json);
```


CODE Is Hiring!

CODE Staffing is accepting resumes for various open positions ranging from junior to senior roles.

We have **multiple openings** and will consider candidates who seek full-time employment or contracting opportunities. **For more information:** www.codestaffing.com.


```
  "age": 31,
  "isActive": true
}
```

Write a JSON Array

As just mentioned, you may also write a JSON array of data using the Utf8JsonWriter class. The only difference is that you start writing using the WriteStartArray() method, and then call the WriteStartObject() method to create your first JSON object in the array. You then continue this process until all your array elements are written. Finish the JSON array document by calling the WriteEndArray() method, as shown in **Listing 25**. Type the code shown in **Listing 25** into the Program.cs file and run the application to display the output shown below in the console window:

```
[
  {
    "name": "John Smith",
    "age": 31,
    "isActive": true
  },
  {
    "name": "Sally Jones",
    "age": 39,
    "isActive": true
  }
]
```

Summary

In this article, you were introduced to the many different classes in .NET that are used to manipulate JSON documents. If you need fast, read-only access to JSON documents, the JsonDocument class is what you should use. If you need the ability to modify JSON in your application, use the JsonNode class. Serialization is accomplished using the JsonSerialization, JsonDocument, or the JsonNode classes. When you're dealing with configuration files such as the appsettings.json file in your application, take advantage of the IConfiguration interface and the ConfigurationBuilder class. Finally, to write JSON documents one name/value pair at a time, the Utf8JsonWriter class gives you the most control over how your document is formatted.

Paul D. Sheriff

Value Object's New Mapping: EF Core 8 ComplexProperty

EF Core 8 was released late in 2023 and, if you haven't kept up, there are some important and interesting things to be aware of. There were over 100 tweaks and additions and another 125 bug fixes. I'll be highlighting those that are most important and a few that piqued my interest or just my curiosity. If you want to explore all of the enhancements and new features on GitHub, here's a link for those issues: https://bit.ly/EFCore8Features. The list of issues for the fixes can be perused at https://bit.ly/EFCore8Fixes.

If you've followed my tech wanderings over the years, it may be no surprise that my A#1 favorite new feature is the ComplexProperty mapping, an alternative to using Owned Entities to map complex types and value objects. The new ComplexProperty mapping provides a far superior way to map complex types (and therefore, value objects) than the Owned Entity mapping we've been using since the beginning of EF Core. To be clear, there are still some scenarios that are not yet supported, so you may end up using a mix of the two mappings until ComplexProperty is complete. It's the team's intention for this to eventually replace Owned Entities in their entirety. ComplexProperty is a big deal and it was a big deal for the team to execute. It's at the top of their "what's new" lists as well.

Although the OwnsOne and OwnsMany mappings have fulfilled the basic need to map classes that are used as properties of entities, the work that they were doing under the covers was complicated and led to numerous side effects. The team had tweaked the inner logic a number of times across versions, creating breaking changes along the way, but never really solved the problem properly. They have been contemplating a replacement for some time and have finally pulled it off.

There are some caveats, however, which are a few capabilities that didn't make it into EF Core 8 but will be ready for EF Core 9. In those cases, we just continue using the owned entity mappings. I'll explain the caveats after I allow you to feast your eyes on the new ComplexProperty mapping.

TLDR Complex Types and Value Objects

Let's be sure we're all on the same page. A complex type is a class that doesn't have any identity and is used as a property of another class. An easy example is if you have a first name property and a last name property in a Customer class.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateOnly FirstPurchase { get; set; }
}
```

Instead of using two string types for every class that needs a person's name, you can create a new class that only has those two strings.

```
public class Customer
{
    public int CustomerId { get; set; }
    public PersonName Name { get; set; }
    public DateOnly FirstPurchase { get; set; }
}
public class PersonName
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

PersonName is a complex type and can be used as a property of any other class, such as this ShipLabel class, which is obviously missing an address but that's only to keep this explanation simple.

```
public class ShipLabel
{
    public int Id { get; set; }
    pubic DateOnly Printed {get; set;}
    public PersonName Name { get; set; }
}
```

The most important attribute of PersonName is that it has no identity.

Julie Lerman
@julielerman
thedatafarm.com/contact

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 blogs at thedatafarm.com/blog, is the author of the highly acclaimed "Programming Entity Framework" books, and many popular videos on Pluralsight.com.

Listing 1: PersonName class implemented as a value object

```
public class PersonName
{
    public PersonName(string firstName,
                      string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public string FirstName { get; init; }
    public string LastName { get; init; }

    public override bool Equals(object? obj)
    {
        return obj is PersonName name &&
               FirstName == name.FirstName &&
               LastName == name.LastName;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(FirstName,
          LastName);
    }
}
```

A value object is a critical Domain-Driven Design construct that enhances a complex type by ensuring that it's immutable and that its equality is always based on the values of every property in the type by overriding the Equals and GetHashCode methods. **Listing 1** shows a version of PersonName that's defined as a value object.

Whether PersonName is a simple complex type or a value object, EF Core will see that type in its model when it's used as a property of another entity, such as Customer. However, it will balk because it can only assume that it's another entity but can't figure out what to do with it because it has no key property. This results in a runtime exception when EF Core is trying to work out the data model.

The Problem(s) with Owned Entities

Owned entities originally came into EF Core to enable this mapping. They were a new paradigm for handling complex types, different from EF6 and earlier. By mapping the Name property of Customer as an owned entity (using the OwnsOne mapping), EF Core knows that it's okay that there's no key. However, in order to track and persist it, EF Core, under the covers, treats that PersonName object as an entity in a relationship with Customer. It does so by using a shadow property to infer a key in memory but ensures that the values are stored as individual fields in the Customers table in the database. That is the default behavior. You can configure the mapping to store the values in a separate database table.

It was a very clever solution that leveraged the existing behavior of EF Core. However, because of the complexity of faking the key, there were problematic side effects. For example, EF Core couldn't comprehend if you left the property null or needed to edit it. Some of those problems were resolved but there are others still. For example, you can't copy an owned object from one entity instance to multiple other classes. **Listing 2** shows logic that attempts to create two separate shipping label objects for one person. It will fail.

Why? When this code assigns a person.Name to the second label, EF Core **moves** it from the label it was already assigned to. The first label will no longer have a Name—the property is now null. In the docs, the EF Core team explains other common use cases that will fail as well. Keep in mind that in unit tests that don't involve EF Core, the code poses no problem and is sensible. It's EF Core's tracking that fails. And as I said earlier, the team tried variations on how to handle these types of problems across versions of EF Core, all the while pondering how to implement a better mapping, rather than continuing to try to make owned entities work across the various needed scenarios.

Listing 2: Retrieving a Customer and Using its Name for a new Label

```
var storedCustomer = ctx.Customers.First();
var label = new ShipLabel
{
    Name = storedCustomer.Name,
};
var label2 = new ShipLabel
{
    Name = storedCustomer.Name,
};
ctx.AddRange(label, label2);
ctx.SaveChanges();
```

Hello, or Welcome Back, Complex Properties

Entity Framework, and I mean pre-EF Core, had a concept of complex property mappings that were more natural than owned entities. EF Core 8 harkens back to that concept, although with a different implementation. EF Core still won't make an assumption that a complex type is anything but a malformed entity that needs a key property. But we have a new way to map it with the ComplexProperty mapping.

Whereas previously you'd have used the OwnsOne method for the owned property, now you use the ComplexProperty method.

```
override protected void
  OnModelCreating(ModelBuilder modelBuilder)
{
    //modelBuilder.Entity<Customer>()
      .OwnsOne(c => c.Name);
    modelBuilder.Entity<Customer>()
      .ComplexProperty(c => c.Name);
}
```

Like OwnsOne, ComplexProperty has its own methods where you can further define the property, for example, tying it to a backing field or specifying that it's required.

But what's most important is that EF Core just treats this as a property and when storing it, explodes the FirstName and LastName properties out to fields in the Customer's table (by default). It doesn't set up a fake relationship or fake key and then have to tangle with those every time you track, save, or retrieve data. Because it's not being treated as a separate entity, you can also share it among instances as needed. The logic in **Listing 2** will succeed.

It's interesting to compare the visualizations of the model as well (using the wonderful EF Core Power Tool's diagram tool) shown in **Figure 1**. As an Owned Entity, the DbContext sees the PersonName as its own entity in a one-to-one

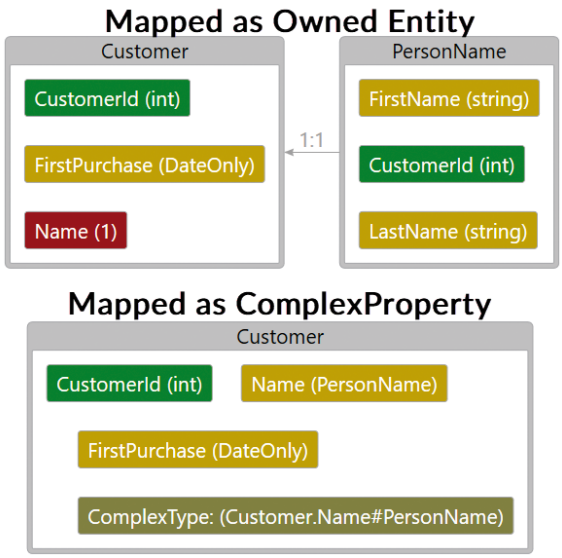

Figure 1: Data Model with Person mapped as an Owned Entity vs. a Complex Property

relationship with Customer. Notice that there's even a CustomerId shadow property. As a ComplexProperty, EF Core comprehends that it's just another property of Customer.

In addition to noting the differences between the model in **Figure 1**, it's also interesting to see the DebugView for the Customer and ShipLabel entities as they're seen by the change tracker prior to calling SaveChanges in **Listing 2**.

First is the view for the OwnsOne mapping, and there's so much going on here that I'm only showing the ShortView.

```
Customer {CustomerId: 1} Unchanged
Customer.Name#PersonName {CustomerId: 1}
  Unchanged FK {CustomerId: 1}
ShipLabel {Id: -2147482647} Added
ShipLabel {Id: -2147482646} Added
ShipLabel.Name#PersonName
  {ShipLabelId: -2147482646} Added FK
  {ShipLabelId: -2147482646}
```

With the owned entity mapping, the Customer's Name and the Name of only one of the ShipLabels (remember, EF Core moved it, not copied it), are tracked separately and it's a bit convoluted.

Listing 3 shows the DebugView when PersonName is mapped as a ComplexProperty: This time, I'm sharing the LongView with more details because it's so easy to read. The details look just as you would expect. And EF Core is doing a lot less work to manage the PersonName data.

It's much simpler and the side effects of the fake entities just disappear.

Classes, Records, and Value Objects, Oh My!

The PersonName value object shown in **Listing 1** includes a primary constructor to make it simpler to instantiate.

This leads us to the first caveat of ComplexProperty. In EF Core 8, it won't work with a class that has a primary constructor—emphasis on **class**. If you want to map this class with ComplexProperty, you need to remove that constructor and use an object initializer to instantiate a new PersonName like this:

```
new PersonName{FirstName ="John",LastName="Doe"}
```

Otherwise, you'll need to go back to mapping with OwnsOne.

What about records instead of a class? I recall first seeing the exploration that the C# team was doing on records at an MVP summit quite a few years ago. Because of how they simplified creating value objects, I was definitely eager to see them come into the language. Record types internalize equality comparison so you don't need to override the Equals or GetHasCode methods every single time.

However, records did not play very well with owned entities and again, there were side effects to worry about. Therefore, I never used records until EF Core 8 brought us the ComplexProperty mapping and I had a bit of catching up to do to learn about records because there are many ways to express them.

Listing 3: DebugView (LongView) with Name mapped as a ComplexProperty in Customer and ShipLabel

```
Customer {CustomerId: 1} Unchanged
    CustomerId: 1 PK
    FirstPurchase: '1/1/2021'
    Name (Complex: PersonName)
      FirstName: 'John'
      LastName: 'Doe'
ShipLabel {Id: -2147482647} Added
    Id: -2147482647 PK Temporary
    Name (Complex: PersonName)
      FirstName: 'John'
      LastName: 'Doe'
ShipLabel {Id: -2147482646} Added
    Id: -2147482646 PK Temporary
    Name (Complex: PersonName)
      FirstName: 'John'
      LastName: 'Doe'
```

A Quick Records Overview

Records have a number of formats. I spent a lot of time understanding the various ways to express a record to choose the correct flavor. The documentation was very helpful (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record), but it still took a few read throughs for me. I have encapsulated some of the important details in **Table 1** for a quick reference.

To begin with, a record, by default, is a reference type. But a record struct is a value type—the correct choice for a value object. There are more decisions to make.

You can define a record struct as a positional record, which has nothing more than a primary constructor and looks like this:

```
public record struct PersonName (
 string FirstName, string LastName);
```

That's the entire implementation! Internally, C# infers the FirstName and LastName string properties.

Currently my PersonName record has no logic and is a good candidate for a positional record. If you have no need for logic or further constraints in the object, the streamlined positional syntax is awesome.

But—and this is a big but—on its own, a record struct is not, I repeat, **not**, immutable**.** Therefore, it fails the requirement of a value object. Luckily, C#12 added the capability to make a record struct read only.

```
public readonly record struct PersonName (
 string FirstName, string LastName);
```

That's a very succinctly expressed and simple value object.

If you do need additional logic, you can express the record more like a class with properties and other logic explicitly defined, as I'm doing here, using init accessors to ensure that it's still immutable. This is an example of a read-only record struct without positional properties.

Declaration	Type	Mutability
class	Reference type	Mutable unless designed otherwise
record	Reference type	Immutable
record struct	Value type	Mutable unless designed otherwise
readonly record struct	Value type	Immutable

Table 1: How classes and records are interpreted

```
public readonly record struct PersonName
{
    public FirstName X { get; init; }
    public LastName Y { get; init; }

    public string FullName =>
      $"{FirstName} {LastName}";
}
```

There's an interesting capability of records that you should consider, which is that it's possible to create a new instance of a record with new values. This feature uses a **with** expression to replace property values.

For example, I might have instantiated a PersonName using:

```
var jazzgreat=new PersonName("Ella",
  "Fitzgeralde");
```

Then I discover the typo of the "e" at the end of her name. Of course, with only two properties, I could easily create a new instance from scratch. But if you have a lot of properties, you could use the **with** expression syntax:

```
jazzgreat=jazzgreat with
  {LastName = "Fitzgerald"};
```

There are a lot of other nuances of records that you can learn about in the docs at https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record.

Because I found it confusing to sort out all of the behaviors of the various flavors of record types, I've listed the critical aspects of each (as well as class for comparison) in **Table 1**.

Null Value Object Properties

The most important caveat about EF Core 8's implementation of ComplexProperty, which is a deal breaker for some, is that it doesn't currently support null objects. We saw this same problem with owned entities in an earlier rendition, but owned entities now support null properties.

As an example, let's say I made PersonName nullable in the customer type. Perhaps this isn't a logical change, but it serves my demonstration purpose.

```
public class Customer
{
    public int CustomerId { get; set; }
    public PersonName? Name { get; set; }
    public DateOnly FirstPurchase { get; set; }
}
```

Mapped as an OwnedEntity, EF Core can sort this out. The default mapping results in Name_FirstName and Name_LastName columns in the Customers table both being nullable.

The OwnedEntity mapping comprehends the null property when I create and store a customer without a name.

```
var customer=new Customer
{
   FirstPurchase=new DateOnly(2021, 1, 1)
};
ctx.Customers.Add(customer);
ctx.SaveChanges();
```

When Name is null, the Name_FirstName and Name_LastName database fields are both null as well. When I retrieve the customer, EF Core returns a Customer with a null Name property.

At some point, ComplexProperty will have the same behavior. But currently (in EF Core 8), specifying Name as a nullable type results in a runtime exception. The exception you get is dependent on how the value object is defined.

If the value object is a class, you get a message about the fact that it can't be optional when EF Core is attempting to build the data model based on the DbContext mappings.

```
System.InvalidOperationException: 'Configuring the complex
property 'Customer.Name' as optional is not supported,
call 'IsRequired()'. See https://github.com/dotnet/efcore/
issues/31376 for more information.'
```

Making it required just so EF Core is happy is not a pleasing solution. It should only be required if your domain invariants specify that Name should be required. If it's required, you can use ComplexProperty. If not, you're stuck with OwnsOne.

If PersonName is a record struct (with or without positional properties), you'll trigger a different exception. EF Core configures the database fields from the value object properties (Customer_FirstName, and Customer_LastName) as non-nullable fields. At runtime, the database will throw an exception saying that it can't insert a null value into a non-nullable column.

Note: If you look into the GitHub issue referenced in the exception message, please add your vote to make sure the EF Core team addresses this. I do expect it to be supported in EF Core 9 but every vote from the community helps them prioritize.

What Else Is and Isn't Supported?

Nullability of complex types, whether or not they are value objects, is an important topic, indeed. But there are a few other points to be aware of: some limitations as well as things that are supported. Let's start with the good news about primary constructors.

Records and Record Structs with Primary Constructors

I said above that you can't map a class with primary constructors using ComplexProperty. Well, happily, it works with the records! You can map a ComplexProperty with

the positional records (which are declared in their entirety by a primary constructor) and non-positional records that have a primary constructor. I also successfully tested this with positional record structs, positional read-only record structs, and non-positional record structs. I definitely prefer primary constructors over expression builders to instantiate an object.

JSON Support, or Lack Thereof, for Now

Although JSON support has been improving in EF Core, some of it thanks to Owned Entities, it does not exist yet for ComplexProperty.

For example, if PersonName is mapped as an owned entity, you can append the ToJson() method to the OwnsOne mapping resulting in the object being stored in some type of char field in your relational database table. A PersonName is stored as

```
{"FirstName ":"John","LastName":"Doe"}
```

ComplexProperty does not yet support this capability. Additionally, its inability to transform complex types to JSON also means that you cannot use ComplexProperty with the CosmosDb provider that stores all its data as JSON. Not yet. This is another feature that is tagged in the GitHub repo as "consider for current release," so hopefully that means EF Core 9.

Collections of Complex Types: Coming Soon

Owned Entity not only provides the OwnsOne mapping, but also OwnsMany. Therefore, it's possible to have a property in your entity that's a collection of the owned types. ComplexProperty doesn't yet support this, but the team has said it will be in EF Core 9. Keep in mind that value object collections are a disputed topic. Some call them an anti-pattern. But I've found some edge cases where they are quite useful. In fact, I have a collection of Author value objects in my EF Core and Domain-Driven Design course on Pluralsight. And even though I'm using EF Core 8 in the course, I still had to map that particular value object as an owned entity. Happily (and intentionally), the sample application in that course has another value object that provided a great example of using a record and ComplexProperty mapping.

Nested Complex Types: Also Coming Soon

That Pluralsight course also demonstrates nesting value objects. The Author value object has a PersonName property similar to the one I've been using in this article. Because Author is already mapped as an owned entity, I had to tack on its PersonName property as an owned entity as well. You definitely can't combine Owned Entities and ComplexProperty when nesting.

More importantly, even if Author was a ComplexProperty, nesting is not yet supported in EF Core 8. In the end, not only was I forced to use Owned Entity mappings for those two value objects, because they were owned entities, I had to declare them as classes, not records.

Is ComplexProperty Ready for Your Software?

I'm very happy to see and use ComplexProperty. Although it's not perfect yet, it already does solve many scenarios.

	ComplexProperty	Owned Entity
Class	Yes	Yes
Record	Yes	With side effects
Record Struct	Yes	No (must be a ref type!)
Record with Primary CTOR	Yes	Yes
Class with Primary CTOR	No	Yes
Collections	No (EF Core 9)	Yes (OwnsMany)
Nested	No (EF Core 9)	Yes
Data Annotation available	Yes	Yes
Store in its own table	No	Yes
Map to JSON column	No (EF Core 9?)	Yes
Seeding via DbContext/Migrations	No	Yes
Supported in Cosmos provider	No	Yes

Table 2: EF Core 8 support for ComplexProperty and OwnedEntity mappings

The question becomes (for some): Should you mix and match the two mappings? I think the answer is yes. I don't think it needs to be seen as a maintenance problem. What ComplexProperty currently solves, it does very well—and does so better than OwnedEntity. For the cases that you still need to use Owned Entity, continue to use them. But keep an eye on those cases because you'll be able to replace more (or all?) of those mappings when EF Core 9 comes out.

Table 2 provides you with a list of possible ways to define complex types and value objects and whether or not that expression is supported with ComplexProperty and Owned Entity mappings in EF Core 8. The EF Core team absolutely plans for a near future when we can completely eliminate OwnsOne and OwnsMany from our code. Until then, take advantage of the tool that works best for each scenario. And test, test, test.

The limitations are listed in the docs at this link (https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-8.0/whatsnew#current-limitations). Each has a link to the relevant issue on GitHub and you can let the team know which are important to you by voting for these issues in GitHub.

Julie Lerman
CODE

AI Executive Briefing

Experience the game-changing impact of **AI** through **CODE Consulting's** Executive Briefing service. Uncover the immense potential and wide-ranging benefits of **AI** in every industry. Our briefing provides **strategic guidance** for seamless implementation, covering crucial aspects such as infrastructure, talent acquisition, and leadership.

Discover how to effectively **integrate AI** and propel your organization into future success.

Contact us today to schedule your executive briefing and embark on a journey of AI-powered growth.
www.codemag.com/ai

ONLINE QUICK ID 2405051

Preparing for Azure with Azure Migrate Application and Code Assessment

There are many advantages to hosting ASP.NET and ASP.NET Core applications in Azure. Platform-as-a-Service (PaaS) environments like Azure App Service, Azure Kubernetes Service, and Azure Container Apps provide easy and automatic scalability, reliability, and availability in multiple geographies. What's more, these environments allow you to focus on the apps themselves without

Mike Rousos
mikerou@microsoft.com

Mike Rousos is a Principal Software Engineer on the .NET Customer Engagement Team. A member of the .NET team since 2004, he has worked on a wide variety of feature areas and contributed content to the .NET team blog, .NET Conf sessions, Channel 9 videos, and .NET development e-books like ".NET Microservices: Architecture for Containerized .NET Applications." Outside of work, Mike is involved in his church and enjoys reading, writing, and games of all sorts. His primary hobby, though, is spending time with his four kids.

the overhead of maintaining underlying infrastructure. But for apps that are already deployed on-premises, it can be difficult to know how to get started re-platforming to one of these environments. Even though .NET applications can often by deployed to Azure App Service with minimal changes, there are usually **some** changes required and discovering what those are can take some trial and error.

This article introduces a new feature: Azure Migrate application and code assessment. This new feature allows analyzing the source code, configuration, and binaries of an application to discover upfront what changes will be needed for the app to work in Azure. Azure Migrate application and code assessment make it easy to plan re-platforming to Azure and highlights what work will be needed along the way. Azure Migrate application and code assessment is a developer-focused experience available as both a Visual Studio extension and a command line .NET SDK tool. The tool scans ASP.NET and ASP.NET Core solutions (and, optionally, their binary dependencies) for a wide variety of potential issues that need to be addressed prior to running in Azure PaaS environments.

Although this article focuses on the experience of using Azure Migrate application and code assessment for .NET, there is also a Java version of the tool available. To learn more about the Java experience, please visit https://learn.microsoft.com/azure/developer/java/migration/appcat.

Relationship to Other Azure Migrate Features

Using Azure Migrate to prepare for a migration to the cloud isn't new, of course. Azure Migrate has helped users discover and assess on-premises infrastructure for some time. Azure Migrate also includes the Data Migration Assistant to help users assess SQL Server databases for mi-

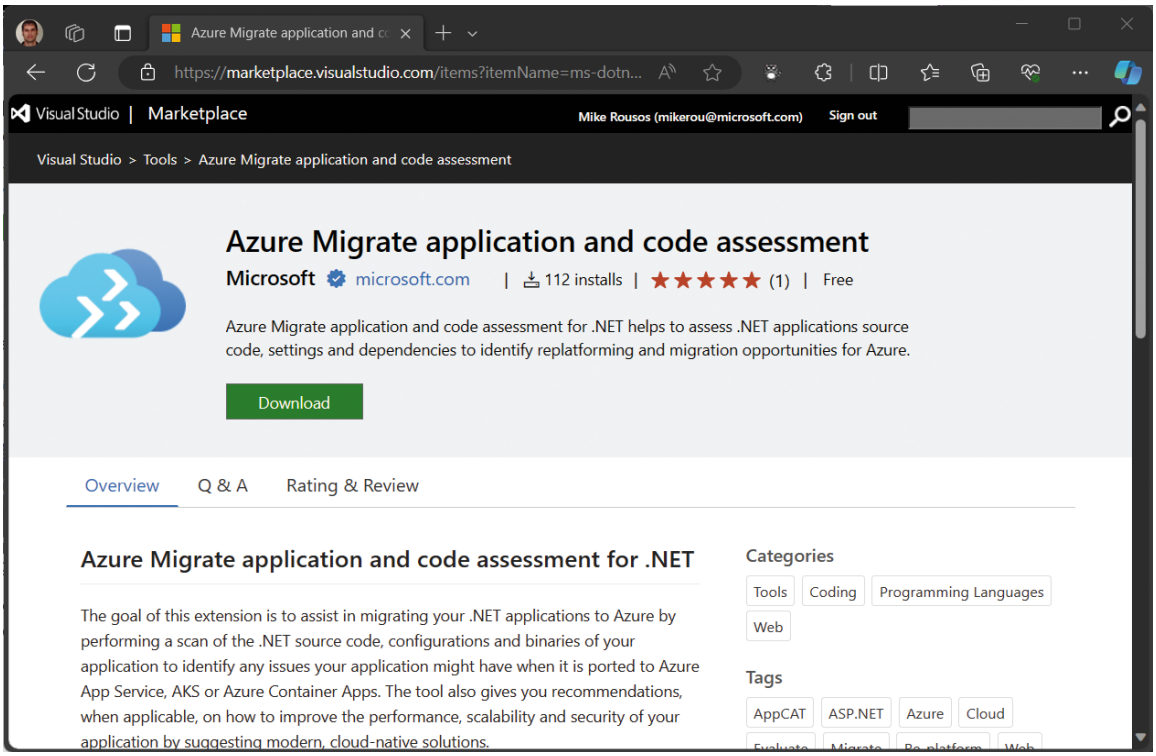

Figure 1: Azure Migrate application and code assessment extension download page

gration to Azure SQL DB, Azure SQL Managed Instance, or SQL Server on an Azure VM. Azure Migrate can even discover web apps hosted on-premises and (if no blocking issues are detected) automate migrating them to Azure with its App Service Migration Assistant tool.

Up until now, though, Azure Migrate hasn't had the ability to look at the source code of the applications it detected. The App Service Migration Assistant tool can provide insight into whether there are likely to be issues migrating to Azure based on IIS configuration, but it doesn't have visibility into what's happening inside the code of the application. The Azure Migrate application and code assessment feature fills this gap. The Visual Studio extension or command line tool can assess your source code of your solution and identify potentially problematic APIs and code patterns. This assessment is a useful follow-up to the discovery by other Azure Migrate features. It is recommended to use Azure Migrate's existing application discovery features to gain insight into the complete set of applications to be migrated. Then, after the applications have been reviewed and prioritized, Azure Migrate application and code assessment can be used by developers to dive deeply into the applications that will be moved to learn more details about potential issues and to plan the migration.

Installing Azure Migrate Application and Code Assessment

As a Visual Studio extension, the Azure Migrate application and code assessment feature is easy to install. It's available on the Visual Studio Marketplace at https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.appcat, as shown in **Figure 1.** From that site, use the download button to download the .vsix installer, and then open the installer and wait while it installs the Visual Studio extension.

Note that the Azure Migrate application and code assessment Visual Studio extension requires Visual Studio 2022. If you don't have Visual Studio 2022, the free community edition is available at https://visualstudio.microsoft.com/downloads.

Analyzing a Solution

To begin analyzing a solution, open the solution in Visual Studio and right-click on it in the solution explorer. With the Azure Migrate application and code assessment extension installed, there will be a new command available: **Re-platform to Azure**, as shown in **Figure 2**. Choosing this option opens a new user interface for managing Azure Migrate application and code assessment reports.

After clicking New Report, you will be able to choose which projects in the solution you wish to assess, as shown in **Figure 3**. Note that the Azure Migrate application and code assessment feature automatically analyzes dependencies of selected projects, so you only need to choose top-level projects you're interested in. All recursive project-to-project dependencies of the selected projects will be included automatically.

After clicking next, the following UI will allow you to choose whether to scan only source code and settings or

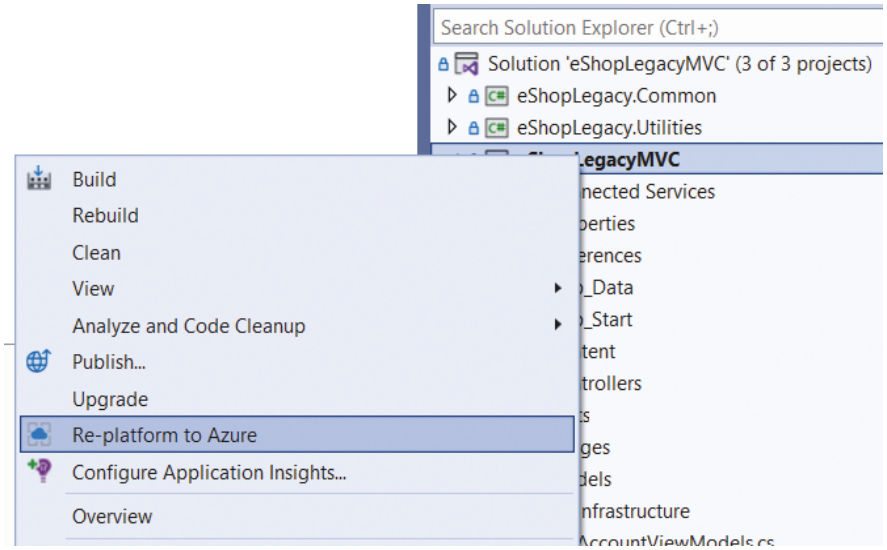

Figure 2: The Re-platform to Azure command opens the new tool UI.

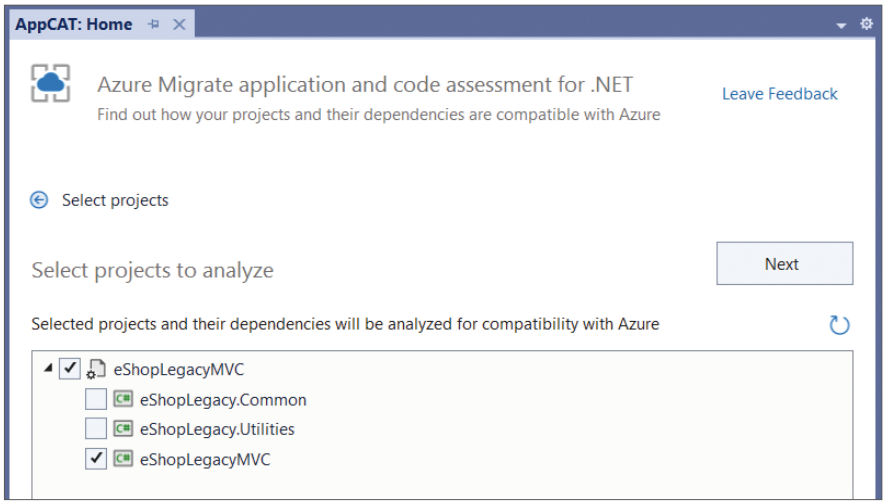

Figure 3: Selecting projects to be analyzed for Azure migration readiness

also binary dependencies, as shown in **Figure 4**. Choosing to analyze source code will look at the source code of the selected projects—C# or Visual Basic code files, project files, config files, and static content. This option should typically be checked. If you also check the **Binary dependencies** option, the tool also analyzes binaries that the projects depend on. This includes references to loose DLLs, references to assemblies (other than .NET Framework components) in the global assembly cache, or referenced NuGet packages. Choosing to analyze binaries produces the most comprehensive set of issues the application may run into while migrating, but it also includes more issues than analyzing only source code and may include issues that you're not able to fix directly (because they exist in components you don't have source code for) or that aren't relevant if they exist in code paths not used from your application. A useful strategy is to begin only by analyzing source code and later consider producing another report with binary analysis enabled if there are binary dependencies whose Azure-readiness you're unsure of and would like to learn more about. Because issues in binary dependen-

cies can't be fixed directly by updating source code, they are typically addressed by finding updated versions of the binaries, working with partners who can change the source code, or finding alternative solutions that work better in Azure.

Once you click the Analyze button, the extension analyzes the selected projects for any potential issues re-platforming to Azure. This analysis will take anywhere from a few seconds to a few minutes, depending on the size of the projects. When the analysis is complete, you'll be shown a report summarizing the results. (See **Figure 5**.) This report can be saved to disk and returned to later using the save icon (or common shortcuts like Ctrl+S).

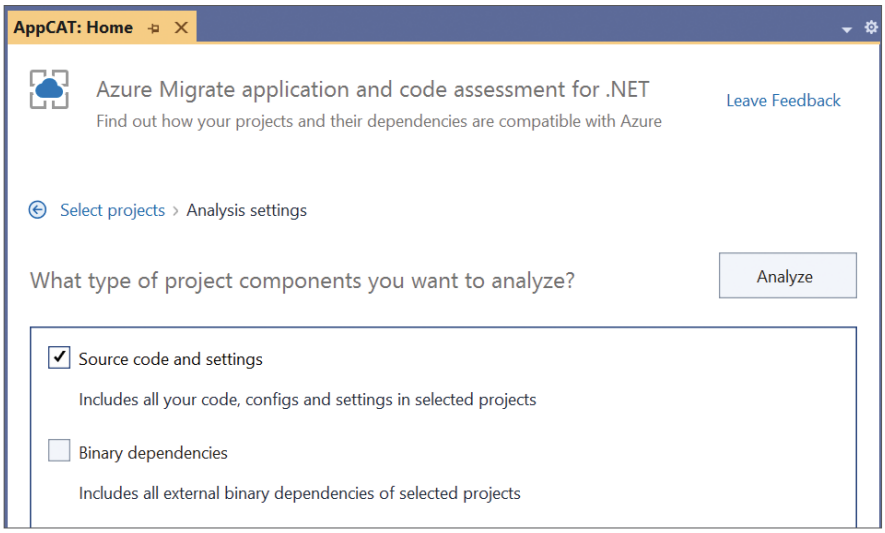

Figure 4: Choosing whether to analyze source code or binaries

The report's dashboard includes the total number of projects scanned, the number of incidents discovered, and graphics showing the incidents by category and severity. The report include both a number of issues that are the types of problems detected and a number of incidents that are the individual occurrences of the issues.

Azure Migrate application and code assessment issues are each assigned one of four severities:

1. **Mandatory**: Mandatory issues are those that likely need to be addressed before the application will work in Azure. An example of a mandatory issue is using Windows authentication to authenticate web app users. Because that authentication mechanism depends on the on-premises Active Directory environment, it will likely need to be updated in the cloud to use Azure AD or some other authentication alternative.
2. **Optional**: Optional issues are opportunities to improve the application when it's running in Azure, but they aren't blocking issues. As an example, storing app settings or secrets in a web.config file is considered an optional issue. That pattern will continue to work when deployed in Azure, just like on-premises, so no changes are required. Apps that are hosted in Azure can take advantage of services like Azure App Configuration and Azure Key Vault to store settings in ways that are easier to share and update and that are more secure. So, there's an optional issue to begin taking advantage of these services as part of the Azure re-platform.
3. **Potential:** Potential issues represent situations where there **might** need to be a change made for the app to work in Azure, but it's also possible that no change is needed, depending on the details of the scenario. This severity is common and requires an engineer to review

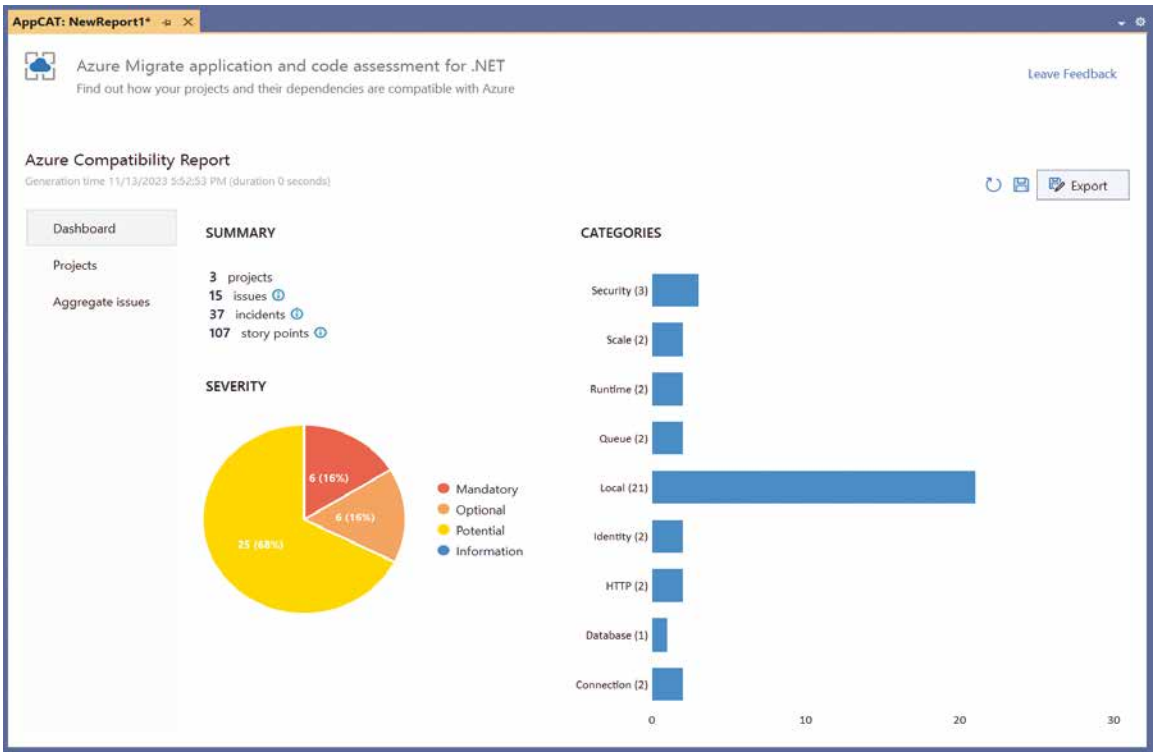

Figure 5: The Azure Migrate application and code assessment report dashboard

the incidents. As an example, connecting to a SQL Server database is a potential issue because whether a change is needed depends on whether the database that's used is accessible from Azure. If the database is already hosted in Azure or is accessible from Azure (via Express Route, for example), no changes are needed. On the other hand, if the database used exists on-premises without a way for the app to connect to it once it's running in Azure, thought will need to be given to how this dependency will work after the app is re-platformed. Perhaps the database will need to be migrated alongside the app or perhaps a solution like Express Route or Hybrid Connections will be needed to make the database accessible.

4. **Information**: Information issues are useful pieces of information for the developer to know but don't require any action. As of the time of this writing, there aren't any information issues in Azure Migrate application and code assessment for .NET (although there are a couple in the Java version of the tool).

In addition to severity, each incident in the report includes a story point number. This is a unitless number representing the relative effort estimated to address the incident (if, in fact, it needs to be addressed). These shouldn't be used to estimate the precise amount of work in terms of hours or days but can be used as a rough estimate for comparing two projects. If one solution has 500 story points worth of issues and another has 200 story points worth of issues, it's probably true that the solution with fewer story points of issues will be simpler and easier to re-platform.

From the initial dashboard, you can navigate to views displaying aggregate issues (all incidents organized by issue type) or projects (incidents organized by project). When viewing incidents for a particular project, you can choose to view all incidents for the project or to view incidents per component (a single source file or binary dependency is considered a component).

In incident detail views, there will be a state drop-down box indicating whether each incident is Active, Resolved, or Not Applicable (N/A). All incidents begin as Active and you can change the state as you investigate. Reports can be saved (using the save icon in the top right of the report) and the state will be persisted so that you can return to the same report in the future and continue to review remaining issues and further update the state. As you review the incidents in the report, mark incidents that don't need to be addressed in your solution as not applicable and those that you've fixed as resolved. The incident detail pages also give descriptions of why the incidents were identified, why they matter, and how you can address them. These detail views, shown in **Figure 6**, include links to documentation and links to the locations in source code where the issues were detected.

In addition to working with reports in the Visual Studio IDE, it's possible to export the reports to share with others. The Export button in the top right of the report interface allows you to export the report in three different formats:

- Export as HTML produces the most readable report for sharing with others. The HTML report, shown in **Figure 7**, has all the same dashboards and views as the Visual Studio UI and includes snapshots of the latest state of all incidents. This report is best for sharing with others for viewing issues and investigation progress.
- Export as CSV produces a report with the same information but in a spreadsheet format. As seen in

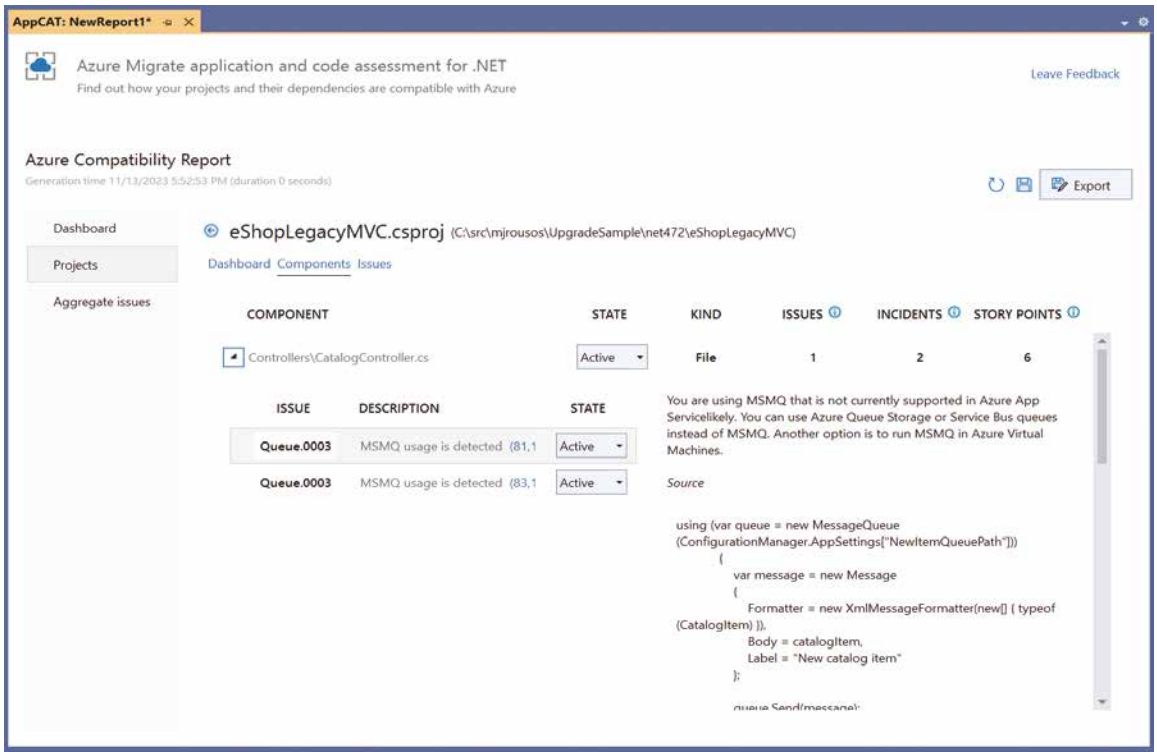

Figure 6: Azure Migrate application and code assessment issue details

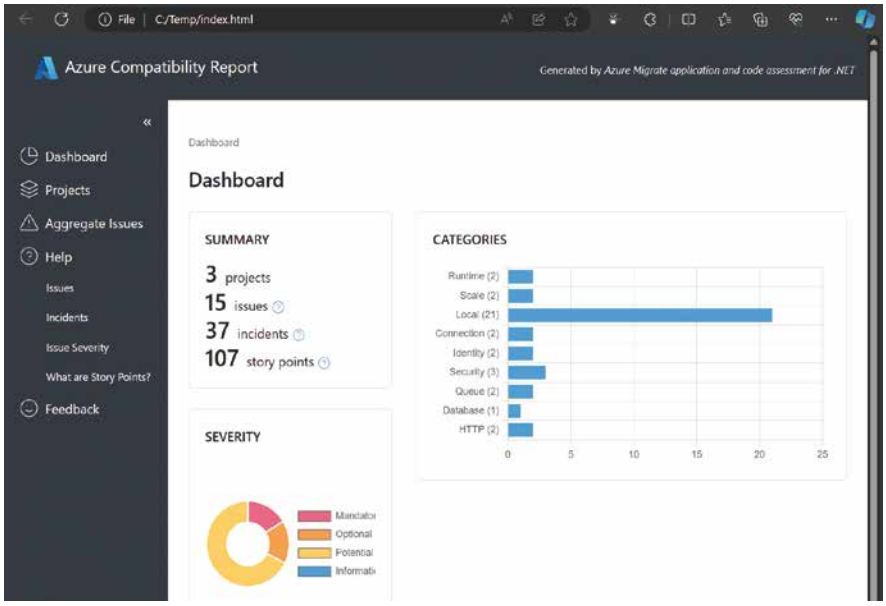

Figure 7: Azure Migrate application and code assessment HTML report

Figure 8, this report format doesn't have the nice charts and graphics of the HTML view but can be useful when you need a simple spreadsheet representation of the issues that you can annotate and update as you discuss and review the incidents.

- Export as JSON produces a machine-readable JSON representation of the incidents. This export option isn't meant for human consumption. Instead, this option produces JSON reports that can be parsed by other applications programmatically.

Not every incident in the report requires action. It's common for Azure Migrate application and code assessment reports to include many potential issues that don't actually require changes. And some of the issues will be optional. The best way to think about the reports is as a helpful resource listing parts of the application requiring review. Once you've looked at the highlighted parts of the application, you can have confidence that you understand what work (if any) is required prior to re-platforming it to Azure.

Example Walkthrough

As an example, I've used the Azure Migrate application and code assessment feature to analyze the updated eShop **Northern Mountains** sample application found at https://github.com/dotnet/eshop. eShop is a multi-service ASP.NET Core e-commerce solution. Although it does have a number of external dependencies, the sample was created with cloud deployment in mind so there shouldn't be too many issues to address aside from the identification of the external dependencies.

Because the eShop sample is comprised of multiple services, I needed to make sure to select all entry points when choosing projects to assess, as shown in **Figure 9**.

Assessing the source code was quick. Even with its multiple projects, the eShop sample isn't large, so analysis finished in just a few seconds. Altogether, 16 projects were scanned and 41 total incidents of nine different types of issues were detected. Of the 41 incidents, eight had mandatory severity, eight had optional severity, and 25 had potential severity. This ratio of issue severities is typical. The report dashboard is shown in **Figure 10**.

Normally, I like to review incidents in Azure Migrate application and code assessment reports one project at a time. In this case, though, with a relatively low number of total incidents, it's just as easy to use the **Aggregate Issues** view and review them all at once. Here are the issues that were identified in the eShop sample:

- **RabbitMQ usage**. The only mandatory incidents are eight instances of RabbitMQ usage detected in the EventBusRabbitMQ project and shown in **Figure 11**.

	A	B	C	D	E	F	G
1	IssueId	Description	State	Severity	Story Points	Project Path	Locatic
2	Runtime.0002	Upgrade to newer target framework to get better c	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacy.Con	File
3	Runtime.0002	Upgrade to newer target framework to get better c	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacy.Utili	File
4	Scale.0001	Static content detected	Active	Optional	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
5	Local.0007	Local OS environment access detected	Active	Potential	1	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
6	Local.0004	Logging to local or network paths detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
7	Connection.0001	Connection string is detected	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
8	Connection.0001	Connection string is detected	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
9	Identity.0002	Windows authentication detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
10	Identity.0002	Windows authentication detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
11	Scale.0002	Session state stored in-proc or in local process is	Active	Optional	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
12	Security.0002	Connection strings without configuration builders	Active	Optional	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
13	Security.0002	Connection strings without configuration builders	Active	Optional	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
14	Security.0003	Hardcoded sensitive data detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
15	Queue.0003	MSMQ usage is detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
16	Queue.0003	MSMQ usage is detected	Active	Mandatory	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
17	Local.0003	Local or network IO operations detected	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
18	Database.0001	Database dependency detected	Active	Potential	5	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
19	HTTP.0001	Access to external resources via HTTP is detected	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
20	Local.0006	Hardcoded URLs detected	Active	Potential	1	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File
21	Local.0003	Local or network IO operations detected	Active	Potential	3	C:\src\mjrousos\UpgradeSample\net472\eShopLegacyMVC	File

Figure 8: Azure Migrate application and code assessment CSV report.

As explained in the issue's description, these incidents relate to a dependency on a RabbitMQ queue that will need to be made available in Azure as part of re-platforming efforts. Several potential strategies are presented, either using an alternative messaging system like Azure Service Bus or running a RabbitMQ cluster in Azure. In order for eShop to work properly in the cloud, though, it will be necessary to follow one of these suggestions to make the messaging done in EventBusRabbitMQ work.

- **Database usage**: The next most common issue is the eight incidents of database usage detected across several different eShop projects. In all these cases, the assessment has found data being read from or written to databases using Entity Framework Core. The incidents' descriptions explain that I need to ensure that the database used will be available from the Azure environment I migrate the eShop solution to. Similar to the RabbitMQ incidents earlier, these relate to an external dependency that needs to be accessible for eShop to work properly.
- **Connection strings:** The next issue category I looked at was connection strings. Spread across five projects, there were six incidents of connection strings being detected in configuration files. These were strings like **EventBus: amqp://localhost** and **Redis: localhost.** Once again, these are indications of dependencies the eShop solution has on services outside its own processes. The connection strings point to localhost but AMQP and Redis services will not be available locally when run in an Azure App Service or AKS environment. As before, these incidents are reminders to make sure you have messaging and caching services available for eShop to use in Azure and that configuration is updated when deploying to Azure to take advantage of those services.
- **Hardcoded URLs**: The next issue type I looked at were the five incidents of hardcoded URLs. In the case of eShop, these were all references to other eShop services that will be invoked via the Aspire framework. URLs included http://catalog-api/health and http://basket-api. Because these services are made accessible via Aspire, the URLs will continue to work in Azure and there aren't any changes needed in the application. I can mark these incidents as N/A to indicate that they're false positives. If there had been external URLs in use, I would have had to review the URLs to make sure that the services they represented would be available in eShop's new environment.
- **Caching:** The next most common issue type was the five incidents of caching APIs being used. In eShop's case, these were all in the Basket.API project and represented Redis usage that the basket API uses to maintain the user shopping basket state. As explained in the issue description, I need to ensure that a Redis instance is available for the basket API to use in the cloud and should also explore using an external caching solution like Azure Cache for Redis so that cached state can be shared between instances of the basket API service if I need to scale out to multiple instances.
- **HTTP usage:** The report also includes four incidents of outgoing HTTP calls being made. Much like many of the other incidents, these potential incidents represent an external dependency that I need to review to ensure that the referenced services will be available when deployed to Azure. Looking at the code, I see that these are the same calls that use the hardcoded URLs reviewed earlier. Those incidents were about the hardcoded URL strings whereas these ones are about the HTTP client API usage, but both relate to the same dependency, so these items have already been reviewed.
- **Local file usage**: The final potential issue reported is a single incident of file IO occurring in the Catalog.API project. I see from the incident details that the project is calling **File.ReadAllText**. To ensure that the accessed file path will be available from Azure, I need to review the file (CatalogContextSeed.cs) where the call occurs. I can click the link included in the incident report to navigate directly to the relevant code in my solution. Doing so, I see that this call is part of seeding the database with information on first run and that the file read is deployed alongside the application, so everything should work just as well in Azure as it did on-premises. This one, also, can be marked N/A.
- **Static content**: The last remaining set of issues are three optional incidents about several eShop projects serving static content. These incidents are optional because no action is required. However, the issue description explains that once deployed to Azure, there may be performance and scalability benefits of serving static content such as the files identified here using Azure Blob Storage and Azure CDN.

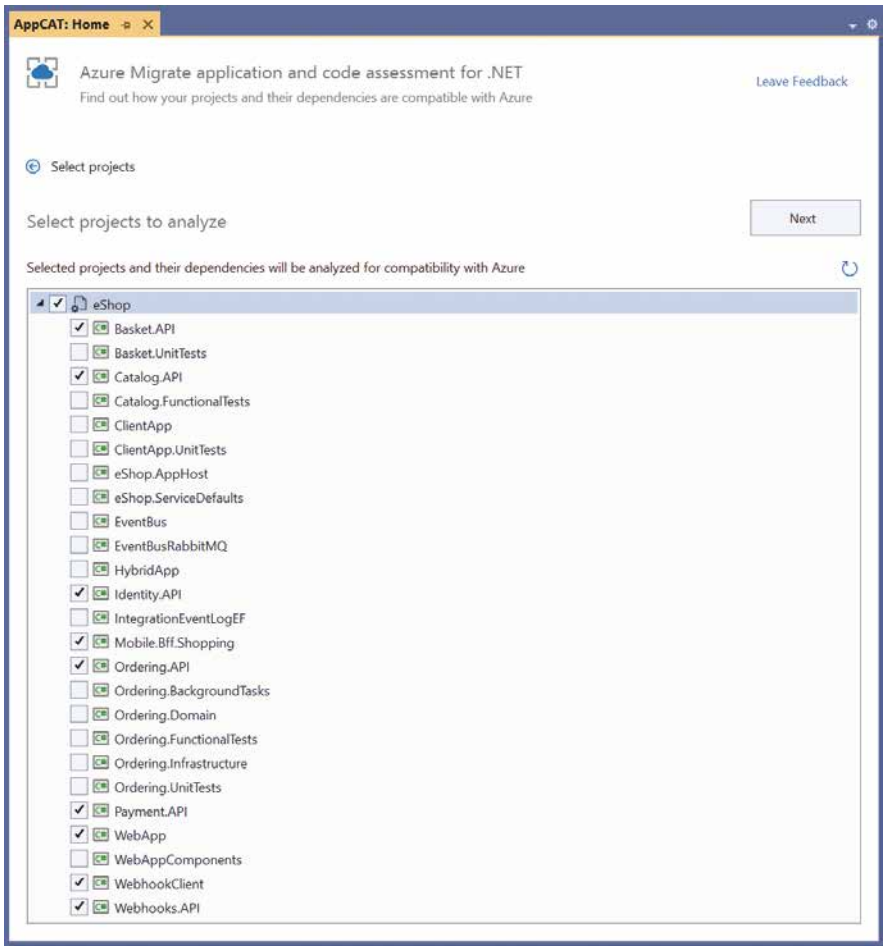

Figure 9: Assessing the updated .NET eShop sample with Azure Migrate application and code assessment

Figure 10: Assessment results for the eShop sample solution

Those are all of the issues detected in the eShop sample. It's a larger list than initially expected, perhaps, but, as we thought, the issues were almost entirely about external dependencies that will need to be migrated to Azure along with the eShop solution. There was nothing blocking in the report except for the helpful reminders that the solution will need database, message queue, and caching solutions in the cloud that may look different from on-premises, and provisioning those (and updating the app to use them) needs to be part of the re-platforming plan.

Using the Command Line Interface

In addition to the Visual Studio extension discussed in the rest of this article, the Azure Migrate application and code assessment feature is also available via a .NET command line tool. The Visual Studio extension offers the most functionality (because the reports maintain state for which incidents have been reviewed), but all the same assessment capabilities are available from the command line, meaning that analysis doesn't need to depend on Visual Studio and can be scripted, if needed.

Like all .NET SDK tools, the Azure Migrate application and code assessment CLI tool is installed using a .NET CLI command:

```
dotnet tool install -g dotnet-appcat
```

That command pulls down a NuGet package containing the latest version of the dotnet-appcat tool and installs it globally. The tool is run using the **appcat** command. The simplest way to use the Azure Migrate application and code assessment CLI is to run **appcat analyze solution.sln** where solution.sln is the path to the solution or project file you want to assess. Running analyze without a project or solution specified causes the tool to search for projects in the current directory. This command starts an interactive command line experience, allowing you to choose which projects from the solution you want to assess, whether you want to analyze only source code or source code and binaries, and what output format you want (HTML, CSV, or JSON)—all the same options as with the Visual Studio extension! See **Figure 12**.

Once the appcat CLI tool gathers the necessary data from you, it proceeds with analysis and publishes results to a report using your desired format.

One important note about the CLI experience is that the solution must be able to build without errors. Visual Studio can provide the necessary symbol information for analysis, but when running from the command line instead, the target project must be in a buildable state.

In addition to mimicking the Visual Studio experience, the Azure Migrate application and code assessment command line tool accepts parameters that allow all decisions to be made up-front so that the tool runs completely automatically, allowing for a non-interactive experience suitable for scripting. To do this, use the --non-interactive parameter and make sure to specify report format and components to analyze via the command line. Here's an example command line for assessing a C# project non-interactively:

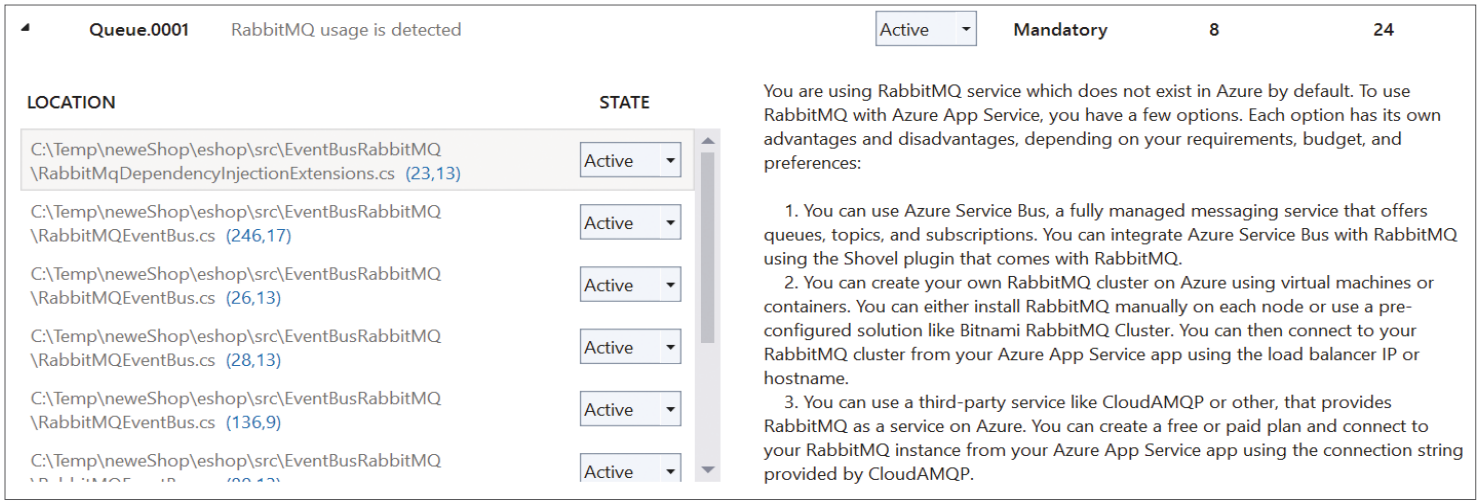

Figure 11: RabbitMQ related issues in eShop

```
appcat analyze eShopLegacyMVC.csproj -s HTML
-r OutputPath --code --non-interactive
```

This command assesses the eShopLegacyMVC.csproj project, only considers source code (thanks to --code), and puts an HTML report in the OutputPath folder. The output is the same as the Azure Migrate application and code assessment Visual Studio extension produced and it all runs automatically from the command line as shown in **Figure 13**.

Road Map

The Azure Migrate application and code assessment feature already provides insights necessary to plan a successful Azure migration. In the future, though, there are even more useful features planned. Key features coming in future versions of Azure Migrate application and code assessment include:

1. **GitHub Copilot Chat integration**: Azure Migrate application and code assessment is great at identifying points of interest in an application that should be reviewed to be prepared for re-platforming to Azure. It's only an analysis tool, though, and guidance that's given on how to address issues is static. By partnering with GitHub Copilot Chat, it will be possible for users to have reports from Azure Migrate application and code assessment summarized for them and to have back-and-forth conversations about the best way to review and remediate issues. GitHub Copilot Chat will be knowledgeable about the issues discovered by Azure Migrate application and code assessment and will be able to provide step-by-step instructions for configuring Azure resources, updating code, and more, to address them.
2. **Multiple Azure targets:** A feature available for the Java version of Azure Migrate application and code assessment that isn't yet available for the .NET version of the tool is the ability to specify a desired Azure target environment during analysis (for example, Windows Azure App Service or AKS with a Linux container). The issues that apply to different Azure environments overlap a lot and, currently, Azure Migrate application and code assessment for .NET assesses a general Azure PaaS environment that covers issues of interest in any potential target. But this feature will be coming soon to Azure Migrate application and code assessment for .NET. In the next version, you will be able to specify an Azure environment to target and issue severity levels and descriptions will be updated for that environment.
3. **Binary output analysis:** Today, Azure Migrate application and code assessment can scan binary dependencies referenced by a solution. But there still has to be a source solution as a starting point. In the next version of the tool, the command line interface will allow scanning compiled binaries in a given folder directly.

Wrap-Up

Azure Migrate has been helping developers migrate solutions from on-premises to Azure for years. With the addition of the Azure Migrate application and code assessment feature, that migration help now extends to understanding the inner workings of your solution and source-level changes that are needed for the application to work in Azure platform-as-a-service environments!

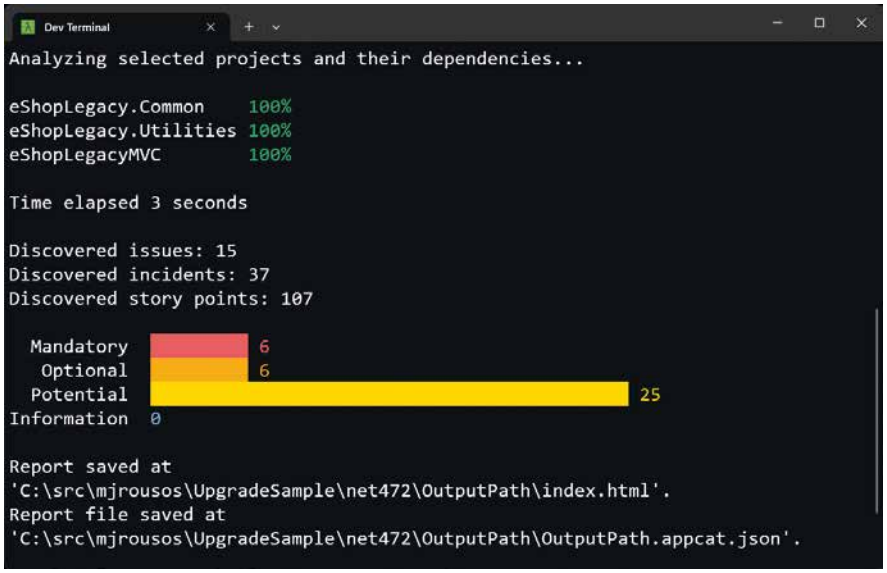

Figure 13: Azure Migrate application and code assessment command line output

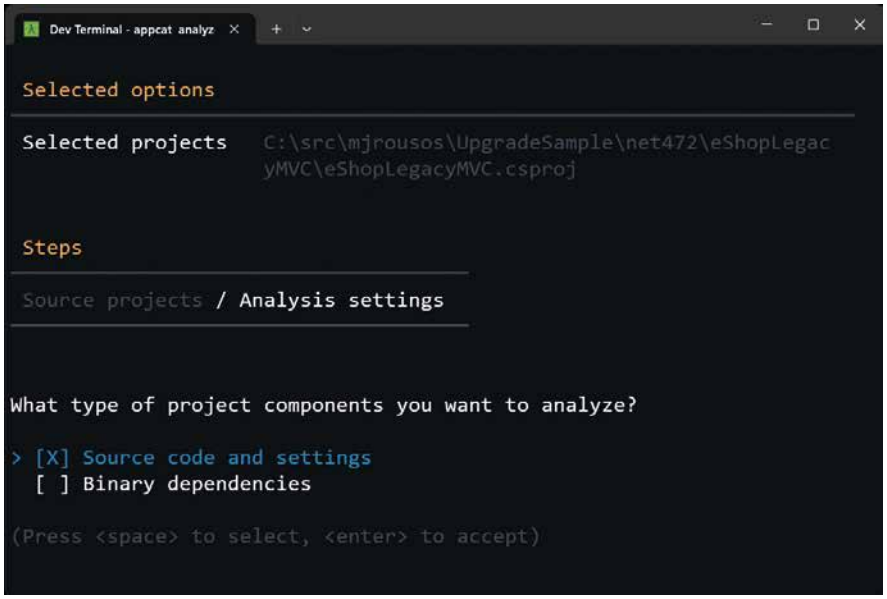

Figure 12: The Azure Migrate application and code assessment CLI offers the same options as the Visual Studio extension.

You can learn more about Azure Migrate application and code assessment from documentation available at https://learn.microsoft.com/azure/migrate/appcat/overview. As you use the new feature, please send feedback if you have ideas for how the experience could be improved or if you run into any bugs. The feature is brand new, so there are bound to be opportunities to improve it as customers use the Visual Studio extension and CLI and share their experiences. To get instructions on how to leave feedback, click the Leave Feedback link in the top right corner of the Azure Migrate application and code assessment's Visual Studio UI or use the Q&A tab of the Visual Studio marketplace page at https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.appcat.

Mike Rousos

CODE

Stages of Data: The DNA of a Database Developer (Part 1)

Long before I started in IT, I wanted to teach. The process of learning something, then performing it, crafting it, learning from mistakes, further crafting it, and sharing that journey with others appealed to me as a great career. Rod Paddock (Editor-in-Chief for CODE Magazine) recently told me that the next issue of CODE would deal with databases. At the same time, I've been seeing many questions on LinkedIn that boil down to, "How do I increase my SQL/database skills to get a data analyst/database developer job?" In this industry, that question is complicated, as it means different things to different people. It's arrogant to claim to have all the answers, because doing so would presume that someone knows about every job requirement out there. Having said that, I've worked in this industry for decades, both as an employee and as a contractor. I'd like to share what skills have helped me get and keep a seat at the table.

Kevin S. Goff
www.KevinSGoff.net
@StagesOfData

Kevin S. Goff is Database architect/developer/speaker/author, and has been writing for CODE Magazine since 2004. He was a member of the Microsoft MVP program from 2005 through 2019, when he spoke frequently community events in the Mid-Atlantic region and also spoke regularly for the VS Live/Live 360 Conference brand from 2012 through 2015.

Lead article

Opening Remarks: A Rebirth of SQLServer Skills

Over the last few years, there have been intense opportunities and job growth in the general area of data analytics. That's fantastic! There's also been a reality check of something that database professionals warned about last decade: the need for those using self-service BI tools to have some basic SQL and data management skills. As part of research, I spent a substantial amount of time reading LinkedIn posts, and talking to recruiters and other developers about this, and there's one common theme: There's still a booming need for SQL and data handling skills. I want to make a joke about being an old-time SQL person and a "boomer," but I was born one month after the official end of the boomer generation.

I'm a big sports and music fan and I often hear people talk about the "DNA" of great athletes and musicians. In this context, the definition isn't referring to the biological aspects of a person. It's more the traits they carry with them and the habits they've burned into themselves that they leverage regularly to do their jobs and do them well. I certainly hope that everyone who's willing to work hard will get a job, and I'm equally excited that I've seen a resurgence of "what SQL Server skills should a data person have?"

I know people who build great websites and great visualizations in reporting tools, where the available data was very clean and prepared by an existing data team. However, for every one individual job out there like that, there's more than one job where you'll have to put on your SQL/data-handling hat.

I've worn multiple hats in the sense that I've always had work. I make mistakes, I underestimate, I still commit many silly errors that we all, as developers, wish we could avoid. Although no one person can possibly cover every possible SQL/data skill that will make someone successful, I stepped back and thought, "What has helped me to help clients? What has been the difference-maker on a project?" And that's why I decided to write this article. This will be a two-part article. Throughout both, I'm going to mention some topics that I covered in prior CODE Magazine articles where the content is still just as relevant today.

You can find many web articles with titles like, "Here are the 30 best SQL Server interview questions you should be prepared for." There are many good ones and I recommend reviewing them as much as you can. I also recommend getting your proverbial hands dirty inside of Microsoft SQL Server Management Studio. On that note, I'm using Microsoft SQL Server, which means I'll be covering some Microsoft-specific topics. Having said that, many of the topics in this article are relevant to other databases.

I didn't want to call this article "The 13 things you should study before a SQL interview," because I'm going beyond that. I'm covering what I think makes for a good SQL/database developer. Yes, there's overlap, as I want to share some of the specific skills companies are often looking for.

First, Know Basic SQL

"Knowing basic SQL" is really two things: understanding the SQL language (according to the ANSI SQL standard) and understanding specific features in the database product (in this article, Microsoft SQL Server) and some of the physical characteristics of Microsoft databases. There are great books out there, but these topics tend to come up again and again. I'll start with some index basics, even before getting into some language basics.

Know the Different Types of Indexes

A common question is the difference between a clustered index and a non-clustered index. With this and other topics in this article, I'm not going to write out a full definition, because other websites have done a great job. But here are things I think you should know.

You can only create **one clustered index per table** and that index defines the sorted order of the table. For a sales table that might have millions of rows, a clustered index on either the sale transaction ID or possibly the sales date will help queries that need to scan over many rows in a particular order.

You can have **many non-clustered indexes.** They serve the purpose for more selective queries: that is, finding sales between two dates, finding sales for specific products, specific geographies, etc. A non-clustered index might contain just one column (composite index), or it could contain multiple columns if you'll frequently need to query on a combination of them.

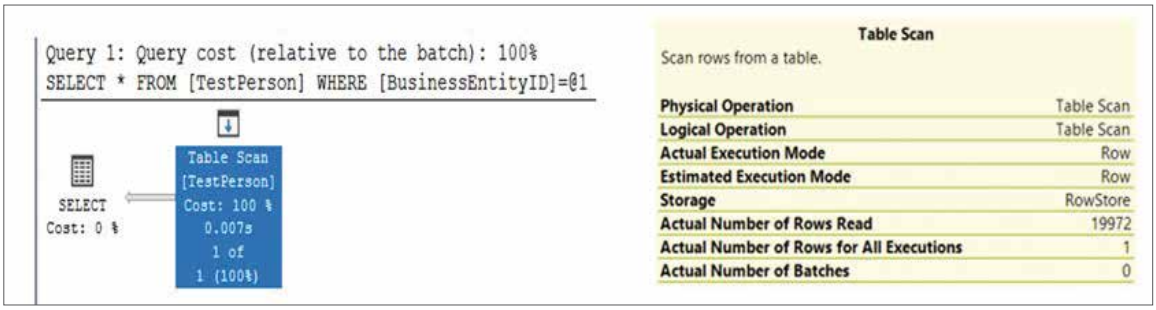

Figure 1: Execution plan, only the execution operator is a Table Scan with statistics on the number of rows read

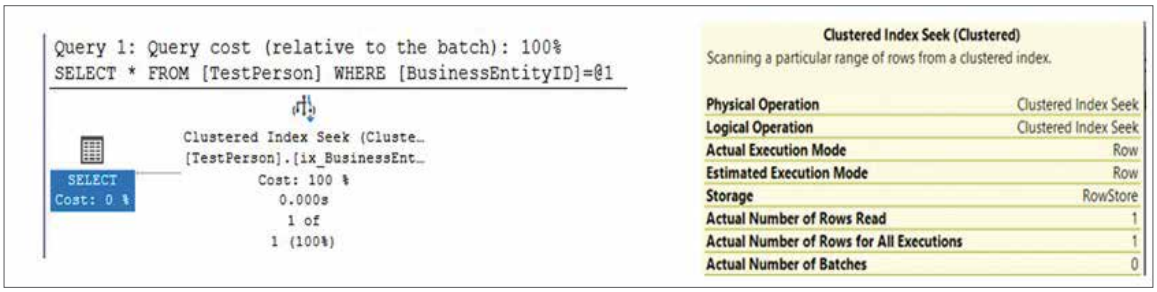

Figure 2: Execution plan with an INDEX SEEK, which is far more efficient (only one row read)

Related: Know When SQL Server Will Use These Indexes: Just because you create an index doesn't mean SQL Server automatically uses it. For instance, you might create indexes that SQL Server won't use, either because the SQL statement you're using isn't search argument optimizable or because you might not realize what something like compound indexes will (and won't) do.

I'm going to take the table **Person.Person** from the **AdventureWorks** database and create my own table. I'll also create two indexes: a clustered index on the primary key (**BusinessEntityID**) and a non-clustered index on the Last Name and the First Name.

```
drop table if exists dbo.TestPerson
go
select * into dbo.TestPerson from Person.Person
```

Now I'll use a single query to retrieve the row for a specific Business Entity ID:

```
select * from TestPerson where BusinessEntityID
  = 12563
```

In the absence of any index, SQL Server must perform a table scan against the table and read all 19,972 rows. Here's what SQL Server returns for an execution plan (**Figure 1**).

Although the query runs in a split second, SQL Server had to read through all the rows. It's not exactly optimal.

Now let's create a clustered index that drives the sort order of the table:

```
create clustered index
    [ix_BusinessEntityClustered] on TestPerson
        (BusinessEntityID)
```

If I run the same query again and then look at the execution, I'll see a very different story: a INDEX SEEKCLUSTERED INDEX SEEK where SQL Server only needed to read one row (**Figure 2**).

Next, I'll query the table for all names where the last name is "Richardson".

```
select * from testperson
  where lastname = 'Richardson'
```

Does the clustered index help us at all? Unfortunately, not really. Although SQL Server scans a clustered index instead of a row table (heap), SQL Server must scan all 19,972 rows (**Figure 3**).

To help with queries based on a last name, I'll create a non-clustered index on the Last Name column.

```
create nonclustered index  [ix_LastName]
        on TestPerson ( LastName)
```

After creating the index on Last Name, let's query for a specific last name, and then for a specific last name and first name:

```
select * from testperson
   where lastname = 'Richardson'

select * from testperson
   where lastname = 'Richardson'and
      firstname = 'Jeremy'
```

In the case of the first query, SQL Server uses a more efficient INDEX SEEK on the new index. However, it does need to perform what's called a KEY LOOKUP into the clustered index, to retrieve all the columns (because I did a SELECT * to ask for all the columns).

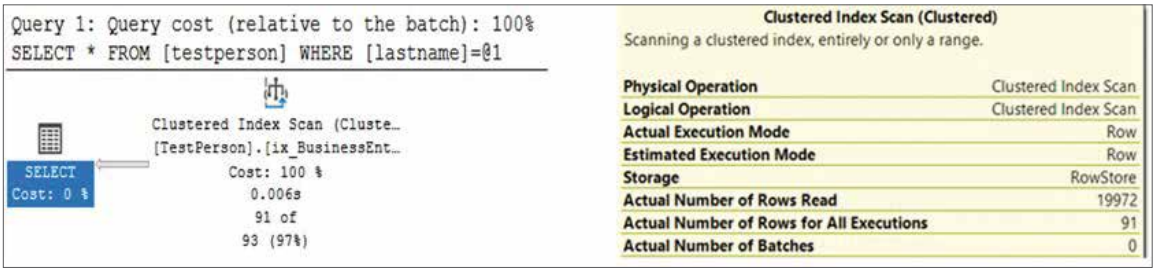

Figure 3: Execution Plan, now showing a Clustered Index scan on all 19K rows

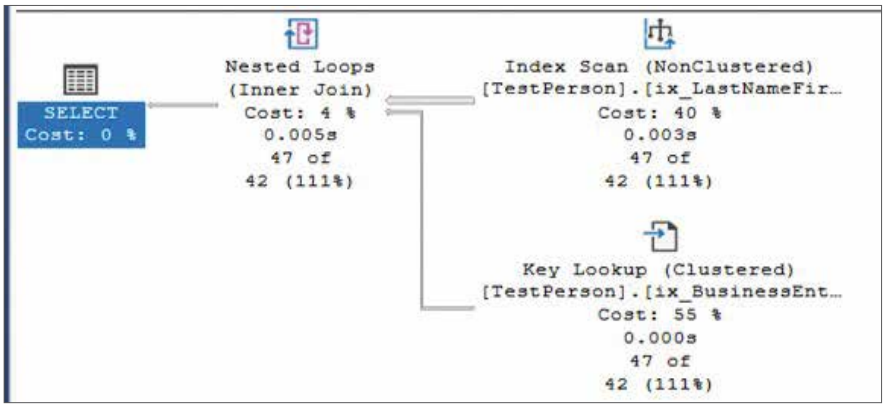

Figure 4: Execution Plan, and you're back to a scan, because the query couldn't use the index

What happens when you query both a first and last name? You only retrieve one row, so the server does less work to return the results back to the application. However, under the hood, you'd see something different. Yes, SQL Server still performs an INDEX SEEK but must perform the filtering on the first name when it reads the 99 rows from the clustered index. In this case, the non-clustered index certainly helped to narrow the search, but SQL Server still needed to "scan" through the 99 rows to find all the instances of "Jeremy".

Okay, so what happens if you create a composite index on the Last name and First name?

```
create nonclustered index  [ix_LastNameFirstName]
        on TestPerson ( LastName, FirstName)
```

If you run the query again to retrieve both the last name and the first name, SQL Server performs an INDEX SEEK and reads just one row into the Key Lookup to get all the columns for that single person. You're back in optimization heaven.

Okay, so a composite index further optimizes the query. Here's the last question: Suppose I query only on the first name to retrieve all the people named Jeremy? Will SQL Server use the FirstName column from the index and optimize as well as it did when I used the last name? **Figure 4** doesn't give us great news.

Unfortunately, SQL Server won't perform an INDEX SEEK. Although SQL Server uses the **LastNameFirstName** index, it performs an INDEX SCAN through all 19,992 rows. It only finds one "hit" and performs a key lookup to retrieve the non-key columns.

To achieve the same general optimization, you'd need to create a separate index where the leftmost column (or only column) is the first name. Bottom line: SQL Server reads the leftmost key columns, so keep that in mind when designing indexes, and query against multiple columns.

You can find many websites that talk about composite indexes. Here's a particularly good one: https://learn.microsoft.com/en-us/answers/questions/820442/sql-server-when-to-go-for-composite-non-cluster-in

Indexes and Fill Factors

Some developers who're very good at SQL queries hesitate to get involved with questions that get closer to what some might normally feel is "the job of a DBA." As an applications developer, I'll freely admit that I don't have full knowledge of a DBA. However, there are *some* topics that cross over into the application space, and even if you don't use them, it's still important to at least understand why some tasks are not performing well.

As a basic introduction, when you create an index, you can define the percentage of space on each leaf-level index page that SQL Server fills with data. Anything less than 100% represents the remainder that SQL Server reserves for future index growth. If I specify 90% (often a default), SQL Server leaves 10% empty for instances when additional data is added to the table. Setting a value too high (or too low) in conjunction with index fragmentation can lead to "opportunities for improvement."

And Now for Some Basic SQL Syntax

Many SQL interviews start with making sure the person knows the difference between an **INNER JOIN, OUTER JOIN, FULL JOIN**, etc. I'm going to go over some examples in a moment, even though there are many websites that cover this. Before I start, here's a little secret during SQL assessments: You can put a person's knowledge of JOIN types to the test by presenting them with a scenario along the lines of this:

"Suppose I have 100 customers in a customer master and 100,000 rows in a sales table. You can assume that every sale record is for a valid customer. In other words, there are no orphaned sales rows. If I do an INNER JOIN between the rows based on customer ID, how many rows should I expect to get? If I do a LEFT OUTER JOIN, how many rows should I expect to get?"

Yes, that's an interview question floating out there, I kid you not. The problem is, you don't know if the person is trying to see what other questions you might ask, or maybe the person is trying to see if the two numbers

(100 and 100,000) will be distractors. Just like spelling bee contestants will ask, "Can you use the word in a sentence," don't be afraid to ask questions about the nature of the tables/keys/row counts that might be relevant.

Want to be able to handle questions on different JOIN types? First, practice with a database. If you don't have one, take some of the data in the Microsoft SQL Server AdventureWorks database or the Contoso retail data. I'll use AdventureWorks for a few examples with INNER and OUTER JOIN.

Suppose I have a Vendor Table with 104 vendors. I have a sales Table with thousands of sales rows. I have sales for some vendors but not all. Assume the following:

- We have a Vendor table with 104 rows.
- Every PurchaseOrderHeader Vendor ID is a valid value as a Business Entity ID in the Vendor table.
- Each TotalDue row in the OrderHeader table is a positive value (i.e., no negative numbers).

What can I say about the number of rows I'll get back from this query, which will generate one row for each Vendor with sales, and a summary of the Order dollars?

```
select Vend.Name,
  SUM(TotalDue) as VendorTotal
    from  Purchasing.Vendor   as Vend
       join  Purchasing.PurchaseOrderHeader AS POH
          on Vend.BusinessEntityID = POH.VendorID
      group by Vend.Name
    order by VendorTotal desc
```

If someone asks you how many specific rows you'll get back, there's no way to answer that. Yes, you'll get back one row for each vendor that had at least one sale. However, because you don't know exactly how many vendors DIDN'T have sales, the row count could be as low as 0 vendors or as high as 104 vendors. You're doing an INNER JOIN (i.e., basic JOIN) that gives you vendors who have sales, will return one row for each vendor (because you specified a GROUP BY), and provides the sum of the Order Dollars.

Okay, let's change the query to this:

```
select Vend.Name,
  SUM(TotalDue) as VendorTotal
   from  Purchasing.Vendor   as Vend
   join  Purchasing.PurchaseOrderHeader AS POH
       on Vend.BusinessEntityID = POH.VendorID
         where (OrderDate
             BETWEEN '1-1-2012' and '12-31-2012')
    group by Vend.Name
order by VendorTotal desc
```

I'll ask the same question: What can you say about the number of rows you'll get back and the sum of the dollar amounts? All you can say is that it won't be MORE than the first query, because you've set a filter condition in the Where clause to only read the rows with an Order Date in 2012. If all the orders were in 2012 to begin with, I'd generally expect the numbers to be the same as the first query. If there were orders in other years, I'd expect either the count of vendors and/or the sum of the dollar amounts to be lower. Again, I can't predict the specific number of rows: All I can say is that it's somewhere between 0 and 104.

Okay, let's take the original query and turn it into a LEFT OUTER JOIN:

```
select Vend.Name,
  SUM(TotalDue) as VendorTotal
    from  Purchasing.Vendor   as Vend
         left outer join
         Purchasing.PurchaseOrderHeader AS
             POHon Vend.BusinessEntityID =
                 POH.VendorID
          group by Vend.Name
       order by VendorTotal desc
```

In **this** case, I can say with certainty (assuming no orphaned/invalid foreign keys) that SQL Server will return 104 rows. I can say that with confidence because I'm doing a LEFT OUTER JOIN. I'm telling SQL Server the following: Give me ALL the rows from the table to the left of the LEFT statement (the vendor table), and either sum the vendor dollars or just give me a NULL value if the vendor didn't have any sales. Given the conditions, I can say with high confidence that I'll get back 104 rows.

Now, I'm going to throw the proverbial "curveball." Suppose I want all the vendors (all 104), regardless of whether they've had an order. This time, I want the dollar amount to only reflect the sales in 2012. Will this give me 104 rows?

```
select Vend.Name,
  SUM(TotalDue) as VendorTotal
    from  Purchasing.Vendor   as Vend
     left outer join
        Purchasing.PurchaseOrderHeader AS POH
           on Vend.BusinessEntityID =
                POH.VendorID
        where (OrderDate BETWEEN
             '1-1-2012' and '12-31-2012'  )
          group by Vend.Name
       order by VendorTotal desc
```

> If your reply is, "Well, I didn't know that, but I'd never write it that way," take a moment to realize that you might inherit code where someone didn't know, but DID write it that way.

Some might say, "Sure, I'll get 104 rows. The dollars could be lower because we're filtering on a year, but we'll still get 104 rows." They might defend the answer of 104 rows, because I've specified a LEFT OUTER JOIN. As one of the Bob contractors said in Office Space, "Hold on a second there, professor." You'll only get the vendors who have an order. (When I run it, I specifically get 79, but the purpose of this discussion is to demonstrate that I'm no longer guaranteed to get all of them).

Here's why. That WHERE clause works a little *too* well. Even with the LEFT OUTER JOIN, the WHERE clause restricts the number of rows. I've known developers who were surprised to learn this, and I've seen code in production that fell victim to this.

How can you get all vendors (in this case, 104), AND only get the sales dollars for 2012 (or a NULL). You need to remove the WHERE from the main query. There are several ways. Some will localize the condition in the LEFT OUTER JOIN, or (my preference) use a subquery. Here are both solutions:

```
select Vend.Name,
 SUM(TotalDue) as VendorTotal
  from  Purchasing.Vendor   as Vend
    left outer join
         Purchasing.PurchaseOrderHeader AS POH
           on Vend.BusinessEntityID = POH.VendorID
             and (OrderDate BETWEEN
                 '1-1-2012' and '12-31-2012'  )
    group by Vend.Name
  order by VendorTotal desc
```

```
select Vend.Name,
  SUM(TotalDue) as VendorTotal
   from  Purchasing.Vendor   as Vend
    left outer join
     (select VendorID, TotalDue from
        Purchasing.PurchaseOrderHeader
          where (OrderDate BETWEEN
             '1-1-2012' and '12-31-2012' ) ) POH
      on Vend.BusinessEntityID = POH.VendorID
 group by Vend.Name
order by VendorTotal desc
```

In Part 2 of this series, I'll talk more about **CROSS JOIN** and **SELF-JOIN** scenarios. Basically, you use CROSS JOIN when you want to create a Cartesian product of the table rows. For instance, say you have 45 employees in a sales table and a calendar table with 52 weeks, and you want to create 2,340 rows, one for each employee/week combination. You use a SELF-JOIN when you need to query a table back into itself (i.e., you query it as Table A, then join to the same table as Table B, usually because the single table has a relationship across columns). Again, I'll cover them more in Part 2, because I want to review the topic of recursion and common table expressions.

Do you know what a UNION does and know the difference between UNION and UNION ALL? Here's how I answer that question. Suppose you have three different tables (or three query results), with the same number of columns. You want to append all of them into one table. If you know for **absolute certain** that the three tables don't contain duplicated rows (across all the columns), or you don't care if you get duplicate rows, a **UNION ALL** will give all the rows from all three tables. If Table/Result A had 100 rows, B had 200 rows, and C had 300 rows, a **UNION ALL** guarantees a result of 600 rows.

If there's any risk of duplicated rows, you should perform a **UNION**. A **UNION** will perform a duplicate check based on all the columns (under the hood, SQL Server performs a SORT DISTINCT). This can be expensive for large sets of data with many columns but might be necessary. If someone presents the scenario of three tables with the row counts I described, and asked what a UNION (not a UNION ALL) will generate for a final result count, the answer depends on whether any of the rows are duplicated across the tables.

Do you know the difference between **HAVING** and **WHERE?** I know good developers who stumble on this question. Yes, a good developer will surely test out the difference and do the necessary research when writing a query that calls for a HAVING and figure it out through some trial and error. Although it might seem a bit unfair to expect a person to verbalize the answer to every question, the ability to express an answer is sometimes rooted in the amount of actual experience. (This is such an important distinction that I'll cover it a second time in the advanced SQL Server section.)

That's certainly not a full list of basic SQL Server topics, but these are items and nuances that I'd expect database developers to know. Before I move on to some advanced SQL Server topics, I want to stop and bring up a point: *MOST* technical interviews are performed by people who can listen to an answer and judge appropriately. If you can establish a conversational dynamic with an interviewer, you can usually tell if the other person is adept at assessing your responses. I say "*most*, because there's something that still occurs in this industry, and it personally infuriates me, and that's when a "tech screening" is being done by a person who's given a script. It happens. Sadly, there's no hotline you can call to report such intellectual laziness (and believe me, I can use stronger words). You can try to call it out in the interview process, but obviously you run the risk of ruining your chances. My only advice is to just answer the question as best you can.

Some Advanced SQL Server

Ask ten different SQL developers what constitutes "basic SQL" versus "advanced SQL," and you'll get many different answers. Here's my two cents and it's probably controversial: Anything that a standard ORM or Entity Framework generates as SQL code for an OLTP application is **usually** basic SQL. Anything that a data framework can't generate (or doesn't do well, so much so, in fact, that it becomes a good training example of how NOT to write something) is advanced. Yes, that's my story and I'm sticking to it.

Anything dealing with JOIN statements on matching keys, WHERE clauses to filter, and basic aggregation (i.e., summing the sales dollars across millions of sales rows and aggregating to one row per Product Brand), all of that falls under basic SQL querying.

Most of the other language features are "non-basic." Okay, I'll acknowledge that the difference between HAVING and WHERE takes less time than explaining the nuances (and shortcomings) of PIVOT because the former is generally an "either/or" and the latter has several parts to it. Still, the next ones are some advanced topics.

Know the difference between HAVING and WHERE. Here's the bottom line: You use HAVING to filter on aggregated amounts, and you use WHERE when you're directly referring to columns in a table. Suppose you're retrieving the customer ID and the Sum of Sales Dollars and grouping by the customer. In the query, you only want Customers with more than one million dollars in sales. You can use HAVING Sum (SalesDollars) > 1000000 after the GROUP BY statement.

Alternatively, you can sum the sales dollars by customer to a temporary result set or derived table. When you query from the temporary result set or derived table, you can use WHERE, because the resulting rows are already aggregated down to specific columns.

Too Much Theory and Not Enough Fresh Air

When I was a young developer, a project manager fought against me being able to get my hands on anything other than VERY small test rows. He also argued against me sitting in meetings with the business people. He was convinced that he and his colleague could write the perfect specification, and that the team should be able to write pure and optimized code without ever having a chance to test it against large or actual data. He even scoffed when I randomly generated a large set of test rows. Reading and studying is important, but it's when you get your hands dirty (and crushed by the weight of hard lessons) that you become a data professional.

Know how the RANKing functions work. These particularly come in handy instead of trying to use TOP (N). For instance, if you want to get the top five selling products by state, the RANK functions are preferable to TOP (N). Also remember that there are three flavors of the general RANK functions: **ROW_NUMBER(), RANK()**, and **DENSE_RANK().** The first doesn't account for ties, the second accounts for ties and leave gaps, and the third accounts for ties and close gaps. Recently I caught an interviewer trying to trick me into a TOP N response when the query in question called more for a RANK function.

```
Select * from (
  select  PurchaseOrderID, EmployeeID, VendorID,
          ShipMethodID, OrderDate, TotalDue,
          rank() over (partition by shipmethodid
     order by TotalDue desc) as RankNum
  from Purchasing.PurchaseOrderHeader  ) t
where RankNum <= 3
order by ShipMethodID, ranknum
```

Understand how CROSS APPLY and OUTER APPLY work. These have been somewhat controversial topics in the SQL Server World. Introduced in SQL Server 2005, they're great for patterns where you need to perform more gymnastics than a regular JOIN (such as accumulating, reading across—or dare I use a COBOL term—"Perform Varying"). In my last article (CODE Magazine, January/February 2024), I wrote that CROSS APPLY can be used to get cumulative results. (Additionally, Microsoft added optional clauses in SUM OVER for ROWS BOUND PRECEDING in SQL Server 2012). There can be a performance hit with any of these features, as they potentially need to process and aggregate many rows over many iterations. I typically use these for overnight reporting jobs or any instance where users accept that the results won't be instant.

Over the years, I've discovered that CROSS APPLY has some other interesting uses, such as using a CROSS APPLY as an alternative to UNPIVOT:

```
create table TestTable
   ( SaleDate date, Division1Sales money,
     Division2Sales money,
     Division3Sales money)

 insert into testtable values
 ('1/1/2023',100,200,300),
 ('1/2/2023',200,300,400),
 ('1/3/2023',200,300,0)

SELECT SaleDate, Division, Sales
FROM testtable
 CROSS APPLY (
    VALUES ('Division1', Division1Sales),
           ('Division2', Division2Sales) ,
           ('Division3', Division3Sales))
        Temp (Division, Sales)
WHERE isnull( Sales,0) <> 0
```

Know when subqueries are required. I talked about this in my last CODE Magazine article (January/February 2024), using the seemingly simple example of multiple aggregations and when a subquery is needed. Ultimately, almost every developer runs into this situation of aggregating across multiple one-to-many relationships, where the two (or more) child tables have no relation to each other beyond their common relationship with the parent.

Understand Triggers and Capturing changes. I've written about this in prior CODE articles (most recently, January/February 2024), so I don't want to repeat details. Most environments need (or require) solutions to track changes to data. Whether it's writing automated scripts for triggers (or just hand-coding triggers), or using Change Data Capture, or relying on some third-party tool, it's critical to know how to log changes and how to report on them.

How do you optimize queries? You have a query that suddenly runs slowly and you need to figure out why. Here's where there's seldom a single right answer. There are several diagnostic philosophies here. One of the things I'll do is to hit the low-hanging fruit first. Have there been new application deployments, did the volume of data spike astronomically overnight, can you run SQL Profiler or use the SQL Query store to get a bigger picture, etc. The list of the "low-hanging fruit" can be very long and five different people can contribute 10 uniquely valid ideas, but the point is to identify whether some significant event triggered the drop in performance.

Second, I'll review the query, the data, and the execution plan. Again, I might be looking for low-hanging fruit, but at least it's targeted toward the specific query. Over the years, I've become more cognizant that a solution using views (and views calling views) are not unlike blowing air into a balloon: At some point, you provide one more shot of air and the balloon pops. My point is that adding "one more table" or "one more join" to a view five distinct times doesn't mean that the progressive impact will be consistent for all five. This is something I want to devote to a future article, but for right now, keep an eye on solutions that have nested views. They sometimes can lead to "too much air got pushed into the balloon." In Part 2 of this article, I'm going to talk more about this, and how certain execution operators in the execution plan can give you a clue that you have an issue.

If the problem is indeed "views gone wild," what's the solution? Well, "it depends." I'm not a big fan of views calling views calling views to begin with, and so I'll try to delve into the workflow of the desired query. If it's something that only needs to be produced one or two times a day, I'll wonder if a materialized snapshot table with a crafted stored procedure will do the trick. Alternatively, maybe the culprit is a stored procedure that has five queries in it, and maybe the third of the five is the one taking all the time. Ultimately, those who can deal with bad performance are the ones who have gone through it many times. Quite simply, this isn't an easy topic.

Isolation Levels, Snapshot Isolation Levels, and Read Uncommitted ("Dirty Reads")

No one should judge a developer negatively for not knowing about a specific feature. I've known good SQL Server developers who were unaware of a feature, or perhaps got it backwards. Having said that, I'm going to go out on a limb and say that if there's one area of knowledge that can help you stand out as someone with good skills, it's **Isolation Levels**. Odds are (and I emphasize, "odds are") that someone who can speak to Isolation Levels, read/write

> It's 2024 and there's no excuse for not capturing an audit trail of database changes. You have Temporal tables (SQL Server 2016), Change Data capture (SQL Server 2008), and you still have triggers (from the early days)! Pick one, study it, and implement it.

locks, and the SQL Server snapshot isolation level is likely to show a good command of other SQL Server fundamentals. I'd almost call this an intellectual "gateway" feature.

In SQL Server, there are five isolation levels: Read Uncommitted (dirty read), READ COMMMITTED (the default isolation level), Repeatable Read, Serializable, and the Snapshot Isolation Level (added in SQL Server 2005 and it comes in two flavors). I'm going to cover the first two (read uncommitted and READ COMMMITTED) and the last one (snapshots) in this article, and I'll cover Repeatable Read and Serializable in the next article.

Any time I've ever been asked to talk about isolation levels, I explain them in terms of "Task A" and "Task B." Both tasks could be users running an option in an application, or one of them could be automated jobs, etc. If anyone ever wants to test your knowledge of isolation levels, I highly recommend talking in terms of a "Task A" and "Task B" scenario.

I'll start with the default, which is READ COMMMITTED. Let's say Task A initiates a transaction that updates thousands of rows into a table. While that transaction is underway, Task B tries to read some of those rows. Because the default isolation level is "READ COMMMITTED" in SQL Server, that means Task B will be locked from reading updates from Task A (or could possibly timeout) until Task A finishes. The specific reason is because Task A's update has a write lock on the rows, and that prevents Task B from attaining a shared lock to read the committed version of the rows. Task B needs to wait for Task A to finish and commit the transaction (or for Task A to rollback because of some error in Task A).

There can be problems here, and discussions are not without controversy. Task B might simply need to get an "approximation" of data. In other words, it doesn't matter much if Task A commits or rolls back; it's small in the grand scheme of what Task B wants to query. What Task B can do is set the isolation level to a read uncommitted (dirty read). The benefit is that Task B won't run a risk of timeout waiting for Task A to finish, and a dirty read is sometimes a bit faster than a normal READ COMMMITTED, because SQL Server isn't even attempting to put a shared lock on the rows.

Sounds great? Well, dirty reads bring some real risks. and here are three of them.

First, suppose task A's transaction is updating both a header table and two child detail tables. Task B performs a dirty read between the update of the header table and the child tables. Task B sees the new header row but doesn't yet see the related detail rows. In other words, Task B returns an incomplete picture of what Task A's transaction intended to write. That's not good!

Second, suppose Task A starts and updates one or more of the tables, but performs a rollback at the end (because of some post-validation error, etc.). Suppose Task B read the data using READ UNCOMMITTED just before the rollback. Task B will be returning data that never officially saw the light of day because Task A rolled it back. That's also not good!

These two alone should provide caution to developers who use READ UNCOMMITTED. Again, there can be specific instances (often tied to workflow throughout the course of the day) when dirty reads can help, provided those instances are well-managed. There's also a third risk. This one is covered less often in web articles because it's not very common, but it's still a risk. Suppose Task A's transaction is performing inserts such that a page split occurs in the SQL Server table. If Task B is using a dirty read, it's conceivable that Task B could pull back duplicate rows that came from the page splits.

Okay, so a READ COMMMITTED means that users might get timeout errors (or wait longer for the results of queries), and a dirty read means that avoiding those timeout situations might lead you to reading "bad/dirty" data. This was a challenge for a long time. Fortunately, Microsoft SQL Server 2005 addressed this with one of the more important features in the history of the database engine: the SNAPSHOT isolation level. (Notice how I've skipped past REPEATABLE READ and SERIALIZABLE. I'll cover them in Part 2 of this series).

Now I've gotten to the **SNAPSHOT** Isolation Level. I'm biased, but this is a fantastic feature. This comes in two flavors, the standard SNAPSHOT (that I call a static SNAPSHOT) and a READ COMMMITTED SNAPSHOT (that I refer to as a "dynamic SNAPSHOT). This gives us the good benefits of dirty read and READ COMMMITTED but without the downsides.

First, to use the standard "static" snapshot, you need to enable it in the database, because SQL Server is going to use the TempDB database more heavily:

```
-- Code to alter a database to suppport
-- static snapshots
ALTER DATABASE AdventureWorks2014
   SET ALLOW_SNAPSHOT_ISOLATION ON
```

Here's the scenario for a static snapshot:

- Customer XYZ has a credit rating of "Fair".
- Task A starts a transaction and updates the customer's credit rating to "Good". This transaction could take several seconds or even minutes to complete if it's a long and intensive job
- You've seen that Task B will have to wait (or might even timeout) if you use the default READ COMMMITTED. You've also seen that Task B returns a value of "Good" if it uses a READ UNCOMMITTED, which isn't "good" if Task A rolls back.
- Suppose that Task B starts its own read session/read transaction and simply wants to get the last committed version ("Fair"). Task B can initiate a transaction, set the ISOLATION LEVEL to "SNAPSHOT", and then query the table: Task B's query returns "Fair". It won't give you a dirty read and it won't give you a timeout.

> Many DBAs refer to snapshot isolation as versioning.

Sounds great, doesn't it? Overall, it is, although there's one downfall. Suppose Task A commits its transaction, but Task B is still in the middle of its READ session of SNAPSHOT. If Task B queries the row again (in the same read session), it continues to return "Fair" **because it**

continues to read from the last committed version AT THE TIME its own READ session started. (This is why many DBAs refer to snapshot isolation as versioning).

Here's what Microsoft did to handle that. In my opinion, this ranks as one of the finest achievements in the history of the SQL Server Database engine. They added an optional clause to the SNAPSHOT isolation level. By performing the following setting, you are guaranteed that EVERY default READ COMMMITTED will give you the most recently committed version.

```
-- Code to alter a database to suppport
-- READ COMMMITTED SNAPSHOT
-- This will configure SQL Server to read
--  verions from TempDB

ALTER DATABASE AdventureWorks2014
   set read_committed_snapshot on
```

Go back to the previous example. If Task B reads the row a second time, it picks up the change from Task A. Again, it's because of the dynamic nature of READ COMMMITTED SNAPSHOT.

You don't even need to start read sessions with a SNAPSHOT isolation. By turning on READ_COMMITTED_SNAPSHOT in the database, **every single READ COMMMITTED (the default level)** automatically pulls the most recently committed version of the row right at that time.

Does that mean that the READ COMMMITTED SNAPSHOT will work wonders right away? It might, but there's one catch. As SQL Server is using **TempDB** as a version store, DBAs will likely need to step in to deal with the management of **TempDB**.

Now for Something a Little Bit Different

Here are two more SQL Server tricks and approaches that experienced database developers have up their sleeves.

First, suppose you have a report that shows one row per product with the sum of sales. The company sells the product in different sizes, but the report only shows the summary of sizes.

Now suppose the business has an urgent request. At any one time, maybe two or three of the sizes need to be replenished (which you'd pull from a query). You want to show the list of sizes, except the report row granularity is fixed and you can't add additional row definitions to the report. Also, because there might be 20 sizes, you can't add 20 columns out to the right of the rest of the data, with a checkbox for the sizes you need to replenish because that would take up too much space.

But wait! There's something you can do. After talking to the users, they just need to know the X number of sizes, nothing else. It's almost like seeing a tooltip on a webpage, except this is a generated report. Maybe all you need to do is just add one more column to the report, called "Sizes to Replenish", and show a comma-separated list of the sizes?

If you've ever read the famous "Design Patterns" book, you'll know that the authors (affectionately known as "The Gang Of Four", Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson) came up with some great "short but sweet" names for patterns. Well, I'm not talented enough to come up with "short but sweet terms," but I have a name for this pattern. I call it the "stuff the X number of variable business values into a single comma-separated list, and then jam that as a single new column in the report. (Side note: I'd never succeed in product branding.)

I've had business users who were happy with this result, and IT managers who breathed a sigh of relief that there was a simple solution that wouldn't disturb the expected row count.

Okay, so how can you do this? Prior to SQL Server 2019, a common approach was to use the "FOR XML" statement. In SQL Server 2019, Microsoft added a new **STRING_AGG** function to read over a set of rows and "aggregate" values into a comma-separated string. Take a look at STRING_AGG, which works very nicely. Also, look at **STRING_SPLIT,** a long overdue feature to easily do the reverse: convert a comma-separates string of value into a result set.

Second, this is one that initially seems like a simple job for MAX and GROUP BY but winds up being a bit more involved. Because I've established myself as a long-winded branding author, I'd like to refer to this pattern as the "I thought I only needed a GROUP BY and MAX, but now I need to go back and pull one other column that's not directly part of the aggregation." (I'd fail marketing.)

Suppose you have a query that initially shows the single highest transaction by customer. It could be something as simple as this:

```
SELECT CustomerID, Max(SaleAmount)
    From Sales
  GROUP BY CustomerID
```

But now you're asked to show the Sales Person associated with that sale. You can try all day long, but you won't be able to write this as a single query. You'll need a subquery of some type.

Additionally, you could have a tie within a customer. Maybe the customer bought something from Sales Person A for $100 and something else from Sales Person B for $100. There are two sales transactions for the same amount with a different Sales Person.

Here's another situation when spotting this pattern (and writing a correct query) can sometimes indicate a developer's level of experience.

```
;with tempcte as
   ( SELECT CustomerID,
    max(SaleAmount) as MaxSale
       from Sales Group by CustomerID)

Select tempcte.*,  OriginalTable.SalesPersonID,
OriginalTable.SaleID, OriginalTable.SaleDate
From Tempcte
  Join   dbo.Sales OriginalTable
     on Tempcte.ID = OriginalTable.ID and
     tempcte.MaxSale = OriginalTable.Saleamount
```

There are multiple ways to write this. Some might use the Common Table Expression as an in-line subquery, and some might use CROSS APPLY, etc. This point is that when you aggregate across many rows to come up with a SUM or a MAX or whatever, you might need to **bring along** some additional columns for the ride. (Aha! Maybe I can call this the "Tag-along" aggregation pattern!)

Finally, I want to bring up a pattern that I've only had to deal with twice (to my recollection). It was a bit unusual, and very humbling. I had to write a query that summarized process duration, and mistakenly thought that a few subqueries and aggregations would do the trick. It turned into a hair-pulling afternoon, although I benefitted from it. I came up with a working solution, and then learned later that another SQL Server author (Itzik Ben-Gan) had documented the pattern, called "gaps and islands." Because I already presented the code in a prior CODE Magazine article (January/February 2018), I'm not going to re-paste the code here. If you want to see a more advanced example of where a developer can mistakenly oversimplify a problem, I documented my entire thought process in the January/February 2018 issue of CODE Magazine ("A SQL Programming Puzzle: You never stop learning").

On the Subject of Starting Out

There's no single roadmap in learning a discipline (in this case, being a database developer). As people often say, "It's a journey." Just make sure "journey" is a verb as well as a noun.

Data Profiling: I've Seen Fire (Audits) and I've Seen Rain (More Audits)

I'll admit, I have a bit of a wise-guy side to me, and it nearly came out in a recent conversation. Someone asked if I'd done data profiling and the proverbial little devil on my left shoulder wanted to say, "Well, I don't know that specific term, but in my last project, I had to take a large number of rows from a bad OLTP system and search for patterns."

Obviously, I didn't make that joke, but here's what I DID say: You can't spell "accountability" without "count." To me, accountability isn't about "who can we blame if something goes wrong?" Accountability means "what can we count on" or, in the world of databases, "all rows are present and accounted for in some manner."

For example, I worked on a project where we had some pretty significant cost discrepancies between two systems. We knew the issues stemmed from code between the two systems that needed to be refactored. Before we could dive into that, we had to come up with a plan RIGHT AWAY to fix the data. Of course, you can't fix a problem (or in this case, a myriad of problems) without identifying all the issues, and that was the first order of business: identifying all the different ways data had gone bad.

Without going into specifics, we found four different scenarios. Of those four, two of them had sub-scenarios. Some of these were simple and "low-hanging fruit" and some were more complicated. Here were some of them:

- Rows in the legacy system marked as deleted/archived, but still in the production system
- Rows in the legacy system marked as deleted, but should not have been
- Rows in the legacy system with multiple cost components, where the target system only ever recognized the first component
- Rows in the legacy system with cost values that the target system could not process (dirty data that had never been validated)
- Multiple rows in the target system that were marked as active, when only one row should have been marked as active, based on the granularity

Before I go on, I want to talk about that last word: granularity. It might sound pedantic to invoke that term during a meeting about fixing data, but as it turns out, three different functional areas in the company had different opinions about what constituted "granularity of an active row." The problem is that some business users aren't going to think in the most precise terms about this (nor should they be expected to). They'll simply look at three candidate rows on a screen, point to the second, and say, "THAT ONE! That's the active row, and the others shouldn't be, because of this and this and this."

And that, folks, is how you come one step closer to defining (in this case) what constitutes the definition of an active row.

Okay, back to the original story, one of the first things we provided to management was a summary of what percentage of bad rows fell into each category. We had roughly 6,000 rows identified as "bad" with 20% falling into Category A, 50% into Category B (and then a sub-breakdown of Category B), 10% into Category C, etc.

The queries were a series of COUNT and EXISTS

```
DECLARE @CategoryACount int , @CategoryBCount int

SET @CategoryACount = (select count(*) from
 CostDetailsLegacy outside
  where MarkedArchive = true
    And exists
      (select 1 from CostProduction inside
        where outside.<Key1> = inside.<Key1> and
              outside.<Key2> = inside.<Key2> and
       Inside.ConditionForActive = true


SET @CategoryBCount = (select count(*) from
 CostDetailsLegacy outside
  where MarkedArchive = true
   And exists  (select 1 from CostProduction inside
        where outside.<Key1> = inside.<Key1> And
             outside.<Key2> = inside.<Key2> and
             inside.ConditionForActive = true
```

Additionally, management might provide an error factor: Maybe if cost rates differ by less than two cents (rounding errors, bad math approaches involving integer division, etc.), and *maybe* they'll elect to tackle that later. Once I saw a manager flip out when they saw the top of the list of discrepancies sorted by variance and the overall row count. They thought that because the top 10 rows were off by a large percentage and we had thousands of rows, that we had a disaster. It turns out that after row 20, the variances dropped to pennies, with a slew of numbers off by just a small amount. So even within your categories, check the deviation among the rows.

I'm not going to devote two pages of code to the specifics (it'll just give me flashbacks and nightmares anyway).

Hopefully you get the idea: You need to identify all the anomaly conditions before you can fix them. Additionally, management might decide to go over a particular scenario first, because it's the most egregious, or easiest to fix, etc.

Obviously, this won't fix any data, but it gives you a good structured starting point for a game plan, as opposed to playing the proverbial whack-a-mole when you see bad data. This is what I mean by data accountability: what rows you can "count on" as being valid, which ones you can't, and why. Trust me, the auditors are trained to think in these terms (and others).

Always Know the Product/Language Shortcomings

I was on an interview team where I asked a question that some liked and some wondered what planet I came from. It was for an ETL position where SSIS and T-SQL would be used heavily, and so I asked, "What three things about SSIS annoy you the most that you wish worked differently?"

I wasn't so much interested in the specific answers as much as I was interested in their though process and product experiences. At the risk of making an absolute statement, it's unlikely that someone has a strong amount of prior experience with SSIS and has been a happy camper about the product **all the time.** I love SSIS and I've debated with developers who refused to use it, but that doesn't mean I haven't (out of frustration) come up with alternate definitions for what the letters SSIS stand for! The SSIS Metadata manager in the data flow has gone from frustrating to barely acceptable over the years. SSIS's handling of package-level variables at the script level makes me think of two people talking with plastic cups and a string. I have other pet peeves. Again, I love SSIS, but the product has weak spots, and I'm curious to know what a developer thinks is a weak spot because it usually means they've cut their teeth with it.

When Microsoft first implemented the PIVOT in T-SQL 2005, I was mildly excited. I had done many queries involving aging brackets, and looked forward to converting some code I was writing to use PIVOT. The story's ending wasn't quite as happy as I expected. It streamlines some things, but also introduces other "gotchas" to be aware of (dynamic lists, multiple columns, and the need to PIVOT on one pre-filtered/pre-defined result set, as opposed to mixing PIVOT with other T-SQL constructs and with an irregular result sets). I was once asked in an interview what new language feature gave me the most frustration. I chose PIVOT: not because I didn't understand it, but because I'd had enough practical experience that I *sometimes* wondered if the pre-SQL Server 2005 approach was better. I'm reasonably confident that the interviewer would have accepted different answers, so long as I said something that demonstrated I'd gotten my hands dirty with it.

Normalization, Denormalization, and Warehousing Basics

There's a classic argument that goes along the lines of this: Person A will ask, "What's a short definition for X and Y, and what are the differences between X and Y?" Person B might very well have worked on projects involving X and Y, but might stumble on a definition, and finally respond, "Well, I can't give a textbook definition, but I've worked on both and know they're different, and I'm sure I can easily look them up on Google." Person A **might** accept that when push comes to shove, Person B "probably" understands enough.

When developers face questions about the difference between normalization and denormalization and try to describe some common terms in data warehousing, they sometimes can fall a little short, even if they've successfully worked on projects. Yes, I realize that memorizing standard answers might not be the way to go, but....

Let's take normalization and denormalization first. The goal of normalized databases is to reduce/eliminate redundancy, to ensure data integrity, and to **optimize getting data into the database as efficiently as possible**. You might have to write queries to join data across many tables, but a SQL Server developer shouldn't fear that prospect. Typically, application developers are the only individuals who'll be accessing normalized tables with SQL code.

Denormalized databases are a different breed: Power users and "non-developers" might be more likely to read them because data can be redundant in denormalized tables. One of the major goals of denormalized databases is to optimize getting data OUT of the database as quickly as possible. Fewer complex JOIN statements are needed, which can appeal to power users who know how to use reporting tools but want to query the data with minimal effort.

Finally, something you can use to impress your friends at the dinner table: **Some data warehouse initiatives (especially ones with a large number of disparate data sources) shape data into normalized structures FIRST as part of an overall ETL effort, before turning around and flattening them into denormalized structures.** For developers who've only ever worked in one type of database, this concept might initially seem like overkill, but it's an invaluable approach for making sure data in the data warehouse is clean.

Yes, the capabilities of certain BI and reporting tools, and even advancements in database features, have created some grey areas, but these two concepts still breathe on their own today, and they're not going away.

The topic of data warehouses is a bit of a different story. There are developers who can cleanly describe the differences between normalized and denormalized databases but might admit to a shortfall of knowledge on data warehousing concepts.

If you plan to work in the data warehousing area, there's one book I absolutely recommend you read: "The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling" by Ralph Kimball and Margy Ross. There are multiple editions of this book. Even if you can only get access to the first edition, it's still worth reading. I realize that it almost sounds cultish to recommend a book so strongly, but the book actively engages in different busi-

ness cases where Kimball and Ross use repeatable patterns to handle different scenarios. It's not just a book to quote, it's also a book with approaches to live by.

> Next time someone says, "Why should I study the Kimball methodology? Isn't that antiquated?" here is a proper reply. "Do you think that all databases built on Kimball fundamentals have suddenly disappeared?" Knowing how to implement Kimball ideas will be relevant for a LONG time. It's not a technology, it's not a language: It's a collection of approaches, and an outstanding one that that.

There are many different data warehousing concepts, but here are a few you should know:

First, what are Fact Tables and Dimension Tables? Loosely speaking, Fact Tables represent activity/business processes. They could be sales, returns, transactions, manufacturing production and quality, or even survey evaluations: SOMETHING happened, and you want to analyze those somethings over time. They could occur five million times a day or just 100 times a week. Last night, I bought some new art supplies for my daughter—that transaction might very well be sitting in a fact table. More on that in a minute.

Dimension tables represent the business entities that are associated with the activity mentioned above. I bought the supplies at a certain time of the evening from a store in a particular sales district that belongs to a certain sales region. The products were specific art supplies made by a certain company and came in certain packaging.

Here's the single most important word in data warehousing: the word "by." Sales departments want to see the rollup of sales dollars "by" city, "by" customer type, "by" date/month or even "by" time of day, "by" product manufacturer and product type. So those core identifiers that were part of the sale (the exact date/time, the customer ID, the product UPC, the store ID and maybe even sales register number) belong to larger groups.

If the product was a seasonal item, sales might want to track sales across years by season (seasonality), where the actual dates might vary a bit. (I like to use the classic example of tracking sales of fish foods during Lent each year, where Lent can be different weeks each year).

Now, you can look up fact tables and dimension tables and find several equally valid definitions. This is one of the reasons I detest unqualified people doing interviews with an answer sheet who are unable to determine if a particular response aligns with a predefined answer.

Okay, let's go back to fact tables. The scenario I described (a sales record) is the most common type of fact table: a transactional fact table. I can aggregate all the sales dollars (or aggregate by dimensional groupings) and that measure will mean something. You refer to the measures in a transactional fact table as fully additive.

Certain businesses want to take a picture of the sum of transactions over some period. Imagine a bank taking a picture of all a person's charges and all their deposits during the month and coming up with a balance. A company might, as part of a month ending process, read across all the bank transactions and write out a periodic snapshot table that holds the account ID, month ID, sum of charges, some of deposits, and the balance. Let's say they do that for three months and so the snapshot fact table holds three rows. Can you sum the numbers for the charges and deposits into something meaningful? In this case, yes. Can you sum the balance for each of the three months? Well, much as you might like that, the balance isn't cumulative. Unlike the first two measures, the balance isn't fully additive. Yes, you could take an average of it, which means that measure is semi-additive, and not cumulative. **Periodic snapshot fact tables are not uncommon.**

There's also a third type of fact table, usually involving an activity where you're not looking to sum up numbers (like sales or number of items produced) but merely counting the number of times something happened. For instance, let's say I teach two classes a day. Yesterday, 28 out of 30 attended Class A, and 18 out of 19 attended Class B. The next day, the numbers are 29 out of 30, and 17 out of 19, respectively. You might have a **Factless** fact table that records nothing more than the number of times something happened (attendance for a particular class by a particular instructor). All you really get out of the **Factless** fact table is the count, although that count over time (or average) might have analytic value.

Okay, I'm going to bring up one more topic that's likely to come up during an interview: slowly-changing dimensions. Here's how I describe it: suppose a product was introduced in 2022 and sold in all of 2022 for a manufacturer price of $50. On 1/1/2023, the price increased to $52. Then on 7/15/2023, the price increased to $53.

Here's the big question: Do you care about tracking sales during the time when the product was $50, versus $52, versus $53?

Here's another example. From the time I started paying taxes (i.e., age 18, first job), I've lived at 10 different addresses (kind of scary to think about it). Let's say some government agency tracks information about me. Do they care if was a resident in one area for X years/months and another area for a different amount of time?

When you care about aggregating fact data not only by an attribute (a product ID or a Social Security Number), but by changes to that attribute that have occurred over time, you refer to that as a Type 2 Slowly Changing Dimension.

Now you're wondering, "Okay, what's a Type 1 Slowly Changing Dimension?" It's simply when some attribute for an entity changes many times and there's no analytic need to aggregate fact data over time by each change.

There are challenges associated with Type 2 Slowly Changing Dimensions: late-arriving fact data and late-arriving dimensional changes. On the former, suppose you receive sales data today (late April 2024), but it represents transactions that occurred in January 2024. When you write out the fact data, you need to make sure the fact row points to the proper VERSION of that dimension at that time. On the latter (and this can be trickier): suppose you receive information today that a product's price changed back on 11/23/2023? You might be tempted to say, "Okay, we'll go back and update the dimensional pointers for those fact sales transactions that occurred after 11/23/2023 to point to the correct version of the product's change." Well, that depends on the database policy regarding "updating" fact tables. In some environments, fact rows are INSERT ONLY—you CANNOT update a fact row. In this instance, you'd fix the X number of fact rows that would have been associated with the new version of the product change (had you received it in a timely manner), INSERT new rows with reversing sales amounts, and then INSERT the rows the way you would have in the first place. Yes, this happens!

And Speaking of Snapshots…

At some point, many businesses need to see what data looked like based on a point in time. This can take a variety of forms. Suppose you report on daily production every morning at 5AM. You might want to see what a product looked like between January 10 and January 11 (possibly because of some significant event). Here's another one: maybe you need to see what a customer profile looked like back on February 3. Finally, suppose you want to show the trend of inventory on the first day of the month, the last day of the month, and any monthly averages, over the course of a year.

If it sounds like I'm partly repeating myself (that is, I'm mentioning some of the scenarios from the data warehousing section), you're correct. Snapshot tables, audit trail tables, and slowly changing dimensions, they all have some overlap. Even if you don't work in a full-fledged data warehousing team, your ability to solve some of these challenges can increase your value. Some people have this impression of data warehouses as a bunch of old backup tapes from data 30 years ago that's kept offsite in some dusty storage area. I thought that way at the beginning of my career, and I could not have been more wrong.

Database professionals have different opinions about the structure of such tables. On the one extreme, some might simply use change data capture to log every single change, and then use queries to reconstruct history at any one point in time. Granted, that could be a lot of SQL code, but it can be done. Some create temporal tables (or use the temporal table feature added in SQL Server 2016) to possibly reduce the amount of code needed to reconstruct history. Some create periodic snapshot tables where the periods represent specific established business timelines (beginning of month, end of month, average during the month, etc.). On one occasion, a client was stunned that I was able to reconstruct a timeline to show some very irregular production history, believing that I'd somehow concocted a magic formula to suddenly show what happened on a day three months ago. It wasn't magic, it was basic data warehousing concepts. At risk of repetition, and at risk of our famed editor striking this sentence because it's overkill (smile), I can't stress enough the value of reading the Ralph Kimball methodologies in his Data Warehousing books.

Ever see those TV commercials where two people are debating about whether something happened the day before? Suddenly someone shows up with instant replay equipment to show what happened. You want to be the person who can show up with the necessary replay equipment.

Other Miscellaneous Things

Think globally: Your applications aren't the only ones that might touch the data you manage. When you're looking at audit trail requirements, you can't assume that your processes are the ONLY ONES that will touch data. Once I got into a spirited argument with a SQL Server author that I highly respected. In the argument, the author stated that the new **OUTPUT INTO** clause that Microsoft added to SQL Server 2005 meant that triggers and other audit trail features would go away. He defended his position by saying he'd never work for a company that didn't pipe ALL their insertions through stored procedures. I told him that it just wasn't practical to think that way, that other jobs might access and modify data.

> In the database world, there are a few RED rules (rules you always follow) and lots of BLUE rules (rules you try to follow but might need to bend). At the end of the day, the pragmatists usually prevail—so long as they also have a plan for tomorrow and the next day.

Here's another topic: using PowerShell. I'll freely admit, I don't jump with joy when I use PowerShell. However, it can be an invaluable tool for "piecing together" different processes, especially database processes.

Suppose you need to run a standard ETL data job, execute some custom API out in the operating system, and then run a final database job? You'd like to use SQL Server Agent, as you could easily add Steps 1 and 3 as job steps, and then place the PowerShell script in between the steps.

Here are some ways I've used PowerShell:

- To deploy reports to an SSRS or Power BI report server area under a different service account
- When I had a specific custom process where I needed to tap into ADO.NET in between job steps, and SSIS and .NET modules were not a deployment option.

Recommended Reading and Experimenting

There are many fantastic SQL authors out there, such as Itzik Ben-Gan, Greg Low, Brent Ozar. They have content on multiple websites. That's just three, and there are many other great ones as well.

I've written on SQL Server, T-SQL, and Data Warehousing concepts in many CODE Magazine articles. You can go to https://codemag.com/People/Bio/Kevin.Goff and see a listing of all my articles.

Here are ones I'd like to call out in particular:

- Two T-SQL articles
  - https://codemag.com/Article/2401051/Stages-of-Data-Some-Basic-SQL-Server-Patterns-and-Practices
  - https://codemag.com/Article/1801051/A-SQL-Programming-Puzzle-You-Never-Stop-Learning
- A Power BI article:
  - Stages of Data: COVID Data, Summary Dashboards, and Source Data Tips (codemag.com)
- Four articles on SQL Server reporting Services:
  - https://codemag.com/Article/1805051/Refactoring-a-Reporting-Services-Report-with-Some-SQL-Magic
  - https://codemag.com/Article/1711061/SQL-Server-Reporting-Services-Eight-Power-Tips
  - https://codemag.com/Article/1705061/SQL-Server-Reporting-Services-Seven-Power-Tips
  - https://codemag.com/Article/1605111/SQL-Server-Reporting-Services-Eight-Power-Tips

I've created data projects and dashboard pages from personal data for everything from my weekly health stats to personal finances. The more you practice, the better!

It's great to read and absorb information from websites and books. Yes, sometimes it's because you're trying to solve a specific problem at work, so you already know you're getting your hands dirty and you just need to how to use your hands. Other times, you might be researching or learning a topic where you haven't gotten your hands dirty. All learning is kinetic in some way—a person could read a book on how to perform open heart surgery 100 times and be able to quote each line in the book, for instance. Well, I'm not saying that implementing a Type 2 changing dimension is open heart surgery, but the more you can demonstrate to others that you CAN do something.... As someone who's interviewed people, I might find someone's personal example (a good example) of implementing a Type 2 SCD, or someone being able to open two query windows with a test table to demonstrate READ COMMMITTED SNAPSHOT, to be very compelling.

One Thing I Won't Talk About (But One Final Thing That I Will)

There are other great tools and technologies that database developers use. One that comes to mind is Python. Database developers who also work on the .NET side will sometimes use Entity Framework. Those who work on the ETL side might use third-party tools such as COZYROC and possibly different Master Data Management tools. The list goes on and on.

I tried to focus more on core skills in this article. And trust me, a month after this issue is printed, I'll say to myself, "Rats, I forgot to mention some approach that could have made this article better." For that reason, I'm splitting this article into multiple parts.

But here's something I do want to say. Those who follow sports might recall a player who was criticized for not giving full effort during practice sessions, and reacted during a press conference by repeatedly saying, "C'mon, we're talking about practice. Not a game, but practice." Nearly all of us can think back to some time in our lives when we greatly improved our skills in some area. I'm betting that more focus and more practice and more studying played a role.

Reading SQL Server books and blogs is great, but what's even greater is taking some of those skills and trying them out on your own databases. Microsoft has AdventureWorks and Contoso Retail demo databases. When I wrote articles on COVID data, I found many Excel/CSV files with statistics. Yes, it took some work to assemble those into meaningful databases, but it was worth it. If you're starting out at a new job, or even applying for a new job, you want people to watch you do something and say, "Wow, that person has obviously done this before."

Summary: What I Almost Called This Article

I've been working on this article for over four months. As most authors can attest, what you start with and what you finish with can be different things. As I look back over this, the content itself didn't change much, but the reasons I wrote it evolved. As I mentioned earlier, I've seen many LinkedIn questions where something like this would be helpful. I also wrote this because I wanted to share what things I've seen many times. I'll never claim to have all the answers on what makes a good database developer, but I've instructed people at a tech school whose mission was to help people get jobs (or get better jobs) and I've mentored other developers. I always wanted to take an inventory of what fundamentals I think others will find important, just to make sure I hadn't forgotten anything (and I'll freely admit that I'd forgotten the specifics of Fill Factor). I also know someone who's considering a career in this industry. I've been successful in my career: I've made many mistakes, and I've learned hard lessons as well! I wanted to look back on what areas of knowledge have helped me to be successful.

I've been talking to a developer that I mentored for a few years. Their first response was, "Wow, you're really trying to expose interviewers!!!" As Eric Idle said to John Cleese during the famous "Nudge Nudge Wink Wink" skit: "Oh no, no, no, ***(pause)***, YES!"

There are topics I covered in this article in more detail than others. I went into a fair amount of detail on the Snapshot Isolation Level, but only briefly talked about Change Data Capture and logging. There are other web articles out there, including some from me. As I've linked in this article, there were some topics that I've previously covered in CODE Magazine and didn't want to repeat.

Kevin S. Goff
CODE

Ready to Modernize a Legacy App?

Need advice on migrating yesterday's **legacy applications** to today's modern platforms? Take advantage of **CODE Consulting's** years of experience and contact us today to schedule a **FREE** consulting call to discuss your options.

No strings. No commitment.

For more information, www.codemag.com/consulting or email us at info@codemag.com.

From SOAP to REST to GraphQL

Simple Object Access Protocol (SOAP) can be used to build web services that support interoperability between different platforms and technologies. REST (an acronym for Representational State Transfer) is another popular way of building lightweight APIs that can run over HTTP. As an open-source query language, GraphQL promises a more potent way of accessing information than SOAP or REST alone. This article aims to provide a comprehensive overview of the evolution of web APIs, exploring the transition from SOAP to REST, and finally to GraphQL. It will delve into the motivation behind each architectural style, and their characteristics, benefits, and drawbacks. By understanding the progression from SOAP to REST and the emergence of GraphQL, developers can make informed decisions when choosing the right API design for their projects.

If you're to work with the code examples discussed in this article, you need the following installed in your system:

- Visual Studio 2022
- .NET 8.0
- ASP.NET 8.0 Runtime

If you don't already have Visual Studio 2022 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

In this article, I'll examine the following points:

- SOAP, REST, and GraphQL and their benefits
- The key differences between SOAP, REST, and GraphQL
- The benefits and drawbacks of SOAP, REST, and GraphQL
- How to use each of these tools in enterprise apps

After gaining this knowledge, you'll build three applications: one each using SOAP, REST, and GraphQL.

What Is Simple Object Access Protocol (SOAP)?

Simple Object Access Protocol (SOAP) is a communication protocol for data exchange in a distributed environment that allows different applications to interact with one another over a network by leveraging XML as the message format. You can take advantage of SOAP to build interoperable web services that work with disparate technologies and platforms. The structure and content of XML messages, as well as a set of communication guidelines, are outlined in a SOAP document.

Note that ASP.NET Core doesn't have any built-in support for SOAP. Rather, the .NET Framework provides built-in support for working with ASMX and WCF. Using third-party libraries, you can still build applications that leverage SOAP in ASP.NET Core. **Figure 1** demonstrates how SOAP works.

Anatomy of a SOAP Message

A SOAP (Simple Object Access Protocol) message is an XML-based structure used to exchange information between web services across diverse networks and platforms. A typical SOAP message is comprised of several elements that define the message's structure, content, and optional features. SOAP messages are designed to be extensible, neutral, and independent of any specific programming model or transport protocol, typically HTTP or HTTPS.

A typical SOAP message comprises four key elements, as shown in **Figure 2**.

- Envelope
- Header (optional)
- Body
- Fault

Here's the how the structure of a typical SOAP message looks:

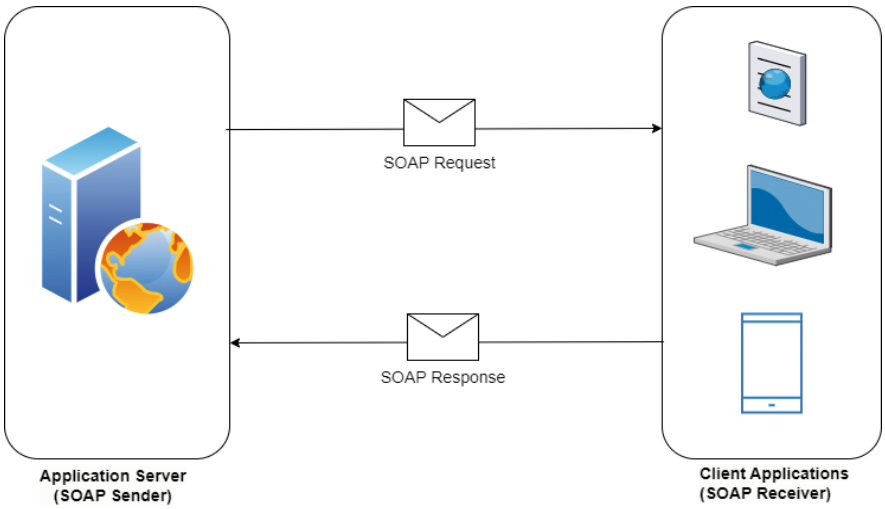

Figure 1: Simple Object Access Protocol (SOAP) at work

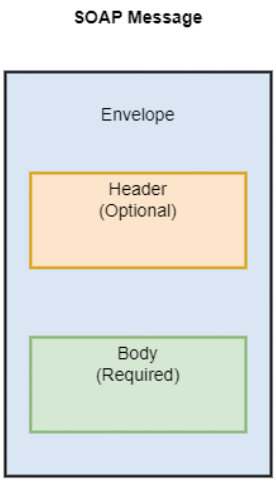

Figure 2: A SOAP message

Joydip Kanjilal
joydipkanjilal@yahoo.com

Joydip Kanjilal is an MVP (2007-2012), software architect, author, and speaker with more than 20 years of experience. He has more than 16 years of experience in Microsoft .NET and its related technologies. Joydip has authored eight books, more than 500 articles, and has reviewed more than a dozen books.

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap=
"http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle=
"http://www.w3.org/2003/05/soap-encoding">

<soap:Header>
          <!-- SOAP Header (Optional) -->
</soap:Header>

<soap:Body>
          <!-- SOAP Body -->
  <soap:Fault>
          <!-- SOAP Fault -->
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

SOAP Envelope

Every SOAP message is encapsulated inside a root element called SOAP Envelope. It defines the XML namespace and contains two mandatory child elements: the SOAP Header and the SOAP Body.

```
<soap:Envelope
xmlns:soap=
"http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle=
"http://www.w3.org/2003/05/soap-encoding">
```

SOAP Header

The SOAP Header is an optional element that contains additional information or metadata about the SOAP message, such as authentication credentials, security tokens, or routing instructions.

```
<soap:Header>
  <!-- Optional SOAP Header elements -->
</soap:Header>
```

SOAP Body

The SOAP Body represents the main body of the SOAP message, which contains the data or parameters for the method being sent to the web service. It should be noted that the SOAP Body element is mandatory. It can have one or more child elements that represent the actual payload and one or more fault elements in the event of an error.

```
<soap:Body>
  <!-- SOAP Body content -->
</soap:Body>
```

SOAP Fault

During the processing of a SOAP message, an optional element called SOAP Fault can be used to convey error or fault information back to the client in case of errors or exceptions that occur during the process.

```
<soap:Fault>
  <!-- Fault details -->
</soap:Fault>
```

Example SOAP Request and Response

The format of a typical SOAP request looks like this:

```
POST /<host>:<port>/<context>/
<database ID> HTTP/1.0
Content-Type: text/xml; charset=utf-8

<?xml version="1.0"?>
<env:Envelope  xmlns:env=
"http://schemas.xmlsoap.org/soap/envelope/">

<env:Header>
</env:Header>

<env:Body>
</env:Body>

</env:Envelope>
```

The following code snippet illustrates how a typical SOAP request looks:

```
POST /ProductPrice HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml;
charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap=
"http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle=
"http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m=
"http://www.abcxyz.org/product">
  <m:GetProductPrice>
    <m:ProductCode>HP_Envy_i9</m:ProductCode>
  </m:GetProductPrice>
</soap:Body>

</soap:Envelope>
```

The format of a typical SOAP response looks like this:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8

<?xml version="1.0"?>
<env:Envelope  xmlns:env=
"http://schemas.xmlsoap.org/soap/envelope/">

<env:Header>
</env:Header>

<env:Body>
</env:Body>

</env:Envelope>
```

And here's how a SOAP response to the above SOAP request looks:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml;
```

```
charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap=
"http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle=
"http://www.w3.org/2003/05/soap-encoding">

<soap:Body
xmlns:m="http://www.abcxyz.org/product">
  <m:GetProductPriceResponse>
    <m:Price>6500.00</m:Price>
  </m:GetProductPriceResponse>
</soap:Body>
</soap:Envelope>
```

WSDL, UDDI, and Binding

This section provides a brief overview of the WSDL and UDDI. It also briefly discusses SOAP binding.

Web Services Description Language (WSDL)

Web Services Description Language, or WSDL, is a language based on XML used for the purpose of describing and locating web services. A standard format is used to describe an interface to a web service that includes operations, inputs, output parameters, and the message format.

Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discover, and Integration, or UDDI, represents a platform-independent registry intended to provide a standard mechanism for service discovery, registration, and versioning to enable organizations to publish and discover web services on the internet.

A UDDI registry is independent of any platform intended to provide a standard mechanism for the discovery, registration, and versioning of services, thereby enabling businesses to publish and discover web services on the internet. The key components of the SOAP architecture work together to define a standardized message format for exchanging structured data between applications using different transport protocols.

SOAP Binding

SOAP binding defines how SOAP messages are transmitted using transport protocols, such as HTTP, SMTP, or TCP. This document specifies the message format, SOAP encoding style, and binding rules that must be followed when sending messages over SOAP.

SOAP Web Services Architectural Components

The SOAP web services architecture thrives on the synergy among the following three components, as shown in **Figure 3**:

- **Service provider:** This is the component that provides the web service and encompasses the application itself, the platform on which the application executes, and the middleware.

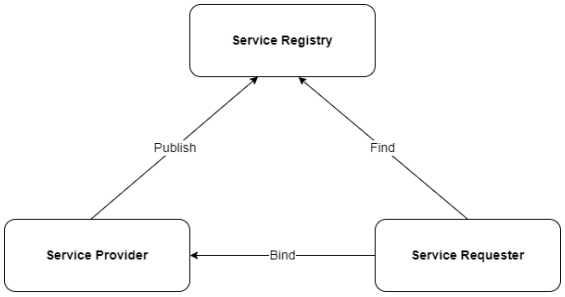

Figure 3: Interactions between components in a web service

- **Service requester or the service consumer:** This component is responsible for requesting a resource from the service provider and, similar to the service provider, it encompasses the application, the platform, and the middleware.
- **Service registry:** This is an optional component that resides in a central location so that service providers can publish service descriptions and the service requesters or service consumers can look up those service descriptions.

Strategies for Building a SOAP Service

To build a SOAP service, you can use any of the following: WCF connected service, third-party libraries, or manual implementation. Let's have a look.

WCF Connected Service

A Connected Service in Windows Communication Foundation (WCF) represents a feature in Visual Studio that facilitates the use of and access to web services, including SOAP-based services. Using it, you can create client code capable of communicating with a web service seamlessly.

Third-Party Libraries

You can also create SOAP services using third-party libraries such as Apache CXF, JAX-WS (Java API for XML Web Services), and Spring-WS (Spring Web Services). You can also leverage Windows Communication Foundation (WCF) for building SOAP services in .NET environment.

Manual Implementation

SOAP messages can be manually constructed and processed using programming languages that support XML parsing and SOAP protocols, such as C# or Java. To do this, follow these steps:

1. Define the service interface, operations, messages, and bindings in the WSDL document.
2. Leverage the programming language of your choice to implement the service logic.
3. Use XML parsing libraries or manually handling XML serialization and deserialization for SOAP messages is the option.
4. Keep an eye on the HTTP server for SOAP requests and responding appropriately.

DataContract and ServiceContract

In SOAP, DataContract and ServiceContract are key concepts used to define the structure of data and the operations supported by the service.

DataContract

In a DataContract, the terms and conditions used in the data exchange between data providers and consumers are outlined along with the format, structure, quality, semantics, etc. It's typically defined using XML Schema Definition (XSD) or a similar language in SOAP. These could be complex types that are passed between service operations. You should decorate your classes or structs with the DataContract attribute to indicate that they are serializable and understandable by the service provider and the service consumer. Leverage the DataMember attribute to decorate the properties or methods of classes or structs that should take part in the serialization process.

Here's an example of a typical data contract:

```
[DataContract]
public class MyClass
{
    [DataMember]
    public int Id { get; set; }

    [DataMember]
    public string Text { get; set; }
}
```

ServiceContract

SOAP web services expose operations and methods through the ServiceContract that specifies the set of available operations, their input parameters, and return types, i.e., the type of the return values. A ServiceContract consists of an interface or class implemented in the server-side code and acts as a blueprint for service implementation, specifying the operations clients can invoke.

An OperationContract represents an individual operation within a ServiceContract and specifies the operation name, input parameters, and output types. You must apply the OperationContract attribute to a service operation to indicate that clients can access the operation.

The following code snippet illustrates a service contract:

```
[ServiceContract]
public interface IMyDemoService
{
    [OperationContract]
    string GetText(int id);
}
```

Implement a SOAP Service in ASP.NET Core

In this section, I'll examine how to build a SOAP service in ASP.NET Core. The section that follows outlines the series of steps needed to create a new ASP.NET Core Web API project in Visual Studio.

Create a New ASP.NET Core 8 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the "Create a new project" window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name as SOAP_Demo and the path where it should be created in the "Configure your new project" window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the "Place solution and project in the same directory" checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," "Do not use top-level statements," and the "Enable OpenAPI support" checkboxes are unchecked because you won't use any of these in this example.
6. Remember to leave the **Use controllers** checkbox checked because you won't use minimal API in this example.
7. Click Create to complete the process.

A new ASP.NET Core Web API project is created. You'll use this project to build the SOAP service in ASP.NET Core.

Install NuGet Package(s)

So far so good. The next step is to install the necessary NuGet Package(s). To install the required package into your project, right-click on the solution and then select Manage NuGet Packages for Solution.... Now search for the package named SoapCore in the search box and install it. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package SoapCore
```

You can also install this package by executing the following commands at the Windows Shell:

```
dotnet add package SoapCore
```

Create the Data Contract

To create the data contract, create a new class named Customer in a file called Customer.cs and write the following code in there:

```
[DataContract]
public class Customer
{
    [DataMember]
    public int Id { get; set; }
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    [DataMember]
    public string Address { get; set; }
}
```

Create the CustomerRepository

The CustomerRepository class extends the ICustomerRepository interface and implements its methods, as shown in the code given in **Listing 1**.

Create the Service Contract

To create a service contract, create an interface called ICustomerService and write the code given in **Listing 2** in there. The CustomerService class extends the ICus-

tomerService interface and implements its methods. Once you've created the ICustomerService interface, create a new class named CustomerService and write the code given in **Listing 2** in there.

Configure the SOAP Service

Lastly, you should configure your SOAP service to run it. To do this, add ICustomerRepository and ICustomerService instances to the container using the following code snippet:

```
builder.Services.AddScoped
<ICustomerService,
CustomerService>();
builder.Services.AddScoped
<ICustomerRepository,
CustomerRepository>();
```

This enables you to use dependency injection to retrieve these instances at runtime. The complete source code of the Program.cs file is given in **Listing 3** for your reference.

Create the SOAP Client

You'll now create a simple Console application to consume the SOAP service you created earlier. The SOAP client application consumes the SOAP service and displays the data retrieved at the Console window.

You can create a project in Visual Studio 2022 in several ways. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

To create a new Console Application Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the Create a new project window, select Console App, and click Next to move on.

Listing 1: The ICustomerRepository interface

```
public interface ICustomerRepository {
  public Task < List < Customer >> GetCustomers();
  public Task < Customer > GetCustomer(int id);
}

public class CustomerRepository: ICustomerRepository {
  private readonly List < Customer > customers =
  new List < Customer > () {
    new Customer() {
        Customer_Id = 1, FirstName = "Rob",
        LastName = "Miles", Address = "Boston, USA"

      },
      new Customer() {
        Customer_Id = 2, FirstName = "Lewis",
        LastName = "Walker", Address = "London, UK"
      },
      new Customer() {
        Customer_Id = 3, FirstName = "Carlton",
        LastName = "Kramer", Address = "New York, USA"
      }
  };

  public async Task < List < Customer >> GetCustomers() {
    return await Task.FromResult(customers);
  }
  public async Task < Customer > GetCustomer(int id) {
    return await Task.FromResult
    (customers.FirstOrDefault(x => x.Customer_Id == id));
  }
}
```

Listing 2: The ICustomerService interface

```
[ServiceContract]
public interface ICustomerService
{
    [OperationContract]
    Task<List<Customer>> GetCustomers();

    [OperationContract]
    Task<Customer> GetCustomer(int id);
}

public class CustomerService : ICustomerService
{
    private readonly ICustomerRepository _customerRepository;
    public CustomerService
    (ICustomerRepository customerRepository)
    {
        _customerRepository = customerRepository;
    }
    public async Task<List<Customer>> GetCustomers()
    {
        return await _customerRepository.GetCustomers();
    }
    public async Task<Customer> GetCustomer(int id)
    {
        return await _customerRepository.GetCustomer(id);
    }
}
```

Listing 3: Configuring SOAP endpoints in the Program.cs file

```
using SOAP_Demo;
using SoapCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddScoped
<ICustomerService, CustomerService>();
builder.Services.AddScoped
<ICustomerRepository, CustomerRepository>();
builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseRouting();
app.UseAuthorization();
app.MapControllers();

app.UseEndpoints(endpoints =>
{
    _ = endpoints.UseSoapEndpoint
        <ICustomerService>("/CustomerService.asmx",
        new SoapEncoderOptions(),
        SoapSerializer.XmlSerializer);
});

app.Run();
```

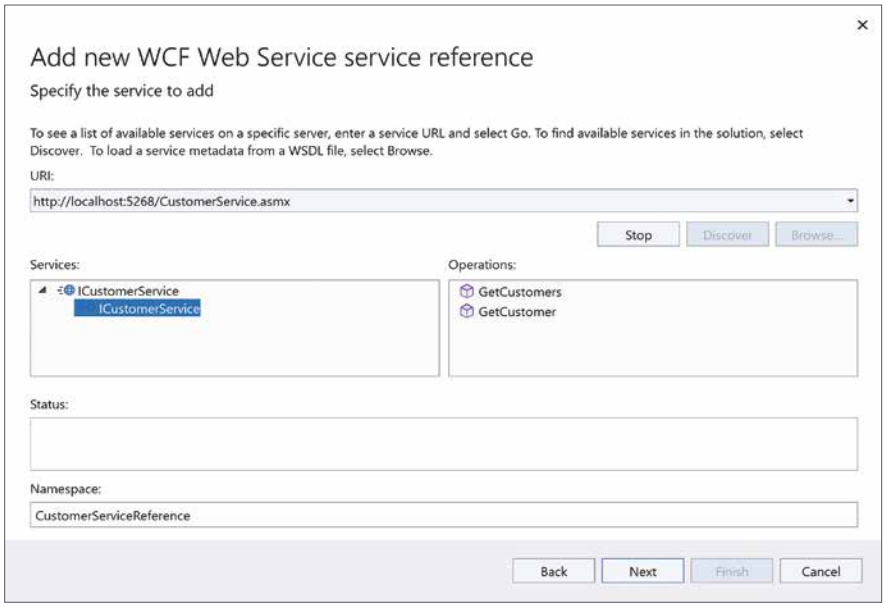

Figure 4: Adding a new Service Reference

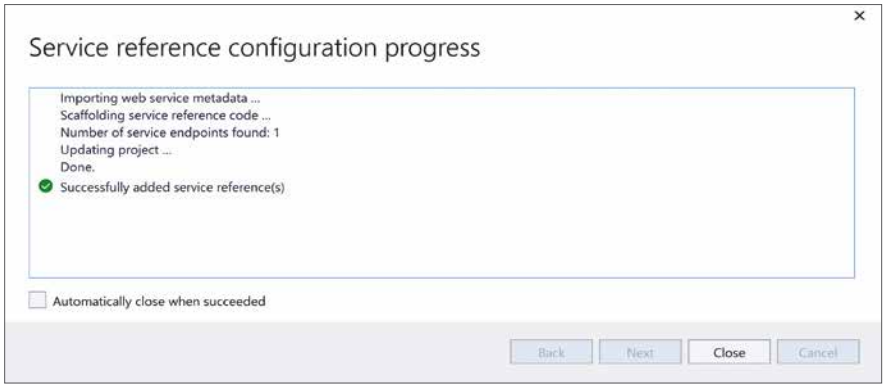

Figure 5: SOAP service reference added

3. Specify the project name as SOAP_Cient and the path where it should be created in the Configure your new project window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the Place solution and project in the same directory checkbox. Click Next to move on.
5. In the next screen, specify the target framework you'd like to use for your console application.
6. Click Create to complete the process.

Once your Console application is ready, you can follow these steps to add a reference to your SOAP service into the client application.

1. Right click on the client application and select Add -> Service Reference, as shown in **Figure 4**.
2. Click on Next three times and let the default selections be used for this example.
3. Click on Submit to initiate the add reference process.

Once the reference to the SOAP service has been added successfully, the following screen will be displayed, as shown in **Figure 5.**

You'll observe that a file named References.cs has been added to the client application. You can now write the following piece of code to invoke the GetCustomers service operation asynchronously.

```
ICustomerService soapServiceChannel =
    new CustomerServiceClient
(CustomerServiceClient.EndpointConfiguration.
BasicHttpBinding_ICustomerService_soap);
var response = await
    soapServiceChannel.GetCustomersAsync();
```

The following code snippet shows the complete source code of the Program.cs file of the client application.

```
using CustomerServiceReference;

ICustomerService soapServiceChannel =
    new CustomerServiceClient
(CustomerServiceClient.
    EndpointConfiguration.
BasicHttpBinding_ICustomerService_soap);
var response = await soapServiceChannel.GetCustomersAsync();

foreach (Customer customer in response)
    Console.WriteLine(customer.FirstName);
```

When you run the application, the first names of the customers will be displayed at the console window.

Call the SOAP Service from Postman

You can also call the GetCustomers service operation using Postman, as shown in **Figure 6**.

From SOAP to REST

Representational State Transfer (REST) is a lightweight, and scalable approach to build web services that can run over HTTP. It's often used for building distributed systems and APIs due to its simplicity, flexibility, and wide adoption among developers. REST is a set of architectural constraints.

RESTful systems adhere to a set of constraints that standardize the communication between components, enhancing scalability and performance. By leveraging HTTP methods such as GET, PUT, POST, and DELETE, REST enables the creation of well-defined and predictable APIs. This approach allows loose coupling between client and server, promoting flexibility and ease of maintenance in distributed systems.

This not only promotes reliability and resilience, but also allows scalability and efficient communication between components. **Figure 7** shows a typical REST-based application at work. By understanding the principles of REST, developers can design robust and flexible systems that are scalable and high performant.

Resources

A fundamental tenet of REST is resources, which are distinguished by unique URLs and may be managed via conventional HTTP methods such as GET, POST, PUT, and DELETE. This approach allows clients to access and modify representations of these resources through well-defined interfaces. Resources can be represented in various for-

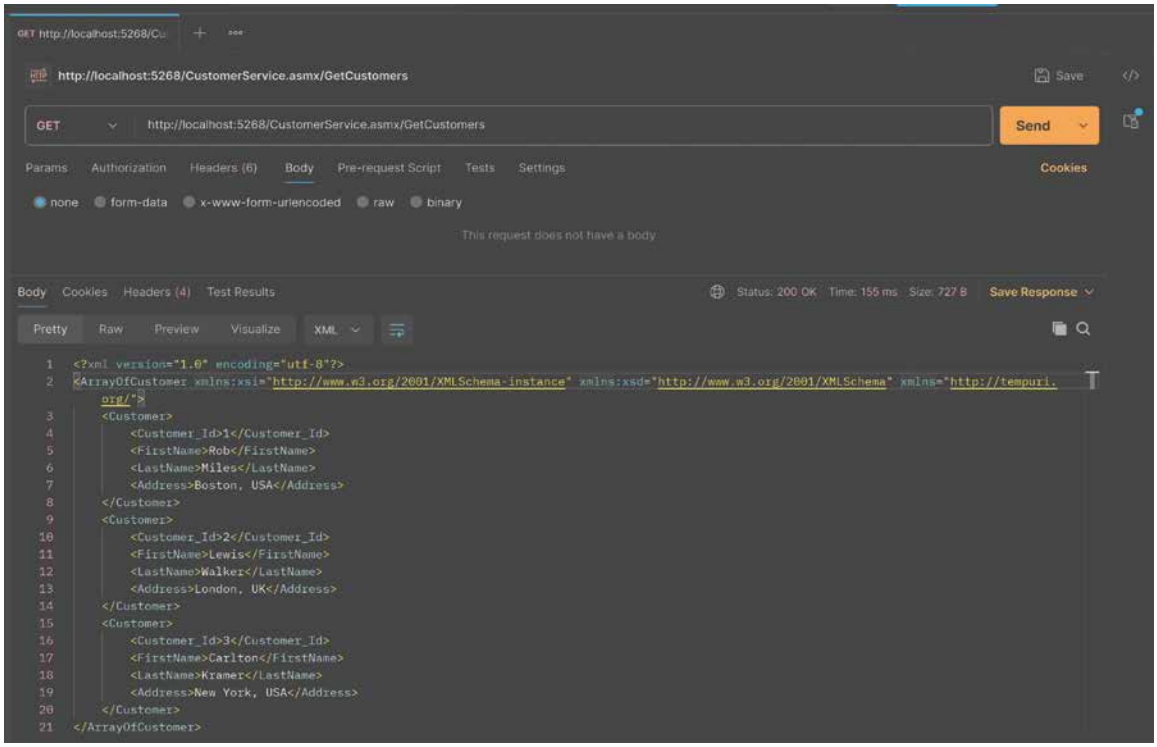

Figure 6: The SOAP response of the GetCustomers service operation in Postman

mats, allowing more flexibility in how data is manipulated and transferred.

Common Misconceptions about REST

In this section, I'll examine a few common misconceptions related to REST among the developer community.

REST Is a Protocol

It should be noted that REST is not a standard or a protocol. Representational State Transfer (REST) refers to an architectural style and a set of architectural constraints used for developing networked applications that defines a set of guidelines and principles for developing web services that are scalable, maintainable, and loosely coupled.

REST Is Only Used for Web Services

Although REST was originally designed for creating web services, it can also be used for other types of applications such as mobile apps or IoT devices. As long as the principles of statelessness, client-server architecture, and resource-based communication are followed, any type of application can be built using REST.

REST Requires the Use of HTTP

Although HTTP is commonly used in conjunction with REST due to its widespread adoption and support for various request methods, it's not a requirement. The principles of resource identification and manipulation are applicable to any network protocol.

Every API that Uses HTTP Is Automatically Considered RESTful

No, not actually. Just because an API uses HTTP doesn't mean it follows the principles of REST. A genuinely RESTful API should adhere to all the constraints set out by the architectural style, including statelessness, caching, uniform interface, etc.

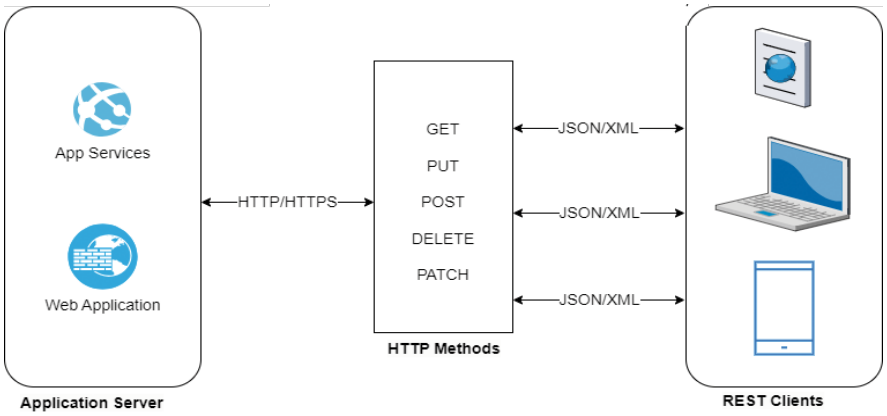

Figure 7: REST application at work

URLs Must Contain Nouns Only

Another misconception about REST is that it can only be used with HTTP. Although RESTful APIs typically use HTTP as the underlying protocol, REST itself is not tied to any specific protocol and can be implemented over other protocols like CoAP or WebSocket.

Key Principles of REST

There are several key principles that underpin REST architecture, which are listed in this section. By adhering to these key principles, developers can design scalable, reliable, and efficient web services that meet the demands of today's applications as far as performance and flexibility is concerned.

Resource-Based

In REST, a resource is an object, data, or service that a client may access across a network, usually the internet, using a specified URL. Resource identifiers in a RESTful system are known as URIs (Uniform Resource Identifiers), and they are conceptual entities or data representations. In a RESTful architecture, you can identify resources using URIs. Resources are concepts that can be represented in any format such as HTML, XML, JSON, etc.

Code on Demand

A RESTful architecture provides support for an optional feature that allows code to be downloaded and executed as applets or scripts to extend client functionality. The number of features that need to be pre-implemented is reduced, simplifying the client experience. The ability to download features after deployment improves the extensibility of the system. By delivering executable code to the client, the server may enhance the client's capability via the code on demand feature. When you fill out any registration form on a site, your web browser alerts you if there are errors. For example, when you fill in a registration form, your browser displays any errors while you type, such as incorrect SSN numbers, email addresses, etc.

Stateless

REST is stateless, requiring each client request to provide all information needed for processing as part of query parameters, request headers or URI. It should be noted that in a typical RESTful architecture, the server doesn't retain any client-specific information between requests, thus enabling improved scalability and load balancing by allowing any server to process any client request.

Layered Architecture

Requests and responses to REST APIs are routed through several layers spread across multiple tiers. A layered system architecture is one in which the application is split across various layers to isolate presentation, application processing, and data management in a layered architecture. These layers work together to fulfill client requests while clients are unaware of them. It helps if you design your RESTful architecture to split your RESTful services across multiple layers, thereby fostering the separation of concerns in your application.

Uniform Interface

RESTful architecture encourages uniform and standardized interfaces for interaction with a resource as part of its basic principles. The interface typically consists of four main HTTP methods: GET, POST, PUT, and DELETE. RESTful services should have a uniform interface. In a RESTful architecture, the server transmits information to a client in a standard format. In RESTful architecture, a formatted resource is known as a representation. Note that the format of a resource may differ from its internal representation on the server. A server can, for instance, store data in text and send it in HTML format. In a RESTful architecture, by decoupling implementations from the services they provide, they can evolve independently.

Client-Server Architecture

As a rule of thumb, a RESTful architecture should be based on a client-server architecture. Although the client requests resources from the server, the server provides resources as appropriate to the authorized clients. In a typical client-server architecture, the client and the server are decoupled and they don't have any knowledge of each other, thereby enabling them to grow and evolve independent of each other.

Cacheable

Resources should be cacheable to improve network efficiency. Each response should state whether it is cacheable on the client side and for how long. When the client requests data in the future, it retrieves the data from its cache, thereby eliminating the need to transmit the request to the server again for future requests. When managed effectively, caching reduces client and server traffic, enhancing availability and performance. However, you should take proper measures to ensure that clients don't have stale data. The server can include caching-related headers (e.g., Cache-Control or Last-Modified) in the response to indicate to the client how long the response can be cached.

Benefits of Using REST

The following are the benefits of REST at a glance:

- **Scalability:** RESTful architectures are inherently scalable due to their stateless nature, allowing for easy scaling of services to meet changing demands and making it easier to handle a large number of requests.
- **Simplicity:** With its emphasis on standard HTTP methods and status codes, REST simplifies the communication process between clients and servers.
- **Flexibility:** The ability to work with different data representations and to support various client types adds flexibility to RESTful services.
- **Performance:** By caching resources, you can reduce client-server interactions, which can greatly improve the performance of your application.
- **Interoperability:** REST APIs work over standard protocols like HTTP, enabling seamless communication between different systems regardless of their implementation details.
- **Technology Agnostic:** REST APIs are technology agnostic, enabling you to write both server and client applications in different programming languages. The underlying technology can also be changed on either side of the communication, if needed, without affecting the functionality.

How Does REST Work?

Initially, the client sends a request to the server over HTTP in order to initiate communication with the server. The server acknowledges and processes the request and then produces a response, as appropriate. Responses are usually in the form of data, such as JSON or XML, which the client can display or use. REST APIs use predefined methods like GET, PUT, POST, DELETE to perform diverse operations on the server. These methods correspond to various actions, such as retrieving data, updating old data, creating new data, and deleting old data.

What Are REST APIs? How Do They Work?

REST APIs communicate data between client and server using HTTP requests. Once a client sends a request, the

server acknowledges and processes it and then sends an appropriate response to the client. Responses are usually in the form of data, such as JSON or XML, which the client can display or use. REST APIs use predefined methods like GET, POST, PUT, and DELETE to perform diverse operations on the server. The methods apply to various actions, such as retrieving data, creating new data, updating old data, and deleting old data. Using these principles and methods, REST APIs can effectively communicate and transfer data between applications. A REST API encompasses a collection of guidelines and standards that can help you build applications that can interact and share information over the internet. It adheres to REST principles, a stateless architectural approach, to develop scalable and adaptable web services.

Challenges of REST

Although REST offers many advantages, some challenges exist, such as maintaining statelessness leading to increased network overhead and designing consistent and meaningful URI structures can be complex in large-scale applications.

Here are the key challenges of RESTful architecture:

- **Limited support for performing complex operations**: It's important to note that the REST API can only be used for CRUD (Create, Read, Update, Delete) operations through a limited set of HTTP methods (GET, PUT, POST, DELETE). Performing complex operations may require multiple requests or custom endpoints.
- **No standard error handling mechanism:** There's no standardized error handling in REST. There can be a range of REST implementations for implementing consistent error handling mechanisms, such as error codes, error messages, and error formats.
- **Over-fetching and under-fetching:** REST APIs typically return a fixed representation of resources. When clients only require a subset of the resource data or need additional data not included in the response, it can result in inefficiencies.
- **Versioning:** As REST APIs evolve, introducing changes can break existing client implementations. When many clients consume the API at the same time, maintaining backward compatibility and versioning can be challenging.
- **Security issues:** Security challenges exist when using REST APIs, such as data exposure, unauthorized access, and protection against XSS and CSRF attacks. Implementing proper security measures, such as authentication, authorization, and encryption, is imperative.

The Future of REST

With technology advancing at such a lightning-fast pace, REST seems to have a bright future. With its simplicity and scalability, REST is expected to remain a fundamental architectural style for designing networked applications. As more businesses embrace cloud computing and microservices architecture, REST will play a crucial role in enabling seamless communication between various services and systems. Furthermore, with the rise of Internet of Things (IoT) devices and mobile applications, RESTful APIs will be essential in facilitating data exchange between these interconnected devices. As the web industry continues to evolve, REST's flexibility and compatibility with different programming languages make it well-positioned to meet the demands of modern web development practices.

Implementing a RESTful Application in ASP.NET Core

Let's now build a RESTful web application in ASP.NET Core. Follow the steps outlined in a previous section where you've built a ASP.NET Core Web API application in Visual Studio. The only difference is that you'll build a minimal API application in Visual Studio by leaving the **Use controllers** checkbox unchecked. You'll use this project to build the RESTful application in ASP.NET Core.

Create the Model Class

Assuming that the minimal API application has been created, create a new class named Product in a file having the same name with a .cs extension and write the following code in there:

```
public class Product
{
   public int Product_Id { get; set; }
   public string Product_Name { get; set; }
   public string Description { get; set; }
   public string SKU { get; set; }
   public decimal Price { get; set; }
}
```

For the sake of simplicity and brevity, I'll skip other model classes here.

Create the ProductRepository

The ProductRepository abstracts all calls to the database. The ProductRepository class extends the IProductRepository interface and implements its methods as shown in **Listing 4**.

Create the RESTful API

Finally, let's create the API endpoints in the Program.cs file. Remember, this is a Minimal API application, so you don't use any APIController class here. The following code snippet shows how you can create the API endpoints:

```
app.MapGet("/getproducts", async
    (IProductRepository productRepository)
    => await productRepository.GetProducts());

app.MapGet("/getproduct/{id:int}", async
    (IProductRepository productRepository, int id)
    => await productRepository.GetProduct(id)
    is Product product ?
Results.Ok(product) : Results.NotFound());

app.MapPost("/addproduct", async
    (IProductRepository productRepository,
Product product) =>
{
    await productRepository.AddProduct(product);
    return Results.Created
($"/addproduct/{product.Product_Id}", product);
});

app.MapDelete("/deleteproduct/{id}", async
```

Listing 4: The IProductRepository interface

```
public interface IProductRepository
{
    public Task<List<Product>> GetProducts();
    public Task<Product> GetProduct(int id);
}

public class ProductRepository
{
    private readonly List<Product> products =
    new List<Product> {
    new Product {
        Product_Id = 1,
        Product_Name = "HP Envy Laptop",
        SKU = "HPL/i9/1TB",
        Description =
        "HP Envy Laptop i9 32 GB RAM 1 TB SSD",
        Quantity = 100,
        Price = 6500.00m
    },
    new Product {
        Product_Id = 2,
        Product_Name = "Lenovo Legion Laptop",
        Description =
        "Lenovo Legion Laptop i7 16 GB RAM 1 TB SSD",
        SKU = "Len/i9/1TB/SSD",
        Quantity = 150,
        Price = 6000.00m
    },
    new Product {
        Product_Id = 3,
        Product_Name = "DELL XPS Laptop",
        Description = "Dell XPS 9730 Laptop,
        Intel Core i9 32GB 1TB SSD",
        SKU = "DEL/i9/1TB",
        Quantity = 50,
        Price = 7500.00m
    }
  };
  public async Task<List<Product>> GetProducts()
  {
        return await Task.FromResult(products);
  }
  public async Task<Product> GetProduct(int id)
  {
        return await Task.FromResult
        (products.FirstOrDefault
        (x => x.Product_Id == id));
  }
}
```

```
    (int id, IProductRepository productRepository) =>
{
    await productRepository.DeleteProduct(id);
});
```

Although the MapGet method has been used to create the HttpGet endpoints, the MapPost method has been used to create the HttpPost endpoint. Similarly, the MapDelete method has been used here to create the HttpDelete endpoint.

Replace the content of the Program.cs file with source code given in **Listing 5**. You can now execute the application and then use Postman to launch the endpoints.

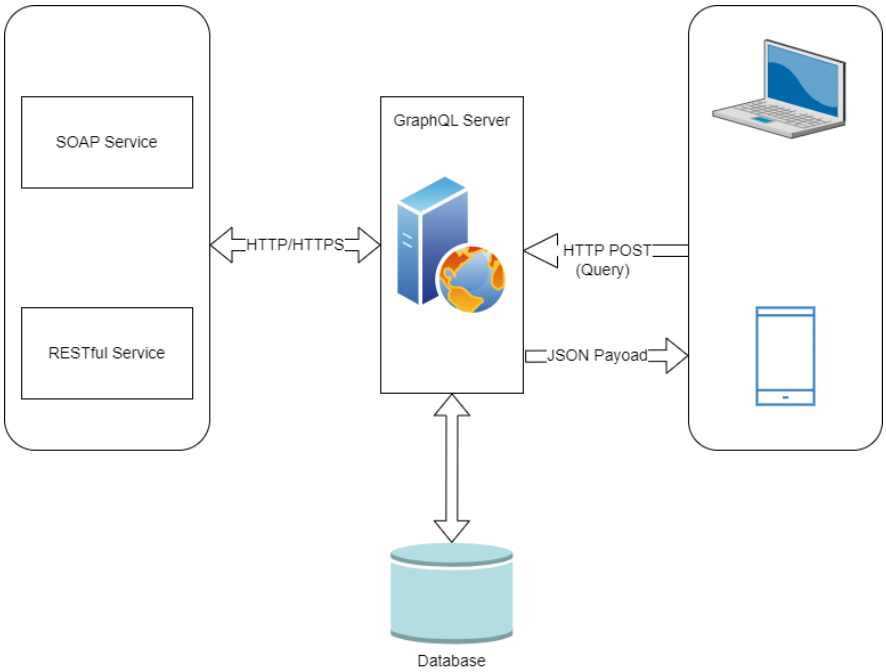

Figure 8: A typical GraphQL API Architecture

From REST to GraphQL

It was Facebook's desire to find a method of accessing data that was both more efficient and more elegant that led to the development of GraphQL in the year 2012. GraphQL achieves this by adopting a declarative approach toward retrieving and manipulating data and enabling the API clients to request only the data they require, thereby improving performance and efficiency while reducing unnecessary data transfer. In order to retrieve data from different endpoints using REST APIs, the client must make multiple requests, which can be inefficient and time intensive. If your API is proficient in retrieving all the data your application requires, you won't encounter any issues with over-fetching or under-fetching.

The primary goal of GraphQL is to simplify the process of querying data by enabling clients to request exactly the format and structure of data they need in a single call. Remember, you need to make several calls to your server to retrieve data split across multiple data stores, as shown in **Figure 8**.

Here are the key reasons why GraphQL is considered a better alternative to REST in some use cases.

- **Data Retrieval:** REST often necessitates accessing various endpoints to collect different pieces of information, potentially leading to over-fetching or under-fetching data. GraphQL enables clients to specify the precise data they require from a single endpoint, minimizing the volume of data transmitted across the network. Over fetching is defined as a situation in which an API returns more information than is necessary for your application. For instance, a client might request Order ID and Order Date and receive Order Id, Order Date, and Product Id instead. Under fetching occurs when an API doesn't provide all the data your application requests. A client may request Order Id and Order Date, but only receive Order Id. As a consequence, the client may experience

Listing 5: Configuring the REST endpoints in the Program.cs file

```
using REST_Demo;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddScoped<IProductRepository,
        ProductRepository>();
var app = builder.Build();

// Configure the HTTP request pipeline.

app.MapGet("/getproducts", async
    (IProductRepository productRepository)
    => await productRepository.GetProducts());
app.MapGet("/getproduct/{id:int}", async
    (IProductRepository productRepository, int id)
    => await productRepository.GetProduct(id)
    is Product product ?
    Results.Ok(product) : Results.NotFound());

app.MapPost("/addproduct", async
    (IProductRepository productRepository,
     Product product) =>
{
    await productRepository.AddProduct(product);
    return Results.Created
    ($"/addproduct/{product.Product_Id}", product);
});

app.MapDelete("/deleteproduct/{id}", async
    (int id, IProductRepository productRepository) =>
{
    await productRepository.DeleteProduct(id);
});

app.Run();
```

reduced performance and improper or inefficient use of memory, CPU, and network resources.

- **Versioning:** Versioning in REST APIs manages changes to the APIs by assigning different versions, such as v1, v2, and so on. GraphQL eliminates the necessity for version control by allowing clients to specify the data they need in the query, making it easier for APIs to evolve without breaking the existing queries. APIs built with GraphQL do not require separate versioning because clients or API consumers can define their requirements in the query and fetch the required data without breaking the existing queries.
- **Type System:** GraphQL employs a strongly typed schema specifying the data format you can request. This schema functions as a consensus between the client and the server, thereby enabling the early detection of errors. By recognizing potential errors up front, you can resolve the errors in a planned way before they impact your clients.

Benefits and Downsides of GraphQL

Here are the key benefits of GraphQL:

- **Efficient data querying:** With GraphQL, clients can query multiple resources and retrieve related data in a single request. They can traverse the data graph and retrieve only the required data, avoiding the over-fetching of unnecessary fields or nested objects.
- **Reduced network traffic:** GraphQL reduces the network traffic and bandwidth consumption by minimizing the payload size of the responses. This explains why applications that leverage GraphQL often exhibit better performance compared to RESTful applications.
- **Versioning and evolution:** With GraphQL, deprecated fields or types can be marked to signal clients for migration, allowing for smooth API evolution without breaking existing clients.
- **Support for real-time data:** With GraphQL subscriptions, clients can subscribe to specific data changes in real-time. Once subscribed, the clients are notified using events about any changes made to the data in real-time.
- **Strongly typed schema:** GraphQL enforces a robust typing system and a well-defined schema, providing clarity for the available data types and fields. The GraphQL schema defines the structure of the data and the types of operations (queries and mutations) that can be performed, thereby helping with validation and introspection.
- **Improved performance:** For applications that require complex queries combining multiple resources, GraphQL can be more efficient than REST because it can gather all data in a single request.

There are certain downsides as well:

- **Learning curve**: Despite its simplicity, GraphQL is quite complex for those who are unfamiliar with its concepts. The learning curve for designing schemas, resolving queries, and securing GraphQL APIs can be quite steep.
- **Caching challenges:** Due to the dynamic nature of GraphQL queries, client-side and server-side caching can be more challenging compared to REST, where URLs can easily serve as cache keys.
- **Increased complexity:** GraphQL adds more complexity to the server-side implementation in contrast to conventional REST APIs. You should use resolvers to get the data you need. Managing complex queries might incur additional effort.
- **Rate limiting:** Implementing rate limiting in GraphQL is more complex than in REST because it's harder to predict the cost of a query due to its flexible nature.
- **Security considerations:** GraphQL APIs must be carefully designed to avoid potential vulnerabilities. Exposing too much data or functionality through the API can increase the attack surface, making proper authentication and authorization crucial.

GraphQL vs. REST

Although REST and GraphQL are two of the most popular approaches for building APIs, there are subtle differences between the two:

- **Request format:** Each endpoint in REST specifies a set of resources and operations, and the client can typically retrieve or modify all resources using HTTP methods, such as GET, POST, PUT, or DELETE. With GraphQL, clients request data based on a specific structure that matches the server's schema.
- **Data Retrieval:** Each resource in REST can only be accessed through a particular endpoint, mean-

ing the client needs to make multiple requests to retrieve related data or complex object structures. On the contrary, GraphQL enables the client to retrieve the exact data they need with a single query, thereby reducing the number of requests required to fetch the data and minimizing network bandwidth consumption. This can lead to fewer data transfers and more efficient API performance.

- **Type System:** In GraphQL, there's a strongly typed schema system that defines the types, fields, and relationships between them. The client and server have a clear contract using this approach, and advanced tooling capabilities, such as schema introspection and auto generation, are also available. On the other hand, REST APIs do not typically include formal type systems, which makes them flexible but also challenging to handle.
- **Caching:** REST enables you to cache frequently accessed data for faster access during subsequent calls to access the same piece of data, thereby reducing network traffic and improving application performance. REST APIs can use HTTP caching techniques to minimize the data sent between clients and servers. Native caching mechanisms, on the other hand, are not supported by GraphQL APIs. A GraphQL API relies on client-side caching mechanisms to optimize performance because query parameters may affect the responses.
- **Protocol:** Although REST works only with HTTP protocol, there are no protocol constraints in GraphQL. In other words, GraphQL is agnostic of the transport layer—you can use it with any transport layer protocol.
- **Partial Responses:** GraphQL allows clients to retrieve specific information in a single query, minimizing data transmission. This is beneficial for sluggish networks or when accessing APIs via mobile apps.

Applications with complex queries, a large number of data sources, or unpredictable data needs may benefit from GraphQL. REST may be more appropriate for simple CRUD applications when a mapping exists between the resources and their endpoints. However, in REST, resources are typically returned in their entirety, which can lead to over-fetching.

Comparing REST and GraphQL

Here's how REST and GraphQL compare against each other in a typical request/response scenario. Consider two entities, Employee and Address. The former stores employee details and the latter contains address details of the employees. The following code snippets illustrate the requests and responses in REST to retrieve the details (including address information) of an employee.

Request:

```
GET /api/employee?id=1
```

Response:

```
{
  "id": 1
  "name": "Joydip"
}
```

Request:

```
GET /api/address?employee_id=1
```

Response:

```
{
  "street": "Banjara Hills"
  "city": "Hyderabad"
  "country": "India"
}
```

You can do the same in GraphQL in a much more elegant way.

Request:

```
query {
  employee (id: 1) {
    id
    name
   address
   {
     street
     city
     country
   }
  }
}
```

Here's the GraphQL response to the preceding query:

```
{
 "employee": {
   "id": 1
   "name": "Joydip"
   "address": {
     "street": "Banjara Hills"
     "city": "Hyderabad"
     "country": "India"
    }
  }
}
```

Note that you could retrieve the entire data in just one call.

Building a Simple Application Using GraphQL

It's time for writing some code. Let's now examine how to build a simple ASP.NET Core 8 Web API application using GraphQL. A typical Order Processing System is comprised of several entities such as Store, Supplier, Order, Product, Customer, etc. In this example, you'll implement only the Store part of it for simplicity.

Let's now examine how to create a ASP.NET Core 8 project in Visual Studio 2022.

Create a New ASP.NET Core 8 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the "Create a new project" window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name as GraphQL_Demo and the path where it should be created in the "Configure your new project" window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the "Place solution and project in the same directory" checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," and the "Enable OpenAPI support" checkboxes are unchecked because you won't use any of these in this example.
6. As you'll not leverage minimal APIs in this example, leave the **Use controllers** checkbox checked.
7. Click Create to complete the process.

In this application, you'll take advantage of HotChocolate to generate GraphQL schemas. With HotChocolate, you can build an extra layer on top of your application layer that uses GraphQL. It's easy to set up and configure, and it eliminates the clutter of generating schemas.

Install NuGet Package(s)

So far so good. The next step is to install the necessary NuGet Package(s). To install the required packages into your project, right-click on the solution and the select Manage NuGet Packages for Solution.... Now search for the packages named HotChocolate.AspNetCore, and HotChocolate.AspNetCore.Playground in the search box and install them one after the other. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package
HotChocolate.AspNetCore
PM> Install-Package
HotChocolate.AspNetCore.Playground
```

You can also install these packages by executing the following commands at the Windows Shell:

```
dotnet add package
HotChocolate.AspNetCore
dotnet add package
HotChocolate.AspNetCore.Playground
```

Create the Model Class

Create a new class named Order in a file having the same name with a .cs extension and write the following code in there:

```
public class Store
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public string Zip { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
}
```

The other entity classes are not being shown here for brevity and also because this is a minimalistic implementation to illustrate how you can work with GraphQL in ASP.NET Core 7.

Create the IStoreRepository Interface

Create a new .cs file named IStoreRepository in your project and replace the default generated code with the following code snippet:

```
public interface IStoreRepository
{
    public Task<List<Store>> GetStores();
    public Task<Store> GetStore(int Id);
}
```

Create the StoreRepository Class

Next, create a new class named StoreRepository in a file having the same name with a .cs extension. Now write the following code in there:

```
public class StoreRepository : IStoreRepository
{

}
```

The StoreRepository class illustrated in the code snippet below implements the methods of the IStoreRepository interface:

```
public async Task<List<Store>> GetStores()
{
    return await Task.FromResult(stores);
}

public async Task<Store> GetStore(int Id)
{
  return await Task.FromResult(stores.
  FirstOrDefault(x => x.Id == Id));
}
```

The complete source code of the StoreRepository class is given in **Listing 6**.

Register the StoreRepository instance

The following code snippet illustrates how an instance of type IStoreRepository is added as a scoped service to the IServiceCollection.

```
builder.Services.AddScoped
<IStoreRepository, StoreRepository>();
```

Create the GraphQL Query Class

A GraphQL query is defined as a request sent by the client to the server. In GraphQL, clients request data from the server using queries that adhere to a specific structure and syntax per the GraphQL specification. When using GraphQL queries, clients may specify data and the response format.

There are several fields in the query that represent the data that was retrieved from the API. The data contained

Listing 6: The StoreRepository class

```
public class StoreRepository : IStoreRepository
{
    private readonly List<Store> stores =
    new List<Store>
    {
      new Store
      {
        Id = 1,
        Name = "Walmart",
        Address = "274 Reagan Apt. 919",
        City = "Huntington",
        State = "West Virginia",
        Country = "USA",
        Zip = "25049",
        Phone = "1111111111",
        Email = "test@xyz.com"
      },
      new Store
      {
        Id = 2,
        Name = "Amazon",
        Address = "57526 Michelle Ferry Suite 714",
        City = "Edmond",
        State = "Oklahoma",
        Country = "USA",
        Zip = "66347",
        Phone = "1111111111",
        Email = "test@xyz.com"
      },
      new Store
      {
        Id = 3,
         Name = "Harrods",
         Address = "Flat 60 Davis Road",
         City = "Bradford",
         State = "Yorkshire",
         Country = "UK",
         Zip = "BD1 1BL",
         Phone = "1111111111",
         Email = "test@xyz.com"
      }
    };
    public async Task<List<Store>> GetStores()
    {
        return await Task.FromResult(stores);
    }
    public async Task<Store> GetStore(int Id)
    {
        return await Task.FromResult(
         stores.FirstOrDefault(x => x.Id == Id));
    }
}
```

Listing 7: The StoreQuery class

```
using HotChocolate.Subscriptions;

namespace GraphQL_Demo
{
    public class StoreQuery
    {
        public async Task<List<Store>>
        GetAllStores([Service]
        IStoreRepository storeRepository,
        [Service] ITopicEventSender eventSender)
        {
            List<Store> stores =
            await storeRepository.GetStores();
            await eventSender.SendAsync
            ("Returned a List of Stores", stores);
            return stores;
        }
    }
}
```

Listing 8: The StoreType class

```
using HotChocolate.Types;

namespace GraphQL_Demo
{
    public class StoreType :
    ObjectType<Store>
    {
        protected override void Configure
        (IObjectTypeDescriptor
        <Store> descriptor)
        {
            descriptor.Field
            (s => s.Id).Type<IdType>();
            descriptor.Field(s =>
            s.Name).Type<StringType>();
            descriptor.Field(s =>
            s.Address).Type<StringType>();
            descriptor.Field(s =>
            s.City).Type<StringType>();
            descriptor.Field(s =>
            s.State).Type<StringType>();
            descriptor.Field(s =>
            s.Country).Type<StringType>();
            descriptor.Field(s =>
            s.Zip).Type<StringType>();
            descriptor.Field(s =>
            s.Email).Type<StringType>();
            descriptor.Field(s =>
            s.Phone).Type<StringType>();
        }
    }
}
```

in each of these fields can be traversed and retrieved by nesting them. Create a new .cs file named StoreQuery in your project and replace the default generated code with the code given in **Listing 7**.

Create the GraphQL Object Type

In GraphQL, Object Types are used to describe the type of data fetched using your API and they are represented by creating a class that derives the GraphQL.Types.ObjectGraphType class. Create a new file named StoreType.cs in your project and replace the default code with the code given in **Listing 8**.

Create a GraphQL Subscription

You should also create a subscription to enable your GraphQL server to notify all subscribed clients when an event occurs. Create a new class named StoreSubscription and replace the default generated code with the source code given in **Listing 9**.

Configure GraphQL Server in ASP.NET Core

Once you've created the Query type to expose the data you need, you should configure GraphQL Server in the Program.cs file using the following code snippet:

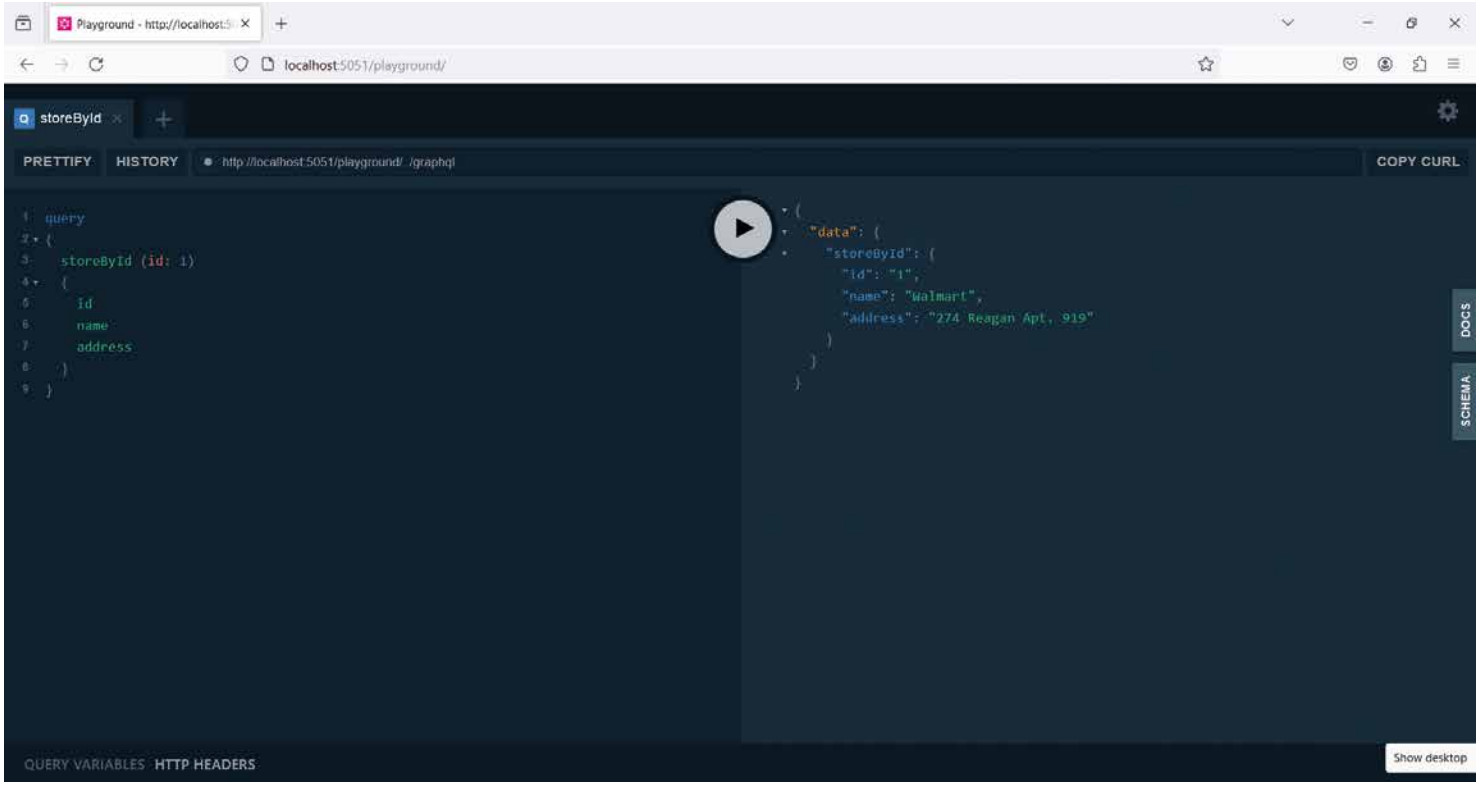

Figure 9: The storeById query in execution!

```
builder.Services.AddGraphQLServer()
    .AddType<StoreType>()
    .AddQueryType<StoreQuery>()
    .AddSubscriptionType<StoreSubscription>()
    .AddInMemorySubscriptions();
```

You can then call the MapGraphQL method to register the middleware:

```
app.MapGraphQL();
```

When you register this middleware, the GraphQL server will be available at /graphql by default. You can also customize the endpoint where the GraphQL server will be hosted by specifying the following code in the Program.cs file:

```
app.MapGraphQL("/graphql/mycustomendpoint");
```

Listing 9: The StoreSubscription class

```
using HotChocolate.Execution;
using HotChocolate.Subscriptions;

namespace GraphQL_Demo
{
    public class StoreSubscription
    {
        [SubscribeAndResolve]
        public async ValueTask<ISourceStream
        <List<Store>>> OnStoreGet([Service]
        ITopicEventReceiver eventReceiver,
           CancellationToken cancellationToken)
        {
            return await eventReceiver.SubscribeAsync
            <List<Store>>
            ("Returned Stores", cancellationToken);
        }
    }
}
```

Listing 10: Configuring GraphQL in the Program.cs file

```
using GraphQL_Demo;
using HotChocolate.AspNetCore;
using HotChocolate.AspNetCore.Playground;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddScoped
<IStoreRepository, StoreRepository>();

builder.Services.AddGraphQLServer()
    .AddType<StoreType>()
    .AddQueryType<StoreQuery>()
    .AddSubscriptionType<StoreSubscription>()
    .AddInMemorySubscriptions();

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseAuthorization();

app.MapControllers();
app.UsePlayground(new PlaygroundOptions
{
    QueryPath = "/graphql",
    Path = "/playground"
});

app.MapGraphQL();
app.Run();
```

Listing 11: The StoreController class

```
using Microsoft.AspNetCore.Mvc;

namespace GraphQL_Demo.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StoreController : ControllerBase
    {
        private IStoreRepository _storeRepository;
        public StoreController
        (IStoreRepository storeRepository)
        {
            _storeRepository = storeRepository;
        }

        [HttpGet("{id}")]
        public async Task<Store> GetStore(int id)
        {
            return await _storeRepository.GetStore(id);
        }

        [HttpGet("GetStores")]
        public async Task<List<Store>> GetStores()
        {
            return await _storeRepository.GetStores();
        }
    }
}
```

Listing 10 shows the complete source of the Program.cs file.

Now execute the application and browse the /playground endpoint. Next, execute the following query:

```
query
{
  storeById (id: 1)
  {
    id
    name
    address
  }
}
```

Figure 9 shows the output on execution of the application.

Create the StoreController Class

Finally, you need to build the controller class to expose the endpoints to the outside world so that they can be consumed by the authenticated clients or consumers of the API. To do this, create a new API Controller in your project named StoreController and write the code given in **Listing 11** in there.

Conclusion

The requirements and constraints of your project will determine the most appropriate choice between SOAP, REST, or GraphQL. SOAP is a good choice for enterprise-level applications because of better support for security and its robust contract definitions. REST can significantly enhance a resource-oriented application because of better performance, enhanced scalability, and simplicity. GraphQL offers a flexible, efficient approach to data retrieval and manipulation, making it an excellent choice for applications that require strong typing, real-time data updates, and efficient data retrieval capabilities.

Joydip Kanjilal

CODE 30 YEARS

May/Jun 2024
Volume 25 Issue 3

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers in This Issue
Kevin Goff, Joydip Kanjilal, Sahil Malik, Paul D. Sheriff, Markus Egger, Julie Lerman, Mike Rousos

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext. 26
tammy@code-magazine.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: Ingram Periodicals, Inc.
International Bonded Couriers (IBC)
Media Solutions
Source Interlink International

Subscriptions

Circulation Manager
Colleen Cade
832-717-4445 ext. 28
ccade@codemag.com

US subscriptions are $29.99 USD for one year. Subscriptions outside the US are $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa and Discover credit cards accepted. Back issues are available. For subscription information, email subscriptions@code-magazine.com or contact customer service at 832-717-4445 ext. 9.

Subscribe online at
www.code-magazine.com

CODE Developer Magazine
EPS Software Corporation / Publishing Division
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379 USA
Phone: 832-717-4445

CUSTOM SOFTWARE DEVELOPMENT

STAFFING TRAINING/MENTORING SECURITY

MORE THAN JUST A MAGAZINE!

Does your development team lack skills or time to complete all your business-critical software projects? CODE Consulting has top-tier developers available with in-depth experience in .NET, web development, desktop development (WPF), Blazor, Azure, mobile apps, IoT and more.

CONTACT US TODAY FOR A COMPLIMENTARY ONE HOUR TECH CONSULTATION. NO STRINGS. NO COMMITMENT. JUST CODE.

codemag.com/code

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

©shutterstock

CODE
STAFFING

UNLOCK STAFFING EXCELLENCE

Top-Notch IT Talent, Contract Flexibility, Happy Teams, and a Commitment to Customer Success Converge with CODE Staffing

Our IT staffing solutions are engineered to drive your business forward while saving you time and money. Say goodbye to excessive overhead costs and lengthy recruitment efforts. With CODE Staffing, you'll benefit from contract flexibility that caters to both project-based and permanent placements. We optimize your workforce strategy, ensuring a perfect fit for every role and helping you achieve continued operational excellence.

Ready to Discuss Your IT Staffing Needs?

Visit our website to find out more about how we are changing the staffing industry.

Website: codestaffing.com

Yair Alan Griver (yag)

Chief Executive Officer

Direct: +1 425 301 1590

Email: yag@codestaffing.com