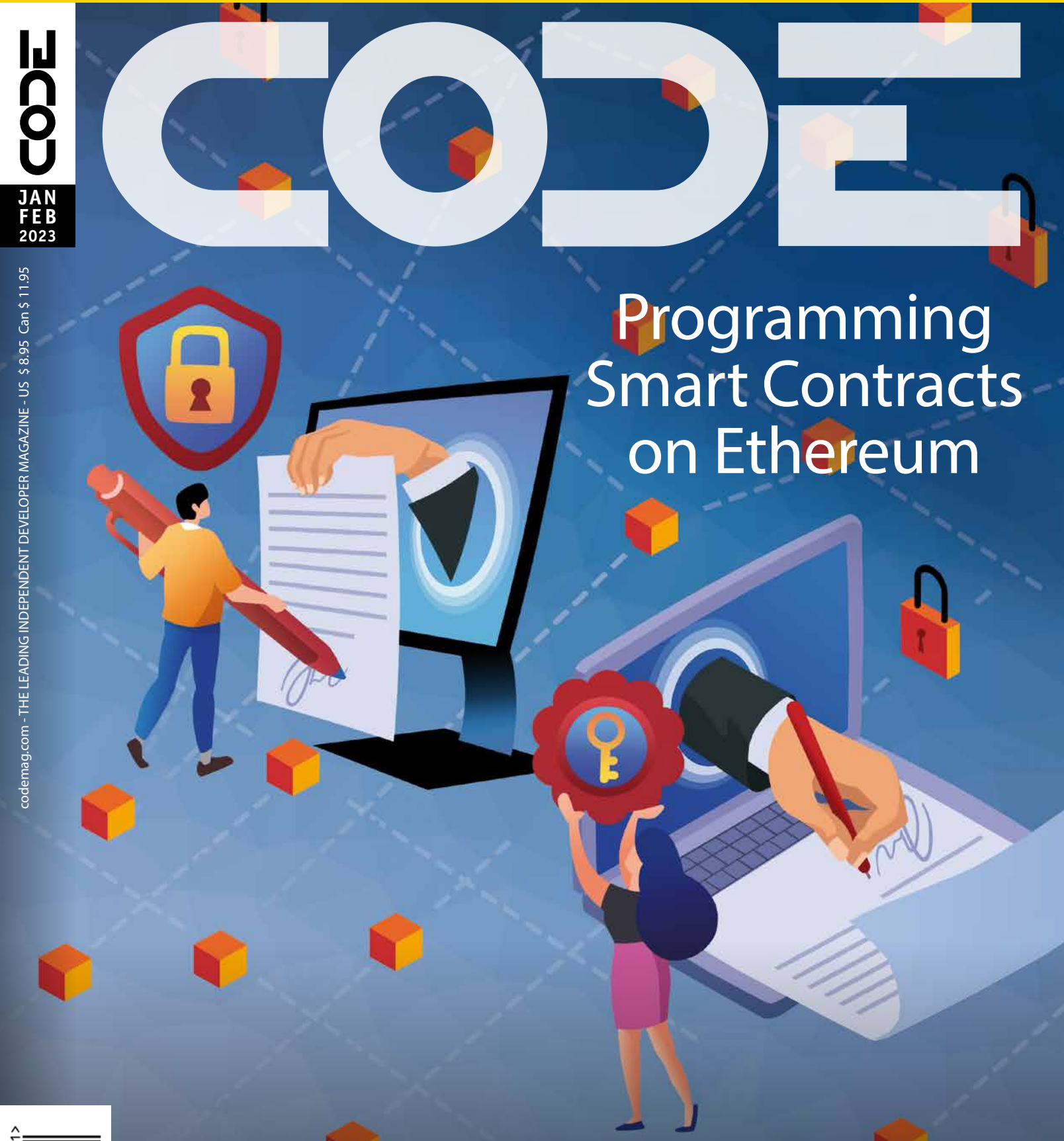
Blockchain, Telemetry, PHP, Data Annotation, Cryptography

CODE

JAN
FEB
2023

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

Programming Smart Contracts on Ethereum



Tracing with Open Telemetry

Understanding Cryptography

The Art of Stepping Away

01>
7 25274 56130 8

DEVintersection Conference

Microsoft Azure + AI Conference

Azure Data Conference

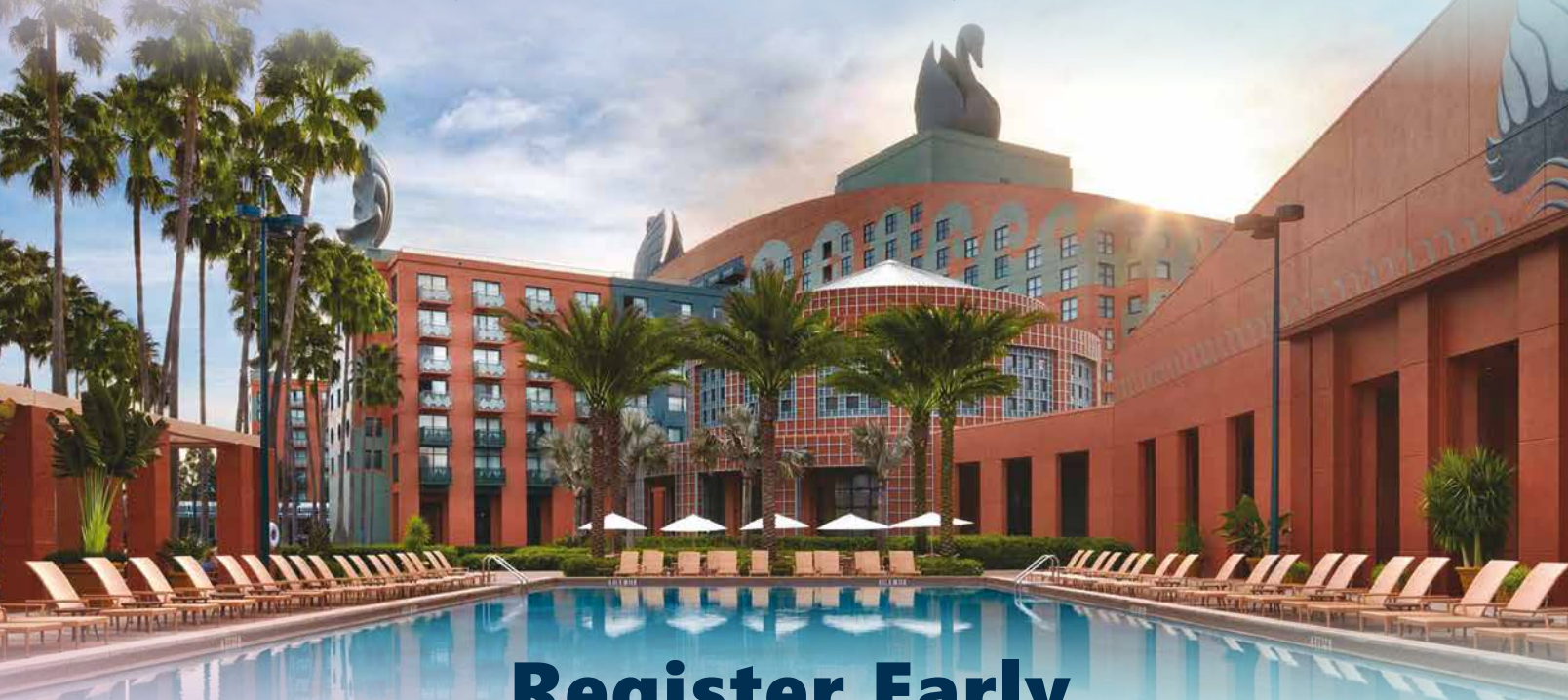
DEC 5-7 2023

WORKSHOPS DEC 3, 4 & 8

DISNEY WORLD

SWAN & DOLPHIN RESORT

AI • Angular • Architecture • ASP.NET • Azure • Azure Data • Azure Databricks • Azure Functions • Azure IoT • Azure SQL• Azure Synapse C# • Blazor • Cloud Security • Codespaces • Cognitive Services • CosmosDB • Dapr • Data Science • DevOps • Docker • GitHub Actions • Machine Learning • Metaverse • Microservices • MySQL • .NET • .NET MAUI • Node.js • PostgreSQL • Power Apps • Power BI • React • Scalable Architectures • Security & Compliance • SQL Server 2022 TypeScript • Virtual Machines • Visual Studio



Register Early

for Optional Workshops

and Receive Your Option of Hardware

DEVintersection.com AzureAIConf.com AzureDataConf.com

Microsoft 365 CONFERENCE

Microsoft Viva
Microsoft Teams
Microsoft SharePoint
Microsoft Power Platform

CO PRODUCED BY MICROSOFT AND M365 CONFERENCE

MAY 2-4, 2023

WORKSHOPS: April 30, May 1 & 5

MGM Grand | Las Vegas

FEATURING:

JEFF TEPER
President – Microsoft Collaborative Apps and Platforms Microsoft

KARUANA GATIMU
Principal Manager, Customer Advocacy Group Microsoft Teams Engineering Microsoft

DAN HOLME
Principal Product Manager Lead for Yammer Microsoft

NAOMI MONEYPENNY
Director, Content Services & Insights Microsoft

CHRIS MCNULTY
Director of Product Marketing, Microsoft 365 Microsoft

LIZ SUNDET
Program Manager Microsoft

HEATHER NEWMAN
Principal PM Manager, Dynamics 365 & Power Platform Community Success Microsoft

APRIL DUNNAM
Power Platform Developer Advocate Microsoft

AND MANY MORE!

REGISTER TODAY!

M365Conf.com

Features

Columns

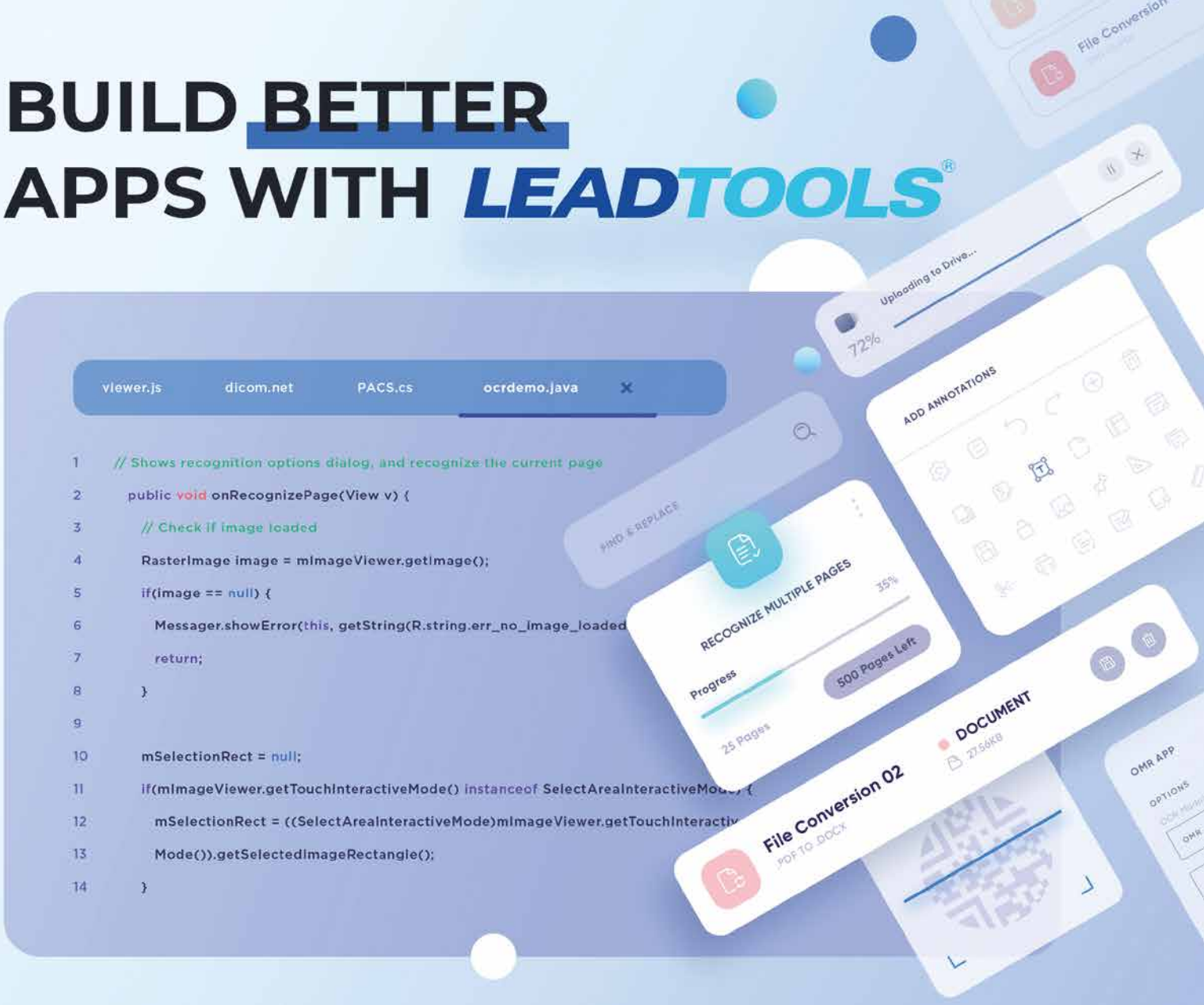
Departments

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.

BUILD BETTER APPS WITH LEADTOOLS®



Powered by patented artifical intelligence and machine-learning algorithms, LEADTOOLS SDKs help you create desktop, web, server, and mobile applications with the most advanced document, recognition, medical, and multimedia technologies, including:

OCR | ID DATA EXTRACTION | DOCUMENT EDITING | BARCODE | PDF

DICOM | OFFICE FORMATS | FORMS RECOGNITION & PROCESSING | VIEWERS | + MANY MORE

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

Take a Break

My community college had a public speaking class as part of their curriculum and, to be honest, this class scared the heck out of me. Standing in front of a group of fellow students caused a deep feeling of dread until I gave my first speech. We were able to present on a topic of our choosing and I chose to talk about the stock market. I was a bit of a finance nerd at the time, so this was in my wheelhouse. I gave the talk and it went as well as it could. I didn't fall flat on my face, nor did I pass out. "Heck," I thought to myself, "maybe I'll survive this class." After that first talk, I gave a few more presentations and what I found is that if I spoke on a topic that I had some expertise in, the talks generally went well.

After finishing this class, it would be around five years before I'd speak in public again. This time, the stakes would be higher as I was about to be a professional speaker/trainer.

My last W2 job was in the early 90s at a company called Pinnacle Publishing. The company ended up having some hard times and I was laid off. Luckily for me, I'd made some great contacts/friends at Pinnacle and was able to use my network to find a few opportunities. One of them was to become a FoxPro for Windows instructor.

It was going to be sink or swim on the whole training gig. I prepared for this job two ways. The first was by attending a class given by one of the top-rated instructors, David Anderson. I loved his style of presenting and adopted it as my own (great artists steal, am I right?). The second step was to get in front of a crowd and speak. I did this by asking the leader of the local FoxPro user group if I might be able to speak. "Yessir, you can speak at the next group meeting," was his response. I gave a talk on something called DDE (Dynamic Data Exchange). I'd become a bit of an expert in this area so I was comfortable speaking on it. The user group went well and I was off to the races.

That was Hartford, Connecticut, which would be the location my first training class, and luckily for me, it went well. My beginner class had—drum roll please—two whole attendees. The intermediate and advanced class added five more people for a grand total of seven people. I consider myself fortunate that it was a low number, as larger classes have their own unique challenges.

Long story short, my first class went well and I signed up for more. I did 11 weeks of training in the first year. I probably gave another 60+ weeks of training over the next five or so years. I'd caught the speaking bug, which has, over the years, afforded me the opportunity to speak to literally thousands of people.

My public speaking wasn't limited to teaching. I continued my love of speaking at user groups and was also lucky enough to speak at numerous technology conferences. These conferences ranged from small regional conferences to epic conferences hosted by companies like Microsoft. These conferences have taken me to many cities in North America and Europe. I feel I've lived a charmed life and am lucky to have been afforded these opportunities.

What I found, though, is that after doing this for 20 years non-stop, it was no longer fun and was starting to wear on me. I felt like it may be time to retire from public speaking.

I'll tell you why that happened. I became isolated when going to conferences. I'd show up, do my talk, and retreat to my hotel room. It's easy to do this, as there was always the excuse, "I need to work." This was the first sign that I needed a break. I'd become jaded and didn't really like interacting with attendees and other speakers. And, to be honest, it's the interactions that are 100% of the value of these shows.

The second thing is that I found myself "phoning it in." I'd create a talk and become bored with it after three, four, or five deliveries. Finally, I was using conference speaking as a score card to keep my MVP status with Microsoft. I was on a hamster wheel. So, what happened? I gave up my MVP and stopped speaking entirely. I was done and, to be honest, it felt good to be done.

It was a tweet that brought me out of "retirement." My long-time friend D'arcy Lussier runs a conference called Prairie Dev Con and had just opened the call for speakers. This tweet stirred something in me and, after careful consideration, I decided to submit a talk or two. The bug returned! I had a desire to speak again and thought I had some fresh ideas to contribute. I submitted topics on SQL Server stored procedures and Elasticsearch, two technologies I was familiar with and thought I had new ideas to share. I was selected to speak and haven't turned back since. I gave talks in 2018 and 2019 and was on schedule to speak in 2020. As a matter of fact, I planned to give my first keynote(s).

Boom! PANDEMIC.

In my last column, I gave a short preview of my keynote for the Regina and Winnipeg legs of PDC in 2022, and I can tell you that I had a real blast delivering the keynote at both shows. But let me tell you what I really learned this year.

I really missed speaking at developer conferences and I especially missed interacting with attendees and speakers. This was an unforeseen set of rewards. I was able to reconnect with old friends as well as make a number of new ones. I hadn't felt this way since my early days on the speaking circuit.

What led to this change of heart? I believe it can be attributed to the old parable: "separation makes the heart grow fonder." I attribute these new feelings to the fact that I needed these connections more than I knew, and that the time off unexpectedly brought that back these feelings.

Sometimes taking a break is a good thing, just what you need to refresh and rejuvenate.

Rod Paddock
CODE

Cryptography

What do ancient Greeks from 1900 B.C. and blockchain have in common? Both have a keen interest in cryptography. The desire to send messages secretly across hostile environments, so that only the intended recipient could read and trust the message, has been around for some time. Today we understand cryptography to have four high-level goals.

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.

- **Confidentiality.** The information must not be accessible to anyone who isn't the intended audience of the message.
- **Integrity.** The information being transmitted must reach the recipient without alteration.
- **Non-repudiation.** The sender of the information is reliable and can be traced. The sender cannot deny sending the information.
- **Authentication.** The sender and receiver are confident of each other's identity.

Sounds simple enough, right? It's really interesting how cryptography has evolved through the ages. In this article, I'll talk about an interesting historical context around cryptography and end up with how cryptography works in the modern world. I think you'll be amazed at how far we've come, and yet we're still solving 6000-year-old problems.

Ancient Greece

A very long time ago, around 500 B.C., there was a Persian prince called Darius. He considered another gentleman, called Histiaeus, to be loyal and asked him to come to Susa as an advisor. Now, as it turned out, Histiaeus was quite unhappy in Susa and he felt he was almost a prisoner there. He wanted to escape and he had faith in his son-in-law Aristagoras to help him. Unfortunately, this was 500 B.C. and there was no Twitter or Facebook back then. Being able to securely communicate across a hostile environment was impossible. Or was it?

As it turns out, Histiaeus had a loyal slave. Histiaeus shaved the head of the slave and tattooed a message on his head, instructing Aristagoras to revolt against the Persians. Histiaeus then waited for the slave's hair to grow back, and the slave then travelled to Aristagoras. The message was securely delivered.

What does all this have to do with encryption and cryptography? Histiaeus wanted to send an encrypted message to Aristagoras and he used a technique called steganography, which is the practice of concealing a message in another perfectly commonplace object. Histiaeus used his slave's head, but you can use an image or a computer file to enclose your secure message.

You may think this is far-fetched, but it isn't. Have you ever written a message in an invisible ink?

In World War II, Ms. Velvalee Dickinson, a spy for Japan, was a dealer in dolls and frequently used them to send information about ship movements.

In Vietnam, prisoner of war Jeremiah Denton blinked his eyes in Morse code to spell out T-O-R-T-U-R-E.

Tesla once caught a leaker by sending identical emails with minor differences, creating a digital signature to identify someone who leaked information. Check out this more recent tweet (https://twitter.com/elonmusk/status/1579101966453858305) by Elon Musk, another version of the same catch.

> That is quite an interesting story. We sent what appeared to be identical emails to all, but each was actually coded with either one or two spaces between sentences, forming a binary signature that identified the leaker.
>
> *Elon Musk*

Now, as impressive as Histiaeus's escapades were, let's be honest. Finding a slave in today's world and shaving his head is not only inefficient, but also probably illegal. Not to mention that it would make the slave quite mad.

The Roman Empire

Sometimes I wonder what it must have been like to live in the Roman Empire. The Roman Empire stretched over many cultures, a vast geographic expanse. Much like today, in times of excess, not everyone was happy. Even back then, there were those trying to exchange messages who feared being caught. So, they invented something called the Caesar cipher. At a high level, it was quite simple. If I wanted to send you a message, for example, we'd agree ahead of time that I'd shift the letters of the alphabet by a fixed number. Let's say we agree to shift everything by three. So, the alphabet ABCDEFGHIJKLMNOPQRSTUVWXYZ now becomes XYZABCDEFGHIJKLMNOPQRSTUVW. The message "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG" now becomes "QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD".

if someone was caught sending a secret message, the guards would think it was written in some foreign language. And they'd just let the person go. ORO. Can you decode that? You can use this tool (https://www.boxentriq.com/code-breaking/caesar-cipher) to help you out.

Ancient Iraq

As smart as the Romans were, there was a geek in ancient Iraq in the 8th century A.D. called Al-Kindi. This guy was a mathematician. When all his friends went partying on Fri-

day evening, he'd sit on his, well, I want to say computer, but you get the idea. He discovered an interesting thing called "frequency analysis."

Frequency analysis is based on the fact that in any given stretch of written language, certain letters and combinations of letters occur with varying frequencies. Moreover, there is a characteristic distribution of letters that's roughly the same for almost all samples of that language.

Did you know that the letter "e" is the most commonly used character in the English and Spanish languages? You can differentiate English and Spanish because the letter A occurs way more often in Spanish than in English. You can see the frequency analysis of English in **Figure 1** and the frequency analysis of Spanish in **Figure 2**.

This created a real problem for the Caesar cipher. If I knew that the shady guy with the weird message was English, I could just examine his message and replace the most common alphabet with "E" and guess the shift in the cipher. In fact, with modern computers, I could do this super-fast for all known languages. Once patterns start to emerge, I could easily decipher the message.

Here's a funny story. In World War II, Germans always ended their messages with Heil Hitler! This really helped the allies crack their secret messages. More on that shortly.

To get around this problem, a better version of Caesar cipher was created called the polyalphabetic cipher. The idea was that a secret word was exchanged ahead of time. The shift would be done per the letters of that secret word.

Let's say the secret word was "sahil", and the numbers this corresponds to are, 18 (S), 0 (A), 7 (H), 8 (I), and 11 (L). Now instead of shifting everything with "3" or a fixed number, we shift using this sequence of 18, 0, 7, 8, 1. So my message of "thisisfine" becomes LHPATKFPVP. The first letter (T) gets shifted by 18, the second letter (H) gets shifted by 0 and so on so forth. When you get to the sixth letter, where the secret word has run out of letters, you just repeat the order of the letters in the secret word. So the S of "is" shifts by 18, followed by 0 and so on. The same letter is going to represented in several different ways as you work through the message.

The clear advantage here is that the distribution of alphabets in frequency analysis becomes much flatter. In fact, for an infinite-length message with an infinite-length key, the frequency analysis is absolutely flat. In order to break this cipher, you'd need to guess the length of the key, and try to shift the length and match with the standard English frequency pattern.

That's much harder, right? But it's still not impossible for a modern computer to break.

This technique was further improved with a one-time pad. In this technique, plaintext is paired with a random, secret key (or pad). Then, each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad. The key is as long as the text itself.

Let's say the pad was five digits long. Using "Sahil," the total number of possibilities here is 26*26*26*26*26 = 11881376. This is a huge number. How huge? A typical sheet of paper is 0.002" thick. A stack of 11881376 sheets would be 700 meters high (almost half a mile).

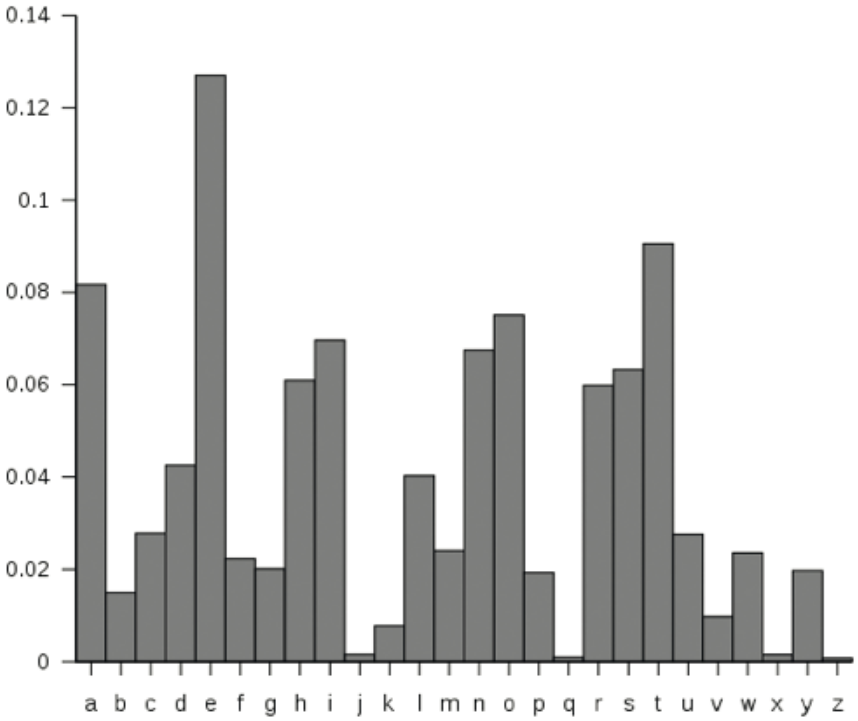


Figure 1: English frequency analysis

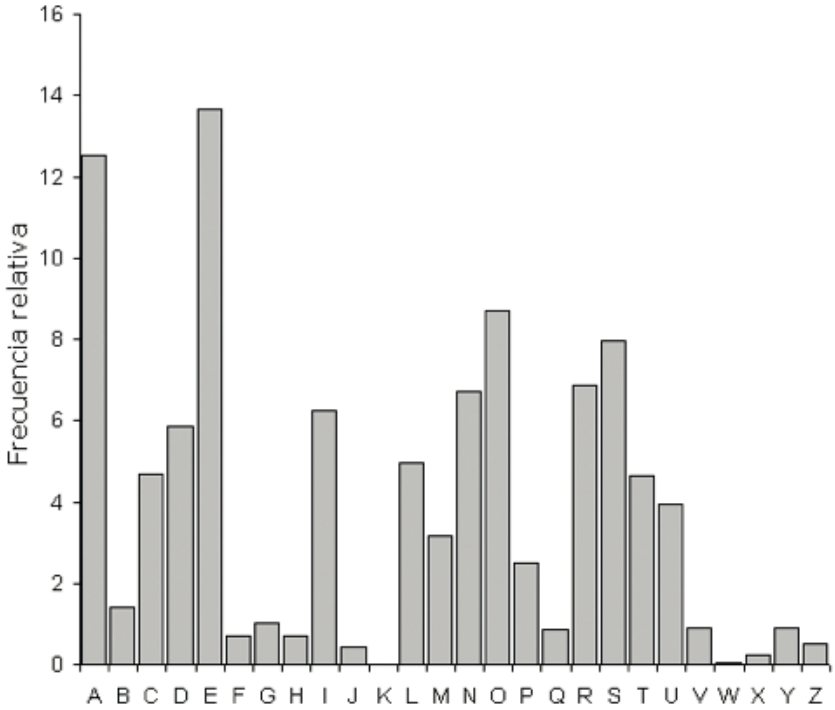


Figure 2: Spanish frequency analysis

Did you know that in the 1960s, KGB agents used magnesium paper with one-time pads. This was a tiny notebook, as shown here (http://www.ranum.com/security/computer_security/papers/otp-faq/) that the Russians

exchanged ahead of time. If someone was busted, they could just burn the notebook and it burned quickly and left no residue. Remember, indoor smoking was normal back then. German U-boats used a similar technique, where the ink dissolved in water. Or sometimes the pad was so small that they'd have a booklet of pages, and when they used a pad, they'd eat that sheet of paper.

So you see, if you could have a one-time pad exchanged ahead of time with a sufficiently large cipher, you could create perfect encryption. Encryption so good that even modern computers couldn't crack it.

There's one problem though. How do you exchange that cipher or one time pad securely ahead of time? If that gets compromised, everything is compromised. I'll get to that in a moment, but first let me tell you another interesting story from World War II.

World War II and the Enigma Machine

World War II was awful. But it was also a triumph of human ingenuity, a story of good over evil. A lot of interesting inventions came out of the duress of need. Nazis took the polyalphabetic cipher approach to create a device called the Enigma machine. It was the size of an oversized briefcase and it needed a battery to run. It had three rotors inside it and what looked like a typewriter in front of it. The idea was that every time you pressed a key on the typewriter keypad area, a corresponding alphabet lit up that was your encoded version. In addition, every time you'd press the alphabets, the rotors moved, effectively changing the cipher. This meant that pressing E multiple times yielded a different outcome each time. This made cracking the cipher a lot harder because it was constantly changing.

There were, in fact, two versions of the Enigma machine. The civilian version that the banks used had those three rotors and that's it. But the military version had a switchboard in front, which further added numerous permutations and combinations; the idea was that the sender and receiver exchanged patterns of switchboards ahead of time. These were printed sheets of paper that covered about the next month ahead, and they were exchanged before, say, a U boat went out into the ocean.

The sender and recipient had the same codes, and they ensured that they used the same pattern on the switchboard every day. They could safely communicate with each other, but as this pattern changed every day, it was impossible to crack. In fact, those codes were written using water-soluble ink, so even if you got hold of the machine, without the codes, it was useless.

Very ingenious. But the allies had an interesting trick up their sleeve. See, the Germans didn't realize that they had unknowingly baked a weakness into the Enigma machine. Additionally, they weren't following the best security practices.

The Germans had engineered a solution so that a letter never yielded the letter itself in the cipher version. This ended up being a weakness of the product. The Germans started the day with the weather report, titled, duly, "Wetterbericht." And they would, of course, sign off with "Heil Hitler." All the allies had to do was align the resulting alphabets with the cipher alphabets to know which letters from the unencrypted version matched which letters in the encrypted version. Then it was a matter of brute-forcing combinations using a huge noisy analog computer they had built to decipher the code. Additionally, over time, the Germans got lazy, and didn't bother to change the code every day. It was just so much hassle really. And that made the allies' job a lot easier.

What do we learn from this? First, don't reinvent security. Modern algorithms are open but the keys are secure. Second. don't re-engineer the algorithms, as they are time-tested. Third don't get lazy and reuse the same password everywhere.

A Common Problem

After World War II, the world changed rapidly. We already had algorithms that offered perfect security under perfect circumstances. As long as you could exchange codes ahead of time and followed all the rules, you could effectively create an encryption mechanism that was undefeatable. This worked fine for banks because you could exchange the keys securely ahead of time.

But what if you didn't know which party you were communicating with ahead of time? For instance, when you open a browser, and visit https anything, how is the connection securely negotiated on the fly without you first having to visit the website and store a key there? Hang on. I've got another story to tell.

World War II ended with the hope of rebuilding a prosperous and peaceful future. But then the cold war started. Both the U.S. and Russia had thousands of missiles tipped with very powerful nuclear bombs. If an ICBM launches from Russia, it has approximately 13 minutes before it lands in the United States. Faster, if the missile is launched from a submarine. Today's hypersonic missiles cut that time by a fifth or more.

A network of radar stations and satellites were built that had to coordinate automatically and securely to judge the trajectory of such a missile and even to confirm that such a missile was indeed an ICBM. There was no space for a mistake here, because the reply to such an attack would be the end of the planet. Conversely, when such satellites and radar stations communicate with each other, they need to have confidence that they are communicating securely, that their messages can't be snooped or tampered with, and that indeed, whoever they are communicating with is who they think they are communicating with. You know, all the stuff cryptography deals with.

How do we securely exchange a key ahead of time among so many parties in the 13 minutes it takes for the missile to land in my backyard? We can't.

We needed a better mechanism, something that lets us communicate securely, without having to exchange a key ahead of time.

The Solution: Symmetric Key Exchange.

The solution lies in one-way functions. If I ask you what is 2 multiplied by 5 multiplied by 7 multiplied by 13, with some difficulty, you'd be able to tell me the answer

is 910. If I asked you what prime numbers I need to multiply to get 912, it would probably take you longer. This is a great example that traversing from A->B is easier than B->A. This is an example of a one-way function. My example was quite simple, but some very smart scientists have come up with algorithms that are much better at this one-way problem. The best one-way problem I could come up with was climbing up a ladder. It's always easier to climb up rather than down, unless it's involuntary.

Let's take the problem we're trying to solve. Joe and Jack wish to exchange a secret, but Joe and Jack have never met each other. Clara, in the middle, is a hacker who can hear every communication between Joe and Jack. How do Joe and Jack quickly and securely exchange a key in such a way that even if Clara listens to every message, Clara cannot get hold of that key, that shared secret Joe and Jack can use to communicate further.

For simplicity, let's assume that the one way algorithm is colors. If I mix yellow and blue, I get green. But if I gave you a bucket of green, and asked you to separate it into yellow and blue, it's a lot harder.

First, Joe and Jack agree on a color and they send it to each other. Clara is listening to every message so she also receives the color. Let's say the color is yellow. This is the "public" color.

Next, the two parties, Joe and Jack, independent of each other, come up with a private color. Let's say Joe's color is red, and Jack's color is blue. They mix those two colors independent of each other and get a resultant color. Now Joe's resultant color is orange and Jack's resultant color is green. These new mixtures (orange and green) are now communicated with each other.

At this point, Clara, the hacker, has yellow, orange, and green. Clara thinks she is very smart, but what comes next may shock you.

Joe and Jack receive each other's mixed colors, so Joe receives green from Jack and Jack receives orange from Joe. But next, both Joe and Jack, mix their private colors with the mixed color they received from the other party. Now the resultant mix is a reddish brown.

The key here now, is that the reddish brown was never sent over the wire. Clara has no way of recreating the reddish brown. But somehow, both Joe and Jack, have access to reddish brown. This reddish brown can now serve as the shared key.

I've explained this with colors, but in the real world, we use numbers. This is the foundation of RSA encryption, which forms the fundamental building block for many kinds of encryption algorithms in use today.

The technical word for what I just explained is "symmetric key exchange."

The Problem with the Solution: Asymmetric Key Exchange

Symmetric key exchange is impressive. It works, it's secure, but it has a huge shortcoming. It requires the management of too many keys. Symmetric key exchange works fine if you have two parties communicating with each other. But what if you're a website with many transactions and many people inputting data, such as a bank. And you have millions of browsers accessing you at the same time. Are you supposed to somehow securely manage the keys of all these millions of browsers? That's millions of keys you'd have to manage.

Have you ever seen a padlock that you can lock by just pressing it down but to unlock the padlock you need a key?

This the foundation of our solution. In computer algorithms, you have an equivalent of this padlock. You have public private keypairs, typically a certificate. The idea is that anyone can close the lock with the public key, but to unlock the lock, i.e., decrypt the message, you need the private key.

So the bank now has a public-private keypair. The private key, the key to the padlock, never leaves the bank. But anyone who wishes to send a message securely to the bank can request the public key. Now the sender can encrypt a message with the public key, i.e., lock it, and send back the locked padlock, i.e., the encrypted message, back to the bank. The bank can now unlock the message with the private key that was never sent over the wire.

Hybrid Solutions

It looks like asymmetric key exchange offers the best solution here, but not so fast. The first problem with asymmetric key exchange is that it's computationally expensive. The second problem is that you have to keep that private key secure. And if such a key leaks, you'd have no idea it has leaked.

There are other issues as well, such as brute-force attacks, but most modern certificates use a key long enough to make such attacks impractical. There is, of course, the risk of man-in-the-middle attack, an intermediary who simply swaps out public keys and pretends to be the recipient (the bank in the example) and forwards messages while reading them. This attack is also mitigated via mechanisms that verify the authenticity of the sender and receiver. I'll talk about this shortly when I talk about TLS.

Most issues are mitigated except keeping the private key secure and the computation expense problem.

The computation expense problem can be mitigated by exchanging a shared key for a symmetric key algorithm using asymmetric key exchange. This way, you pay higher up-front computation costs for only the initial handshake or whenever you renegotiate the key. Subsequent communication occurs using symmetric key exchange. As I'll explain later, the computation expense problem can also be mitigated by using modern algorithms and specialized hardware.

As far as private keys getting compromised, well, that's up to you to keep them secure. But if they do get compromised, there's a mechanism via which a central certificate authority can revoke such an issued cert. This is called a CRL or certificate revocation list that the certificate-issuing authority must maintain. You can imagine that, over time, this CRL list gets huge and unwieldy, espe-

cially when it needs to be reliably communicated over the internet. This is why certificates have a validity: They automatically expire after a certain duration. CRLs greater than a certain age don't need to be maintained.

There are also certificate trust chains, using which you can revoke a group of certificates easily, but I'm getting ahead of myself.

This does make me scratch my head a bit. Imagine a soldier in Afghanistan who needs to swipe a smartcard to gain access to a secure resource, perhaps something that launches a deadly weapon. In the middle of a war zone, with very poor connectivity over satellite, how can you be sure that the CRL at the edge location is updated and trustable? Imagine you're under enemy fire and not being able to respond because IT can't find their head in a bucket.

The real world is messy! In a perfect world, you can guarantee perfectly secure solutions. But in the imperfect world we live in, you always end up taking shortcuts for convenience and practicality. It's these shortcuts that bite us in the long run.

Figure 3: Google's certificate

How do you check for a CRL in a submarine? How do you verify the authenticity of a message to fire nukes at an enemy in a submarine? Did you know that Lieutenant Colonel Stanislav Yevgrafovich Petrov was faced with a similar dilemma in 1983 when the nuclear early warning radar of the Soviet Union reported the launch of an ICBM with four missiles behind it coming from the United States. Deep inside a submarine, Lieutenant Colonel Stanislav Yevgrafovich Petrov had to decide whether this message was false and risk the annihilation of the USSR, or decide that this message was true, respond, and risk the destruction of the planet. Out of pure gut feelings, he decided this was a false alarm and effectively saved the world. Unfortunately, for this, he faced intense questioning by his superiors and was reprimanded.

> It's the shortcuts that bite us.

It's really worrisome how many times since 1945 the world has come close to catastrophe, and how national leaders treat this with such nonchalance. It's frankly a miracle that we've made it this far without an accident we weren't able to manage.

In my daily mundane life, I really wonder sometimes, sitting in meetings where we make security decisions driven by budget, convenience, and adoption, if we are making the right decision.

Anyway, I digress. Let's get back to hybrid solutions, a great example of which is TLS, something you use every day.

TLS

TLS stands for transport layer security. It's a protocol designed to provide communications security over a computer network. You probably use it every day when you check your email, when you open your browser, when you IM anyone. All of them use TLS. TLS guarantees the four tenets of cryptography, confidentiality, non-repudiation, integrity, and authenticity.

At the heart of TLS and many other things is PKI, or the public key infrastructure. PKI is a mechanism to create, manage, distribute, store, and revoke digital certificates across any network. For instance, on the internet, we have a PKI infrastructure. Your local company's network may choose to set up its own PKI and CA (certificate authority).

Typically, when your browser visits www.google.com, how can it trust that it's the real Google? Google's web server presents its certificate to the browser. And the browser must be able to trust that certificate. How can your browser trust Google's certificate? The idea here is that Google's certificate is issued by some authority that your computer already trusts. And that certificate authority is backed by its own certificate. It's possible that the certifying authority isn't trusted directly by your computer, but your computer trusts another authority that backs this certifying authority. This chain of trust can go a few

levels deep, until eventually the site (Google's) cert is backed by some authority you already trust.

Browsers typically show this information to you, as can be seen in **Figure 3**. This is a screenshot from Chrome on MacOS, but every OS and every modern browser has an equivalent of this.

If you explore the certificate a bit deeper, you'll see that it's backed by a certification path, as shown in **Figure 4**.

Google has chosen to create its own internet authority, which may not be directly trusted by your computer. But it's backed by GTS CA 1C3, which is backed by GRS Root R1, which is trusted by your computer. Therefore, you can now trust Google's identity.

In the certificate, there are a lot of details. For instance, you can see that the certificate signature algorithm is PKCS #1 SHA-256 with RSA encryption. If you explore the fields further, you'll find a lot of other interesting things, such as that this certificate can be used from a certain date and time until a certain date and time, that it has a thumbprint, that it can be used for proving identity and encryption, and many other interesting details.

How Does This Really Work?

It all begins with the TLS handshake.

The client (browser), uses HTTPS to get to a location. The first message is a hello from the client, which contains details like the maximum TLS version the client can support, a random number, and a list of cipher suites they support. As of writing this article, TLS 1.2 is the most common version being used, TLS 1.1 or lower are no longer recommended, and TLS 1.3 is an improvement over TLS 1.3.

At this point, the server can look at the details and accept or refuse to communicate with the client. For instance, if the client insists on communicating with TLS 1.1 and the server deems it insecure, the server can refuse to communicate further. But let's say that the server and client agree to use TLS 1.2 and a cipher suite they agree on.

Now the server responds with a message to the client informing the client about which TLS version the server will use, and pick one of the cipher suites and a random number.

Now the server sends a server key exchange message, which includes the public part of the cert. The digital signature is also included, which is again a hash of the previous messages, and signed using the private key of the server cert. Using the public key, the client can verify the digital signature. This is the proof that the server is who they say they are.

It's worth adding that the server can also optionally request a certificate from the client at this point, which can prove the identity of the client to the server. This is common in certificate-based authentication.

Now the client replies with the client key exchange; remember this was the color example I gave earlier. This ends up resulting in a shared key secret.

Figure 4: Google's certification path and signature algorithm

Next, the client sends a change cipher message, which effectively means that I have the shared key, so let's start encrypting. Using that shared key and the agreed upon cipher, the data is interchanged in an encrypted fashion.

Encryption Algorithms

As is evident from this article, security continues to evolve. It's worth getting familiar with some common encryption algorithms and their pros and cons.

As far as symmetric encryption algorithms go, there are three you need to know about: DES, 3DES, and AES.

The first is DES symmetric encryption algorithm. This is one of the most ancient encryption algorithms, introduced in 1976. It uses a 56-bit encryption key. DES is no longer considered secure and can be easily cracked using modern computers. A better replacement is either AES or Triple DES.

3DES or Triple DES is a better version of DES. 3DES applies the DES algorithm thrice to each data block. As a result, this process makes 3DES much harder to crack than

its DES predecessor. Even so, 3DES, although better than DES, is also not considered secure. In fact, TLS 1.3 explicitly excludes the use of 3DES.

AES symmetric encryption algorithm is probably one of the most commonly used algorithms today. Unlike DES, AES is a family of block ciphers that consists of ciphers of different key lengths and block size. This makes it both safer and faster than DES. Today, you'll see AES in use in many places, such as WiFi networks, VPN, SSL, and so many more.

As far as asymmetric encryption algorithms go, the two most popular algorithms in use today are RSA and ECC.

RSA was invented by three smart people: Ron Rivest, Adi Shamir, and Leonard Adleman. It uses prime factorization as its basis. This method involves two huge random prime numbers and these numbers are multiplied to create another giant number. The puzzle here is to determine the original prime numbers from this giant-sized multiplied number. This is a very hard problem to solve. To give you an idea, to crack an RSA-768 bit key, it would take 1500 years of computing time for all computers that exist on the planet today. The common standard is RSA-2048, which would take 300 trillion years. RSA-4096 would take longer than the entire age of the universe. The higher the bits, the more CPU you will consume, but the security gets exponentially better. Maybe some brilliant scientist one day will come up with an ingenious algorithm to crack RSA, but it's been around since 1977 and hasn't been cracked yet. It's postulated that quantum computers will end up cracking RSA in another 20-30 years. Maybe some governments already have secret super powerful computers that can crack RSA in extreme circumstances, but it's safe to say that even with a quantum computer, which, practically speaking, doesn't exist, it takes a long enough time to crack RSA. A lot of governments, therefore, force tech companies to give up their private keys as a precondition to operating in their country. The reality is, though, for all practical purposes, that most governments cannot crack open sufficiently secured communication. Whether that's a good thing or a bad thing is a topic I'd rather not get into.

Another popular asymmetric encryption algorithm is the ECC or elliptic curve cryptography algorithm. ECC is a recent development, and more and more systems are using it. It's particularly popular in the Apple world. In ECC, a number symbolizing a point on the curve is multiplied by another number and gives another point on the curve. To crack this puzzle, you must figure out the new point on the curve. The mathematics of ECC is built in such a way that it's virtually impossible to find out the new point, even if you know the original point. The key advantage of ECC over RSA is that it can offer a similar level of protection as RSA but with a much shorter key. You can increase the key length to gain even better security than RSA, but ECC's ability to work with shorter keys means that it can offer you the same protection as RSA with much less computational power. With the advent of mobile devices, you can imagine why ECC has been so popular. Also, this means that websites load quicker because the initial key exchange can be done quicker.

Isn't the world we live in amazing? Just 10 years ago, we were peddling Internet Explorer 6 with DES, which was totally insecure and slow as a Lada full of elephants going uphill. Now we have dedicated chipsets taking full advantage of secure and faster algorithms on much higher-speed networks.

Then there's the real world, where so many very large organizations continue to make mind-boggling decisions about security. And so many world leaders talk about nuclear weapons like they understand what they're dealing with. Please send them a copy of this article if you can.

Summary

When I was at the helm of picking careers for my life, I chose to be a computer engineer. My main reason was that this field will never be boring, there's so much to learn, and people use fun words like bugs to describe problems. As the field and I have matured, this has been a constant: This field of work has continued to evolve at a furious pace.

Let me ask you this today: What kind of airplane will we be flying in the year 2092? You probably think of something super futuristic. What kind of car will we driving, what kind of houses will we live in, etc. But let's go back 70 years into history. The Boeing 747 flew in 1952. And 70 years later, we're squished into the middle seat, be-

ing treated like dirt, in basically the same technology. The slight improvements we have seen are all because of technological optimizations, but no quantum leaps have been made.

In fact, I'm hard pressed to think of any field of engineering where we've made a quantum leap besides IT. And all the improvements other fields have made are on the back of IT. Look at medicine. The MRI machine is pretty amazing, but at the end of the day, it's a fancy computer or was designed by fancy computers. Look at rockets. They're hyper-optimized using computers.

Now think of the change the tech sector has seen in the past 10 years. What will the next five or 10 years bring? As long as we treat the planet responsibly, this is an incredible time to be alive.

I'm a curious person with many interests. No technology exists in vacuum and cryptography has invented and reinvented itself as a result of circumstances that the human race has been through. I thought it was fun to talk a bit about history, and relate the stories to the real-world. This was an incredibly fun article to write and it's a topic I think a lot about. What do you think? Did you enjoy reading all these stories and their effect on the field of cryptography? Let me know.

Until next time, kdssb frglqj! (This is a Caesar cipher with a shift of three. See if you can decode it).

Sahil Malik
CODE

The Rich Set of Data Annotation and Validation Attributes in .NET

Data annotations are not only for use in ASP.NET web applications. Any type of .NET application can use data annotations for validating data. It only takes about 10 lines of code to programmatically validate data annotations attached to entity classes. There are many built-in data annotations supplied by Microsoft that can validate your data quickly and it's easy to create your own data annotation attributes to apply to your entity classes. If you have some very specific validation needs, you may implement the IValidatableObject interface for your entity classes. If you're developing multilingual applications, you can even move your error messages into resources and specify the name of those resources on each of your attributes.

Paul D. Sheriff
www.pdsa.com

Paul has been in the IT industry over 35 years. In that time, he has successfully assisted hundreds of company's 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 many courses in the www.pluralsight.com library (http://www.pluralsight.com/author/paul-sheriff) on topics ranging from .NET 6, LINQ, JavaScript, Angular, MVC, WPF, ADO.NET, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

In this article, you're going to explore most of the data annotations supplied by Microsoft. You're going to develop a few custom validation attributes to check dates and numeric values. You'll also learn to create a custom validation attribute to compare the values between two different properties. You're also going to see how to implement the IValidatableObject interface to tackle more complicated validation scenarios. Finally, you'll set up a couple of resource files and see how easy it is to localize your error messages.

Traditional Validation Methods

In the distant past, to validate the data a user inputs into a form would be done directly in the code-behind the form. The appropriate messages were displayed on the input form to tell the user what they did wrong. As object-oriented programming (OOP) became the norm, developers moved that input data into properties of a class and wrote a Validate() method to perform the validation. A collection of messages is returned from the Validate() method and those messages were bound to the input form to be displayed. Let's first look at the traditional way of validating data before we move onto using data annotations.

Create a Console Application

To follow along with this article, open Visual Studio and create a console application with the name **Samples**. You may use either .NET 6 or .NET 7 for the samples in this article. Most of the code will work just as well in earlier versions of .NET too. Once you have the application created, right mouse-click on the project and add a new folder named **EntityClasses**. Right mouse-click on the EntityClasses folder and add a class named **Product**, as shown in **Listing 1**, to this project.

There are several properties in the Product class that should be validated, such as making sure the **Name** property is filled in and that it has 50 characters or fewer in it. You might also verify that the value in the **ListPrice** property is greater than the value in the **StandardCost** property. You should also ensure that the value in the **SellStartDate** property is less than the value in the **SellEndDate** property.

To report error messages to the user, you need a class to hold the property name in error, and the error message to display to the user. Right mouse-click on the project and add a new folder named **ValidationClasses**. Right mouse-click on the ValidationClasses folder and add a new class named **ValidationMessage**. This class is shown in the code snippet below.

```
#nullable disable

namespace Samples;

public class ValidationMessage
{
  public string PropertyName { get; set; }

  public string ErrorMessage { get; set; }

  public override string ToString() {
    if (string.IsNullOrEmpty(PropertyName)) {
      return $"{ErrorMessage}";
    }
    else {
      return $"{ErrorMessage} ({PropertyName})";
    }
  }
}
```

Create a Product View Model Class

If you've been doing MVC or WPF programming for a while, you quickly learned that using a Model-View-View-Model (MVVM) design pattern makes your coding easier and more reusable. Let's create a view model class to encapsulate the Product class. Right mouse-click on the project and add a new folder named **ViewModelClasses**. Right mouse-click on the ViewModelClasses folder and add a new class named **ProductViewModel** as shown in the code below. Create a public property named **Entity** that's of the type Product. In the constructor, create a new instance of Product class into the **Entity** property. Create a Validate() method in which you add code to test for valid Product data. This method returns a list of ValidationMessage objects.

```
namespace Samples;
public class ProductViewModel {
  public ProductViewModel() {
    Entity = new();
  }

  public Product Entity { get; set; }

  public List<ValidationMessage> Validate() {
    List<ValidationMessage> msgs = new();
```

Listing 1: Create a Product entity class to test out validation

```
#nullable disable

namespace Samples;

public partial class Product
{
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string ProductNumber { get; set; }
  public string Color { get; set; }
  public decimal? StandardCost { get; set; }
  public decimal? ListPrice { get; set; }
  public DateTime SellStartDate { get; set; }
  public DateTime? SellEndDate { get; set; }
  public DateTime? DiscontinuedDate { get; set; }

  public override string ToString()
  {
    return $"{Name} ({ProductID})";
  }
}
```

Listing 2: Write code to test each property in the Entity object

```
if (string.IsNullOrWhiteSpace(Entity.Name)) {
  msgs.Add(new ValidationMessage() {
    ErrorMessage = "Product Name Must Be Filled In.",
    PropertyName = "Name" });
}
else {
  if (Entity.Name.Length > 50) {
    msgs.Add(new ValidationMessage() {
      ErrorMessage = "Product Name Must Be
        50 Characters or Less.",
      PropertyName = "Name" });
  }
}

if (Entity.StandardCost == null ||
    Entity.StandardCost < 0.01M) {
  msgs.Add(new ValidationMessage() {
    ErrorMessage = "Cost Must Be Greater
      Than Zero.",
    PropertyName = "StandardCost" });
}
if (Entity.ListPrice == null ||
    Entity.ListPrice < 0.01M) {
  msgs.Add(new ValidationMessage() {
    ErrorMessage = "Price Must Be Greater
      Than Zero.",
    PropertyName = "ListPrice" });
}
if (Entity.ListPrice < Entity.StandardCost) {
  msgs.Add(new ValidationMessage() {
    ErrorMessage = $"Price must be greater
      than the Cost.",
    PropertyName = "ListPrice"
  });
}

if (Entity.SellStartDate == DateTime.MinValue) {
  msgs.Add(new ValidationMessage() {
    ErrorMessage = $"Selling Start Date Must Be
      Greater Than '{DateTime.Now.AddDays(-5)
        .ToShortDateString()}'.",
    PropertyName = "SellStartDate" });
}
```

```
    // Insert Validation Code Here
    return msgs;
  }
}
```

Within the Validate() method, where the comment says to Insert Validation Code Here, write the code shown in **Listing 2** to test each of the Product properties for valid data. If the data in a property isn't valid, create a new instance of the ValidationMessage class and put the property name into **PropertyName** property, and the message you want to convey to the user in the **Message** property.

Open the **Program.cs** file and add the code shown in **Listing 3**. In this code, you create an instance of the ProductViewModel class and fill in a few properties of the **Entity** property. Call the Validate() method on the ProductViewModel object and it returns a collection of ValidationMessage objects. Iterate over this collection and display each error message on the console.

Try It Out

Run the program to see the error messages appear on the Console window, as shown in **Figure 1**.

Microsoft Data Annotations

Instead of writing validation code in your view model class as you did in **Listing 1**, you can add attributes above those properties in your Product class that you wish to validate. There are many standard data annotation attributes supplied by Microsoft such as [Required], [MinLength], [MaxLength], [StringLength], and [Range]. From the name of the attribute, you can infer what each of these attributes validates on a property. For a complete list of data annotation attributes, visit Microsoft's website at https://bit.ly/3TJICid.

Each of the data annotation attributes inherits from an abstract class named **ValidationAttribute**. This validation attribute class has properties that each of the inherited attribute classes can use. These properties are shown in **Table 1**.

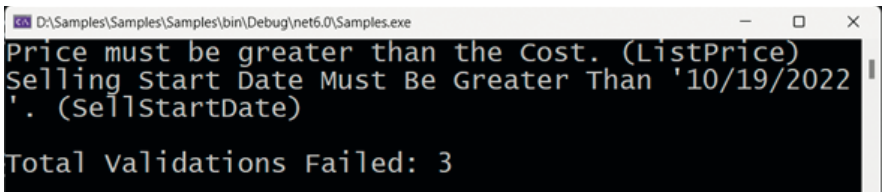


Figure 1: Validation messages can be displayed however you wish, depending on the type of .NET application you're writing.

Property Name	Description
ErrorMessage	Get/Set the error message format string
ErrorMessageString	Gets the localized error message
ErrorMessageResourceName	Get/Set the error message resource name
ErrorMessageResourceType	Get/Set the error resource class type

Table 1: The common properties available to all data annotation attribute classes

The ErrorMessage Property

The **ErrorMessage** property is what you use to report back the error message to the user. You can use a hard-coded string or you can have placeholders in the string to automatically retrieve data from the property the attribute is decorating. The placeholders are what the String.FormatString() method uses where you add numbers enclosed within curly braces as shown in the following [Required] attribute.

```
[Required(ErrorMessage = "{0} Must Be Filled In")]
public string ProductNumber { get; set; }
```

Listing 3: Test the Product validation in the Program file

```
using Samples;

// Create view model and
// initialize the Entity object
ProductViewModel vm = new() {
  Entity = new() {
    Name = "",
    ListPrice = 5,
    StandardCost = 15
  }
};

// Validate the Data
var msgs = vm.Validate();

// Display Failed Validation Messages
foreach (ValidationMessage item in msgs) {
  Console.WriteLine(item);
}

// Display Total Count
Console.WriteLine();
Console.WriteLine($"Total Validations
  Failed: {msgs.Count}");

// Pause for Results
Console.ReadKey();
```

Listing 4: Use the ValidationContext and Validator objects to validate properties decorated with data annotations

```
public List<ValidationMessage> Validate() {
  List<ValidationMessage> msgs = new();

  // Create instance of ValidationContext object
  ValidationContext context =
    new(Entity, serviceProvider: null,
        items: null);
  List<ValidationResult> results = new();

  // Call TryValidateObject() method
  if (!Validator.TryValidateObject(Entity, context,
      results, true)) {
    // Get validation results
    foreach (ValidationResult item in results) {
      string propName = string.Empty;
      if (item.MemberNames.Any()) {
        propName = ((string[])item.MemberNames)[0];
      }
      // Build new ValidationMessage object
      ValidationMessage msg = new() {
        ErrorMessage = item.ErrorMessage,
        PropertyName = propName
      };

      // Add validation object to list
      msgs.Add(msg);
    }
  }

  return msgs;
}
```

The {0} placeholder is replaced with the name of the property the attribute is decorating. In the above example, the resulting string is "ProductNumber Must Be Filled In". Next, look at the following code snippet that uses the [Range] attribute.

```
[Range(0.01, 9999,
  ErrorMessage = "{0} must be between
                  {1} and {2}")]
public decimal? StandardCost { get; set; }
```

The {1} placeholder is replaced with the value in the first parameter to the Range attribute and the {2} placeholder is replaced with the value in the second parameter. If you have more parameters, then you keep on incrementing the placeholder accordingly.

The [Required] Attribute

Let's explore the [Required] attribute in a little more detail and see how to check any properties with this attribute applied. Open the **Product.cs** file and add the [Required] attribute just above the **Name, ProductNumber**, **StandardCost**, **ListPrice**, and **SellStartDate** properties as shown in the following code.

```
[Required]
public string Name { get; set; }
[Required]
public string ProductNumber { get; set; }

[Required]
public decimal? StandardCost { get; set; }
[Required]
public decimal? ListPrice { get; set; }
[Required]
public DateTime SellStartDate { get; set; }
```

If you're using ASP.NET and MVC, the data annotation attributes attached to the properties in a class are automatically validated. If you are using WPF, Windows Forms, or a console application, you need to manually validate those data annotations. There are three classes built-in to .NET you use to perform this validation. The **ValidationContext**, the **Validator**, and the **ValidationResult** classes are used to generate the error messages from the data annotations. In the example above, you want these classes to return a message that says the user needs to fill in data into those properties decorated with the [Required] attribute. Open the **ProductViewModel.cs** file, locate the Validate() method and replace all of the code with the code shown in **Listing 4.**

This code creates an instance of a ValidationContext object passing in the **Entity** property. Create an instance of list of ValidationResult objects so the TryValidateObject() method can fill in this list with all the ValidationResult objects.

The TryValidateObject() method is responsible for checking all data annotations attached to each property in the entity object. If any validations fail, the appropriate error message, along with the property name, is returned in the **results** variable. Loop through the results collection and add a new ValidationMessage object to the **ValidationMessages** property. The **ErrorMessage** property is filled in

with the **ErrorMessage** property from the current ValidationResult item. The property name is retrieved from the first element of the **MemberNames** property (see **Figure 2**) on the ValidationResult item. It's possible for a data annotation to have two properties to which it applies, but for most simple properties, you only need to grab the first property name.

Try It Out

Open the **Program.cs** file and modify the view model so the **Entity** object is initialized with an empty string for both the **Name** and **ProductNumber** properties as shown in the following code.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "",
    ProductNumber = ""
  }
};
```

Run the application and you should see the output shown in **Figure 3**.

Add ErrorMessage Property to the [Required] Attribute

If you want to change the default error message generated in the ValidationResult object, fill in the **ErrorMessage** property on the [Required] attribute. Open the **Product.cs** file and modify the [Required] attributes above the **Name** and **ProductNumber** properties to look like the following.

```
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
public string Name { get; set; }
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
public string ProductNumber { get; set; }
```

Try It Out

Run the application and you should see the **Name** and **ProductNumber** property error messages are different from the **StandardCost** and **ListPrice** property error messages as shown in **Figure 4**.

The [DisplayName] Attribute

Displaying the property name to the user is generally not a good idea. Sometimes the property name won't mean much to the user. It's better to use a more readable string, such as the same label displayed on an input form. You can accomplish this by adding the [DisplayName] attribute to any property in your class. Open the **Product.cs** file and add the [DisplayName] attribute above the properties shown in **Listing 5**. If the [DisplayName] attribute is attached to a property, the {0} placeholder in the **ErrorMessage** property uses the **Name** property from the [DisplayName] attribute instead of the actual property name.

Try It Out

After applying the [DisplayName] attribute, run the application and you should now see better error messages, as shown in **Figure 5**. The property name is displayed in the parentheses, so you can clearly see the difference.

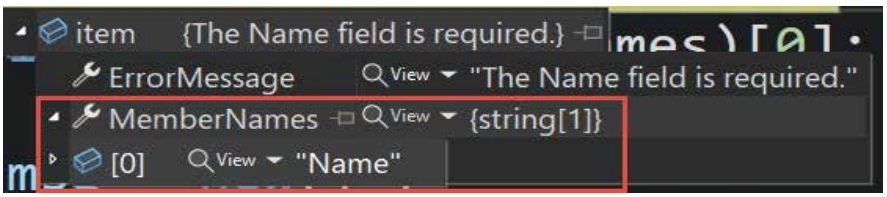


Figure 2: The ValidationResult class contains an error message and a list of property names to which the error message applies.

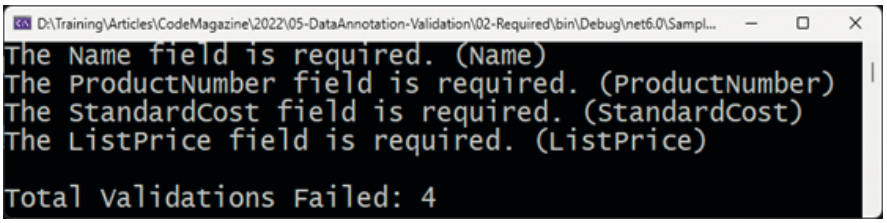


Figure 3: The [Required] attribute creates a string with the property name in the error message.

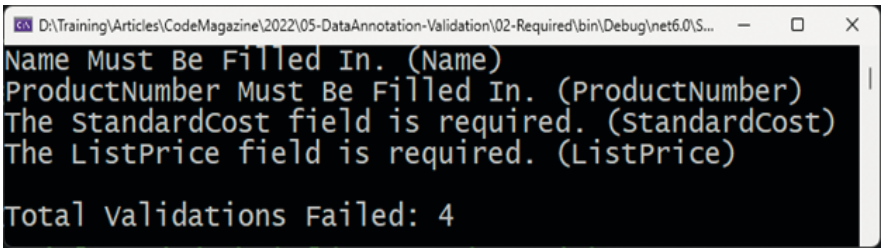


Figure 4: Use the ErrorMessage property to change the message to display to the user.

Listing 5: Apply the [DisplayName] attribute to your properties to receive better error messages

```
[Display(Name = "Product Name")]
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
public string Name { get; set; }

[Display(Name = "Product Number")]
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
public string ProductNumber { get; set; }

[Display(Name = "Product Color")]
public string Color { get; set; }

[Display(Name = "Cost")]
[Required]
public decimal? StandardCost { get; set; }

[Display(Name = "Price")]
[Required]
public decimal? ListPrice { get; set; }

[Display(Name = "Start Selling Date")]
[Required]
public DateTime SellStartDate { get; set; }

[Display(Name = "End Selling Date")]
public DateTime? SellEndDate { get; set; }

[Display(Name = "Date Discontinued")]
public DateTime? DiscontinuedDate { get; set; }
```

Listing 6: Create a generic helper class to perform all the validation for your application

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public static class ValidationHelper {
  public static List<ValidationMessage>
    Validate<T>(T entity) {
    List<ValidationMessage> ret = new();

    ValidationContext context = new(entity,
      serviceProvider: null, items: null);
    List<ValidationResult> results = new();

    if (!Validator.TryValidateObject(entity,
        context, results, true)) {
      foreach (ValidationResult item in results) {
        string propName = string.Empty;
        if (item.MemberNames.Any()) {
          propName =
            ((string[])item.MemberNames)[0];
        }
        ValidationMessage msg = new() {
          ErrorMessage = item.ErrorMessage,
          PropertyName = propName
        };

        ret.Add(msg);
      }
    }

    return ret;
  }
}
```

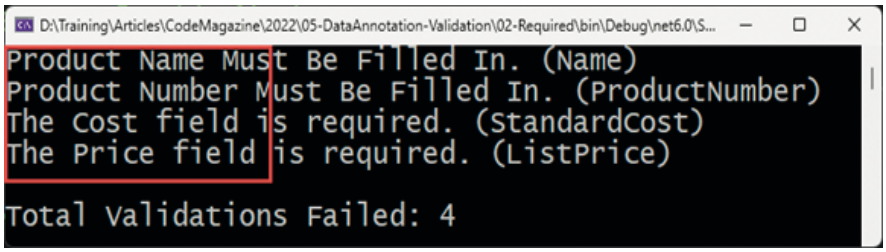


Figure 5: Using the [DisplayName] attribute provides more user-friendly error messages.

Create a Generic Helper Class

Most likely, you're not going to only have a single view model class in your application. Thus, you don't want to write a duplicate of the Validate() method in each of your view models. You can either inherit from a base view model class, or you can create a static class with a method to perform the validation for you. Right mouse-click on the ValidationClasses folder and create a new class named **ValidationHelper**. Into this new file, replace the code with the code shown in **Listing 6**.

The code in **Listing 6** is similar to the code in the Validate() method you wrote in the Product class, but the Validate() method in this code is generic and can accept any type. A new List<ValidationMessage> collection is built each time the Validate() method is called, and it's this list that's returned from this method.

Try It Out

Open the **ProductViewModel.cs** file and replace the code in the Validate() method with the following code.

```
public List<ValidationMessage> Validate() {
  // Use Helper Class
  return ValidationHelper.Validate(Entity);
}
```

Run the application and you should see the same results as previously, but the validation is now happening in the Validate() method in the ValidationHelper class. Each view model class you create from now on just needs this very simple Validate() method. Of course, you can still create a view model base class and move this method into the base class, then have all your view model classes inherit from this base class.

Attributes for Length of Data

There are a few different attributes that you can use to check for the length of string data within your properties. Let's look at the most common data annotations that you're most likely to use.

The [MaxLength] Attribute

The [MaxLength] attribute allows you to specify the total length the string data within a property should be. If the data within that string property exceeds the specified length, an appropriate error message is returned from the validation. Open the **Product.cs** file and decorate the **ProductNumber** and **Color** properties with the following code.

```
[Display(Name = "Product Number")]
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
[MaxLength(25)]
public string ProductNumber { get; set; }

[Display(Name = "Product Color")]
[MaxLength(15)]
public string Color { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the properties on the **Entity** object to look like the following.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "Product 1",
    ProductNumber = "A very long product name
      to illustrate the [MaxLength] property.",
    Color = "A very long color name."
  }
};
```

Run the application and you should see the appropriate error messages displayed for both the **ProductNumber** and **Color** properties.

The [MinLength] Attribute

Sometimes you need to ensure a minimum amount of string data is added to a property. For example, the product color shouldn't have any string data that is less than

three characters, as there are no one- or two-letter colors. Add a [MinLength] attribute to the **Color** property in the Product class, as shown in the following code.

```
[Display(Name = "Product Color")]
[MinLength(3,
  ErrorMessage =
    "{0} Must Have {1} Characters or More.")]
[MaxLength(15)]
public string Color { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code:

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "Product 1",
    ProductNumber = "PROD001",
    Color = "Re"
  }
};
```

Run the application and you should see the appropriate error message displayed for the **Color** property.

The [String Length] Attribute

If you don't want to apply both the [MinLength] and [MaxLength] attributes to a single property, you may use the [StringLength] attribute as it supports both maximum and minimum length properties. When adding the [StringLength] attribute, the first parameter is the maximum length, and then you can specify the **MimimumLength** as either the second parameter or as a named parameter, as I have done in the following code. Add the [StringLength] attribute above the **Name** property in the Product class.

```
[Display(Name = "Product Name")]
[Required(ErrorMessage =
  "{0} Must Be Filled In.")]
[StringLength(50,
  MinimumLength = 4,
  ErrorMessage = "{0} Can Only Have Between
    {2} and {1} Characters.")]
public string Name { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A"
  }
};
```

Run the application and you should see the appropriate error message displayed for the **Name** property. If you want to ensure the maximum length of the **Name** property also works, try setting the **Name** property to the following and then run the program again.

```
Name = "A very long product name used
  to illustrate [StringLength] attribute."
```

The Range Validator

If you have decimal or int properties in your class, you can use the [Range] attribute to check for a minimum and a maximum value that can be entered into those numbers. You may also use a DateTime with the [Range] attribute, but you need to add an additional parameter.

Use the [Range] Attribute with Numeric Values

In the Product class, you have the **StandardCost** and the **ListPrice** properties that you should apply a [Range] attribute to. You don't want a cost or a price to be less than zero dollars. Open the **Product.cs** file and locate the **StandardCost** and **ListPrice** properties and add the [Range] attribute as shown below. Be sure to include the **ErrorMessage** property so you can format the cost as currency.

```
[Display(Name = "Cost")]
[Required]
[Range(0.01, 9999,
  ErrorMessage = "{0} must be between
    {1:c} and {2:c}")]
public decimal? StandardCost { get; set; }

[Display(Name = "Price")]
[Required]
[Range(0.01, 9999,
  ErrorMessage = "{0} must be between
    {1:c} and {2:c}")]
public decimal? ListPrice { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Black",
    StandardCost = 0,
    ListPrice = 10000,
    SellStartDate = DateTime.Now,
    SellEndDate = DateTime.Now.AddDays(+365)
  }
};
```

Run the application and you should see error messages that look like **Figure 6**.

Use the [Range] Attributes with DateTime

When using the [Range] attribute with numbers, you specify the minimum and maximum values as the first and the second parameters. When you wish to check a property for a date range, you must pass to the first parameter

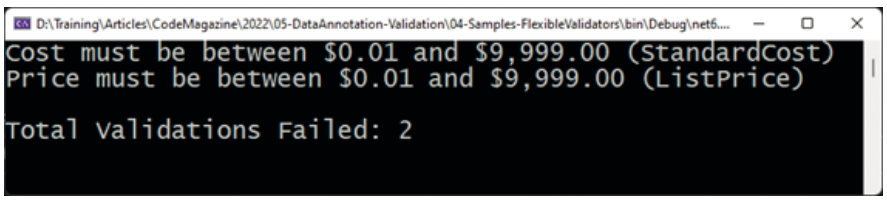


Figure 6: Modify the ErrorMessage property to display decimal values as currency if appropriate.

a typeof(DateTime) so the [Range] attribute class knows to check for a DateTime range. Open the **Product.cs** file and add a [Range] attribute to the **SellStartDate** and the **SellEndDate** properties.

```
[Display(Name = "Start Selling Date")]
[Required]
[Range(typeof(DateTime), "1/1/2000",
  "12/31/2030", ErrorMessage = "{0} must be
     between {1:d} and {2:d}")]
public DateTime SellStartDate { get; set; }

[Display(Name = "End Selling Date")]
[Range(typeof(DateTime), "1/1/2000",
  "12/31/2030", ErrorMessage = "{0} must be
    between {1:d} and {2:d}")]
public DateTime? SellEndDate { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Black",
    StandardCost = 1,
    ListPrice = 10,
    SellStartDate =
      Convert.ToDateTime("1/1/1999"),
    SellEndDate =
      Convert.ToDateTime("1/1/2031")
  }
};
```

Run the application and you should see error messages that look like **Figure 7**.

Regular Expression and Compare Validators

In some classes, you may need the user to adhere to a specific format for the data. For example, phone numbers, social security numbers, etc. The [RegularExpression] attribute can enforce the formatting of data. You need to understand regular expressions to use this attribute, but luckily, there are many resources on the internet to help you with regular expressions. I like www.regexlib.com to look up all sorts of regular expressions. To try out a regular expression, right mouse-click on the EntityClasses folder and add a new class named **User** that looks like the following code.

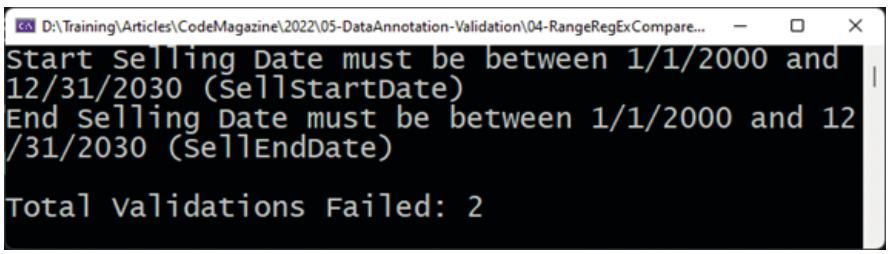


Figure 7: The [Range] attribute can be used with DateTime values as well as numeric values

Get 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 dropdown. Then select "The Rich Set of Data Annotation and Validation Attributes in .NET" from the Item dropdown.

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public partial class User {
  public int UserId { get; set; }
  public string LoginId { get; set; }
  public string Password { get; set; }
  public string ConfirmPassword { get; set; }
  public string EmailAddress { get; set; }
  public string Phone { get; set; }
}
```

Right mouse-click on the ViewModelClasses folder and add a new class named **UserViewModel**. This class is exactly like the ProductViewModel class in that encapsulates the User class as a property named **Entity** and has a Validate() method. Enter the code shown below into the UserViewModel.cs file.

```
namespace Samples;

public class UserViewModel {
  public UserViewModel() {
    Entity = new();
  }

  public User Entity { get; set; }

  public List<ValidationMessage> Validate() {
    // Use Helper Class
    return ValidationHelper.Validate(Entity);
  }
}
```

The [RegularExpression] Attribute

Let's add a [RegularExpression] attribute to both the **EmailAddress** and the **Phone** properties in the User class. When adding the regular expression, don't break them across two lines. I had to break them due to formatting limitations of this printed magazine. I'd highly recommend you include the ErrorMessage property, otherwise, it spits out the regular expression to the user.

```
[RegularExpression("^\\w+@[a-zA-Z_]+?\\.
  [a - zA - Z]{2, 3}$", ErrorMessage =
    "The email address entered is not valid.")]
public string EmailAddress { get; set; }

[RegularExpression("((\\(\\d{3}\\) ?)|
  (\\d{3}-))?\\d{3}-\\d { 4}", ErrorMessage =
    "The phone number entered is not valid.
     Please use the format (nnn) nnn-nnnn")]
public string Phone { get; set; }
```

Try It Out

Open the **Program.cs** file and create a new instance of the UserViewModel class and initialize the **Entity** property with the values shown in the code below so you can test the regular expressions.

```
UserViewModel vm = new() {
  Entity = new() {
```

```
    UserId = 1,
    LoginId = "JoeSmith",
    Password = "Joe!Smith@2022",
    EmailAddress = "test!test.com",
    Phone = "xxx-xxx-xxxx"
  }
};
```

Run the application and you should see the error messages, as shown in **Figure 8**.

The [Compare] Attribute

A common business rule to enforce is when a user is setting up a new account and they need to put in a new password. It's best to ask them to input that password two times to ensure that they don't misspell the password. In the User class, there's both a **Password** and **ConfirmPassword** properties. The [Compare] attribute lets you check to ensure that the data contained in both properties is an exact match. Apply the [Compare] attribute to one of the properties and pass in the name of the other property to compare the data to as shown in the following code. It's a best practice to use the nameof() operator so you can rename the property using the Visual Studio rename menu and it will get refactored correctly.

```
[Compare(nameof(ConfirmPassword))]
public string Password { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code. Notice the two different values within the **Password** and the **ConfirmPassword** properties.

```
UserViewModel vm = new() {
  Entity = new() {
    UserId = 1,
    LoginId = "JoeSmith",
    Password = "JoeSmith@2022",
    ConfirmPassword = "JoeSmith",
    EmailAddress = "JoeSmith@test.com",
    Phone = "(999) 999-9999",
  }
};
```

Run the application and the error message you get tells you the names of the properties that don't match, as shown in **Figure 9**.

Standard Business Rule Validations

Microsoft realizes that working with regular expressions isn't always the easiest thing to do. They therefore added many attributes to help you enforce the most common business rules such as email, phone, URL, and credit cards. Open the **User.cs** file and remove the two [RegularExpression] attributes you added to the **EmailAddress** and **Phone** properties in the last section.

The [EmailAddress] Attribute

Apply the [EmailAddress] attribute to the **EmailAddress** property, as shown in the following code.

```
[EmailAddress]
public string EmailAddress { get; set; }
```

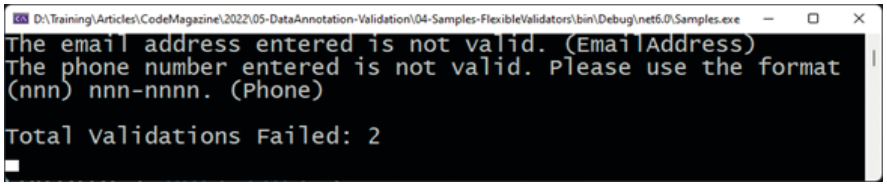


Figure 8: It's best to use your own error messages for the regular expression attributes.

Figure 9: The [Compare] attribute compares the data between two properties.

Apply the [Phone] attribute to the Phone property as shown in the following code.

```
[Phone]
public string Phone { get; set; }
```

Try It Out

Open the **Program.cs** file and modify the initialization of the **Entity** object to look like the following code. Notice that there's an invalid format for both the **EmailAddress** and **Phone** properties.

```
UserViewModel vm = new() {
  Entity = new() {
    UserId = 1,
    LoginId = "JoeSmith",
    Password = "Joe!Smith@2022",
    ConfirmPassword = "Joe!Smith@2022",
    EmailAddress = " Joe!Smith.2022",
    Phone = "12.34.asdf"
  }
};
```

Run the application and you should see the appropriate error messages for both the email address and phone number properties.

The [Url] Attribute

If you have a URL property in your class, you can use the [Url] attribute to ensure the data contained within that URL is valid. Be aware that the URL entered into your property must start with http://, https://, or ftp://. If you don't want these prefixes, you won't be able to use the [Url] attribute. Open the **Product.cs** file and add a **ProductUrl** property and add a [Url] data annotation to it as shown below.

```
[Display(Name = "Product URL")]
[Url]
public string ProductUrl { get; set; }
```

Try It Out

Open the **Program.cs** file and create a new ProductViewModel object and instantiate the **Entity** property, as shown in the following code.

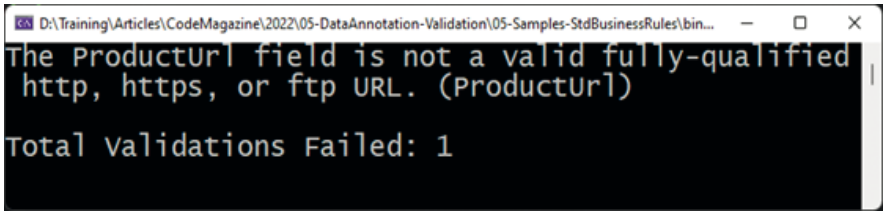


Figure 10: The [Url] attribute must contain a valid internet prefix to be considered valid.

Listing 7: Create a CreditCard class to test the [CreditCard] annotation

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public partial class CreditCard
{
  public string CardType { get; set; }
  public string NameOnCard { get; set; }
  [CreditCard()]
  public string CardNumber { get; set; }
  public string SecurityCode { get; set; }
  public int ExpMonth { get; set; }
  public int ExpYear { get; set; }
  public string BillingPostalCode { get; set; }
}
```

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    Color = "Black",
    StandardCost = 5,
    ListPrice = 10,
    ProductUrl = "asdf.test"
  }
};
```

Run the application and you should see the error message shown in **Figure 10**.

The [CreditCard] Attribute

Another common business rule is to check for valid credit card data entered by a user. To try this out, right mouse-click on the EntityClasses folder and add a new class named **CreditCard**. In the new CreditCard class add the code shown in **Listing 7**. Notice the use of the [CreditCard] attribute decorating the **CardNumber** property.

Right mouse-click on the ViewModelClasses folder and add a new class named **CreditCardViewModel,** as shown in the following code.

```
namespace Samples;

public class CreditCardViewModel {
  public CreditCardViewModel() {
    Entity = new();
  }

  public CreditCard Entity { get; set; }

  public List<ValidationMessage> Validate() {
    // Use Helper Class
    return ValidationHelper.Validate(Entity);
  }
}
```

Try It Out

Open the **Program.cs** file and create a new instance of the CreditCardViewModel class and set the appropriate properties of the **Entity** property, as shown in the following code.

```
CreditCardViewModel vm = new() {
  Entity = new() {
    CardType = "Visa",
    CardNumber = "12 13 123 1234",
    NameOnCard = "Joe Smith",
    BillingPostalCode = "99999",
    ExpMonth = 01,
    ExpYear = 2026,
    SecurityCode = "000"
  }
};
```

Run the application and you should see an error message informing you that the **CardNumber** property is not a valid credit card number.

Create Custom Validators Using the [CustomValidation] Attribute

There's no way that Microsoft can anticipate all the needs for business rule validation. They've provided a couple of different methods to create custom validation using attributes. The first method is to use the [CustomValidation] attribute. This attribute accepts two parameters, the first parameter is the type of a class in which you write a static method that returns a ValidationResult object. The second parameter is the name of that static method. Right mouse-click on the ValidationClasses folder and add a new class to your project named **WeekdayOnlyValidator** and to this new file add the following code:

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples {
  public class WeekdayOnlyValidator {
    public static ValidationResult
      Validate(DateTime date) {
      return date.DayOfWeek == DayOfWeek.Saturday
       || date.DayOfWeek == DayOfWeek.Sunday
         ? new ValidationResult("Invalid date
           because it falls on a weekend")
         : ValidationResult.Success;
    }
  }
}
```

The Validate() method checks the date passed in to ensure that it doesn't fall on a Saturday or a Sunday. If the date does fall on a weekend, return a ValidationResult object with the error message inside. Otherwise, return a ValidationResult.Success from this method. Right mouse-click on the EntityClasses folder and add a class named **Customer**. Add the [CustomValidation] attribute to decorate an **EntryDate** property in the class as shown below.

```
using System.ComponentModel.DataAnnotations;

namespace Samples;

public class Customer {
  [CustomValidation(typeof(WeekdayOnlyValidator),
   nameof(WeekdayOnlyValidator.Validate)]
   public DateTime EntryDate { get; set; }
}
```

Right mouse-click on the ViewModelClasses folder and add a new class named **CustomerViewModel**. Add the code shown in the code snippet below to this new file.

```
namespace Samples;

public class CustomerViewModel {
  public CustomerViewModel() {
    Entity = new();
  }

  public Customer Entity { get; set; }

  public List<ValidationMessage> Validate() {
    // Use Helper Class
    return ValidationHelper.Validate(Entity);
  }
}
```

Try It Out

Open the **Program.cs** file and add a new instantiation of the CustomerViewModel class that sets the **EntryDate** property of the Entity object to an invalid date.

```
CustomerViewModel vm = new() {
  Entity = new() {
    EntryDate = DateTime.Parse("10/1/2022")
  }
};
```

Run this code and because the date 10/1/2022 falls on a weekend, the validation message "Invalid date because it falls on a weekend." should appear on your console window.

Create Custom Validators by Inheriting from the ValidationAttribute Class

To me, a better method for performing validation is to inherit from the ValidationAttribute class rather than using the [CustomValidation] attribute. The main reason is that you can name your custom validation class something that's distinct, and easy to read and understand.

There's a common design pattern you use when creating your own custom validation attribute by inheriting from the ValidationAttribute class. The following list describes the things you must do in your validation class.

- Name your class to describe what the validation does.
- Pass to the constructor any parameter(s).
- Override the IsValid() method.
- Validate the data entered is correct.
- Return a ValidationResult object.
  - Add an error message if validation is not successful.
  - Return ValidationResult.Success if validation is successful.

Is Date Greater Than Minimum Date

A good use of a custom validation attribute is to determine if a date entered by a user is greater than or equal to a specific date. Below is an example of how you might use this attribute to enforce this business rule.

```
[DateMinimum("9/1/2022")]
public DateTime? DiscontinuedDate { get; set; }
```

Pass to this custom attribute the minimum date that the data entered by the user should be. If the date entered is this date or greater, then the data is valid. Otherwise, an error message is returned from this attribute, and you can display that message to the user.

Right mouse-click on the ValidationClasses folder and create a new class named **DateMinimumAttribute**. To this new file, add the code shown in **Listing 8**. The first thing you must do is to inherit your class from the ValidationAttribute class. The constructor needs to receive the string that represents the minimum date. You should then convert that string to a DateTime and store it into a read-only private field named **_minDate**. I'm not performing an error checking to ensure that the date passed is valid, but in production code, you should add error checking.

Override the IsValid() method to write the code you need to enforce your business rule in your validation class. In this

Listing 8: Create a DateMinimumAttribute class to test for a valid minimum date

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public class DateMinimumAttribute
  : ValidationAttribute {
  public DateMinimumAttribute(string minDate) {
    _minDate = Convert.ToDateTime(minDate);
  }

  private readonly DateTime _minDate;

  protected override ValidationResult IsValid(
      object value, ValidationContext vc) {
    if (value != null) {
      // Get the value entered
      DateTime dateEntered = (DateTime)value;

      // Get display name for validation message
      string displayName = vc.DisplayName;

      // If the date entered is less than
      // or equal to the minimum date set
      // return an error
      if (dateEntered <= _minDate) {
        // Check if ErrorMessage is filled in
        if (string.IsNullOrEmpty(ErrorMessage)) {
          ErrorMessage = $"{displayName} must be
            greater than or equal to
            '{_minDate:MM/dd/yyyy}'.";
        }

        return new ValidationResult(ErrorMessage,
          new[] { vc.MemberName });
      }
    }

    return ValidationResult.Success;
  }
}
```

case, I verify that the value passed in is not equal to null. If it isn't null, convert the value entered into the property to a DateTime object. Next, get the name of the property, or the value specified in the [DisplayName] property, so you can use this if you need to return an error message.

Check to see if the date entered is less than or equal to the **_minDate** field. If it is, return a new ValidationResult object. If the **ErrorMessage** property is not filled in, create a message to display. The first parameter to the ValidationResult constructor is the error message you wish to display. For the second parameter, create a new string array and fill in the **MemberName** from the ValidationContext. This **MemberName** property is the actual property name that this attribute is decorating. If there's no validation error, return a ValidationResult.Success from this attribute class to signify that the data was valid.

Try It Out

Open the **Product.cs** file and add the DataMinimum attribute to the **DiscontinuedDate** property, as shown below.

```
[DateMinimum("9/1/2022")]
```

Open the **Program.cs** file and create an instance of the ProductViewModel class and initialize the Entity property to the following code. Notice that the **DiscontinuedDate** property is set to a date less than the minimum date specified in the [DateMinimum] attribute.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 12,
    SellStartDate = DateTime.Today,
    SellEndDate = DateTime.Today.AddYears(+5),
    DiscontinuedDate =
      Convert.ToDateTime("1/1/2020")
  }
};
```

Run the application and view the error message you get back from the DateMinimumAttribute class.

Is Date Less Than Maximum Date

If you want to try out another custom attribute, copy the DateMinimumAttribute.cs file to a new file in the ValidationClasses folder named **DateMaximumAttribute.cs**. Open this new file and change the name of the class to **DateMaximumAttribute**. Rename the **_minDate** field as **_maxDate**. Change the comparison operator from a less than or equal sign (<=) to a greater than or equal to sign (>=). Modify the error message to display "less than" rather than "greater than." You now have another validation attribute that you can use to enforce business rules.

Try It Out

Open the **Product.cs** file and add to the **SellEndDate** property the [DateMaximum] attribute, as shown below.

```
[DateMaximum("12/31/2030")]
```

Open the **Program.cs** file and initialize the Entity property within the ProductViewModel class to the following code. Notice the **SellEndDate** property is set to a date greater than the maximum date specified in the [DateMaximum] attribute.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 12,
    SellStartDate = DateTime.Today,
    SellEndDate = DateTime.Today.AddYears(+20),
    DiscontinuedDate =
      Convert.ToDateTime("12/15/2022")
  }
};
```

Run the application and view the error message you get back from the DateMaximumAttribute class.

Listing 9: Create a DateYearRangeAttribute class to test for a valid date within two date ranges by year

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public class DateYearRangeAttribute
  : ValidationAttribute {
  public DateYearRangeAttribute(int yearsPrior,
                                int yearsAfter) {
    _minDate = DateTime.Now.AddYears(yearsPrior);
    _maxDate = DateTime.Now.AddYears(yearsAfter);
  }

  private readonly DateTime _minDate;
  private readonly DateTime _maxDate;

  protected override ValidationResult IsValid(
      object value, ValidationContext vc) {
    if (value != null) {
      // Get the value entered
      var dateEntered = (DateTime)value;

      // Get display name for validation message
      string displayName = vc.DisplayName;

      // Is date entered within the date range
      if (dateEntered < _minDate ||
          dateEntered > _maxDate) {
        // Check if ErrorMessage is filled in
        if (string.IsNullOrEmpty(ErrorMessage)) {
          ErrorMessage = $"{displayName} must be
            between '{_minDate:MM/dd/yyyy}' and
            '{_maxDate:MM/dd/yyyy}'.";
        }

        return new ValidationResult(ErrorMessage,
          new[] { vc.MemberName });
      }
    }

    return ValidationResult.Success;
  }
}
```

Listing 10: Create a CompareDateLessThanAttribute class to test for one date must be less than another date

```
#nullable disable

using System.ComponentModel.DataAnnotations;
using System.Reflection;

namespace Samples;

public class CompareDateLessThanAttribute
    : ValidationAttribute {
  public CompareDateLessThanAttribute(
     string propToCompare) {
    _propToCompare = propToCompare;
  }

  private readonly string _propToCompare;

  protected override ValidationResult IsValid(
      object value, ValidationContext vc) {
    if (value != null) {
      // Get value entered
      DateTime currentValue = (DateTime)value;
      // Get PropertyInfo for comparison property
      PropertyInfo pinfo = vc.ObjectType
        .GetProperty(_propToCompare);
      // Ensure the comparison property
      // value is not null
      if (pinfo.GetValue(vc.ObjectInstance) != null) {
        // Get value for comparison property
        DateTime comparisonValue = (DateTime)pinfo
          .GetValue(vc.ObjectInstance);
        // Perform the comparison
        if (currentValue > comparisonValue) {
          return new ValidationResult(ErrorMessage,
            new[] { vc.MemberName });
        }
      }
    }

    return ValidationResult.Success;
  }
}
```

Custom Validator: Dynamic Date Range by Year

The problem with the [Range] attribute when working with dates is that the minimum and maximum dates you enter must be hard-coded strings. What if you want to make the range a little more dynamic? For example, you might want to specify the minimum year the user may enter is two years prior to today's date. And the maximum year the user may enter is five years after today's date. To accomplish this, create a [DateYearRange] attribute class and pass in two integer values that specify the years prior and after that are valid for the date entered.

```
[DateYearRange(-2, 5)]
public DateTime SellStartDate { get; set; }
```

Right mouse-click on the ValidationClasses folder and add a new class named **DateYearRangeAttribute**. Into this file add the code shown in **Listing 9**. In the constructor, you accept the integer values and use those to calculate the two private read-only fields: **_minDate** and **_maxDate**.

Within the IsValid() method, retrieve the value entered from the user, then check to see if that date entered is less than the **_minDate** field or if it's greater than the **_maxDate** field. If either of these conditions fail, return a ValidationResult object with the error message telling the user the date range their input value must fall within.

Try It Out

Open the **Product.cs** file and add to the **SellStartDate** property the [DateYearRange] attribute, as shown below.

```
[DateYearRange(-2, 5)]
```

Open the **Program.cs** file and initialize the **Entity** property within the ProductViewModel class to the following code. Notice that the **SellStartDate** property is set to six years prior to today's date. This will cause the [DateYearRange] attribute to fail the validation.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 12,
    SellStartDate = DateTime.Today.AddYears(-6),
    SellEndDate = DateTime.Today,
    DiscontinuedDate = DateTime.Today
  }
};
```

Run the application and view the error message you get back from the DateYearRange class.

Custom Validator: Is One Date Property Less Than Another

Earlier in this article, you learned about the [Compare] attribute, which allows you to check if the data in one property is equal to the data in another. Let's create a validation attribute that does something similar but checks to see if one date value is less than another value. For example, you might want check to see if the **SellStartDate** is less than the **SellEndDate** property in the Product class.

Right mouse-click on the ValidationClasses folder and add a class named **CompareDateLessThanAttribute.cs**. Replace all the code in this file with the code shown in **Listing 10.** The constructor accepts the name of the property to compare to as a string value. Place this value into a private read-only field named **_propToCompare**. Retrieve the value entered by the user from the **value** parameter passed in and convert it to a DateTime type. Use the GetProperty() method on the ValidationContext.ObjectType object to retrieve the actual address of where the property to compare to is located in memory. Once you have the PropertyInfo object call the GetValue() method to get the value in the property to compare to. If that value is not null, use that value to perform the comparison to the current value so you know whether to return an error message.

Try It Out

Open the **Product.cs** file and add to the **SellStartDate** property the [CompareDateLessThan] attribute, as shown below. The first parameter to the attribute is the name of the property you want to compare it to.

```
[CompareDateLessThan(nameof(SellEndDate),
  ErrorMessage = "Start Selling Date must be
  less than the End Selling Date.")]
```

Open the **Program.cs** file and initialize the Entity property within the ProductViewModel class to the following code. Notice that the **SellEndDate** property is set to one day prior to the **SellStartDate**. This causes the [CompareDateLessThan] attribute to fail the validation.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 12,
    SellStartDate = DateTime.Today,
    SellEndDate = DateTime.Today.AddDays(-1),
    DiscontinuedDate =
      Convert.ToDateTime("1/1/2023")
  }
};
```

Run the application and view the error message you get back from the CompareDateLessThan class.

Custom Validator: Is One Numeric Property Less Than Another

If you want to check if a numeric property is less than another numeric property, copy the CompareDateLessThanAttribute.cs file to a new file named **CompareDecimalLessThanAttribute.cs**. Open this new file and change the name of the class to **CompareDecimalLessThanAttribute**. Change all instances of **DateTime** to **decimal**. You now have another validation attribute that you can use to ensure that one decimal property is less than another.

Try It Out

Open the **Product.cs** file and add to the **StandardCost** property the [CompareDecimalLessThan] attribute, as shown in the code below. The first parameter to the attribute is the name of the property you want to compare it to.

```
[CompareDecimalLessThan(nameof(ListPrice),
  ErrorMessage =
    "Cost must be less than the Price.")]
```

Open the **Program.cs** file and initialize the **Entity** property within the ProductViewModel class to the following code. Notice that the **ListPrice** property is set to a value less than the value in the **StandardCost** property. This causes the [CompareDecimalLessThan] attribute to fail the validation.

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "A New Product",
    ProductNumber = "PROD001",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 2,
    SellStartDate = DateTime.Today,
    SellEndDate = DateTime.Today.AddDays(+1),
    DiscontinuedDate =
      Convert.ToDateTime("1/1/2023")
  }
};
```

Run the application and view the error message you get back from the CompareDecimalLessThan class.

Implement the IValidatableObject Interface

If you have business rules that are very specific to a class, you may not want to inherit from the ValidationAttribute. Instead, you might want to keep the code that performs

Listing 11: Instead of using data annotations, you may implement the IValidatable interface to check properties on your entity classes

```
#nullable disable

using System.ComponentModel.DataAnnotations;

namespace Samples;

public partial class Employee
    : IValidatableObject {
  public int EmployeeId { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public decimal? Salary { get; set; }
  public DateTime StartDate { get; set; }
  public DateTime? TerminationDate { get; set; }

  public override string ToString() {
    return $"{LastName}, {FirstName}";
  }

  public IEnumerable<ValidationResult>
    Validate(ValidationContext vc) {
    List<ValidationResult> ret = new();

    if (FirstName.Length < 2) {
      ret.Add(new ValidationResult("First Name
        must have at least 2 characters.",
        new[] { nameof(FirstName) }));
    }
    if (LastName.Length < 3) {
      ret.Add(new ValidationResult("Last Name
        must have at least 3 characters.",
        new[] { nameof(LastName) }));
    }
    if (Salary < 1) {
      ret.Add(new ValidationResult("Salary
        must be greater than $1.00.",
        new[] { nameof(Salary) }));
    }
    string minStartDate = DateTime.Now
      .AddDays(-7).ToString("D");
    if (StartDate <
        DateTime.Parse(minStartDate)) {
      ret.Add(new ValidationResult($"Start Date
        must be later than {minStartDate}.",
        new[] { nameof(StartDate) }));
    }
    if (TerminationDate.HasValue &&
        TerminationDate < StartDate) {
      ret.Add(new ValidationResult($"Termination
        Date must be later than {StartDate}.",
         new[] { nameof(TerminationDate) }));
    }

    return ret;
  }
}
```

the validation within the class itself. To do this, implement the IValidatableObject interface on your class. Right mouse-click on the EntityClasses folder and add a new class named **Employee**. Replace all the code in this new file with the code shown in **Listing 11**.

This Employee class defines several properties and implements the Validate() method. Within the Validate() method is where the business rules are implemented. This is a simple example, and these rules could have been implemented using data annotations, but I wanted to show you how this method works.

One thing to be aware of is that if you have both data annotations and this interface, the [Required] data annotations are validated first. Once all the [Required] annotations are resolved, other data annotations are then validated. The Validate() method on your class is not called until there are no ValidationResult objects passed back from the TryValidateObject() method in the ValidationHelper class. Then, and only then, is the Validate() method on your class called.

Right mouse-click on the ViewModelClasses folder and add a new class named **EmployeeViewModel**. This class contains an **Entity** property that's of the type Employee and has a Validate() method used to check the business rules of the Employee object.

```
namespace Samples;

public class EmployeeViewModel {
  public EmployeeViewModel() {
    Entity = new();
  }

  public Employee Entity { get; set; }

  public List<ValidationMessage> Validate() {
    // Use Helper Class
    return ValidationHelper.Validate(Entity);
  }
}
```

Try It Out

Open the **Program.cs** file and instantiate the EmployeeViewModel class and set the appropriate properties on the **Entity** object as shown in the following code.

```
EmployeeViewModel vm = new() {
  Entity = new() {
    EmployeeId = 1,
    FirstName = "A",
    LastName = "AA",
    Salary = 0.01m,
    StartDate = DateTime.Now.AddDays(-10),
    TerminationDate =
      DateTime.Now.AddDays(-11)
  }
};
```

Run the application and view the validation messages.

Localizing Error Messages

If you need to work with multiple languages such as English, Spanish, German, etc. You should not hard-code error messages in the data annotation attributes. Instead, move them to a resource file and use the **ErrorMessageResourceName** property on the data annotation instead of the **ErrorMessage** property. You also need to include the **ErrorMessageResourceType** property to specify the namespace and class where the resources are compiled.

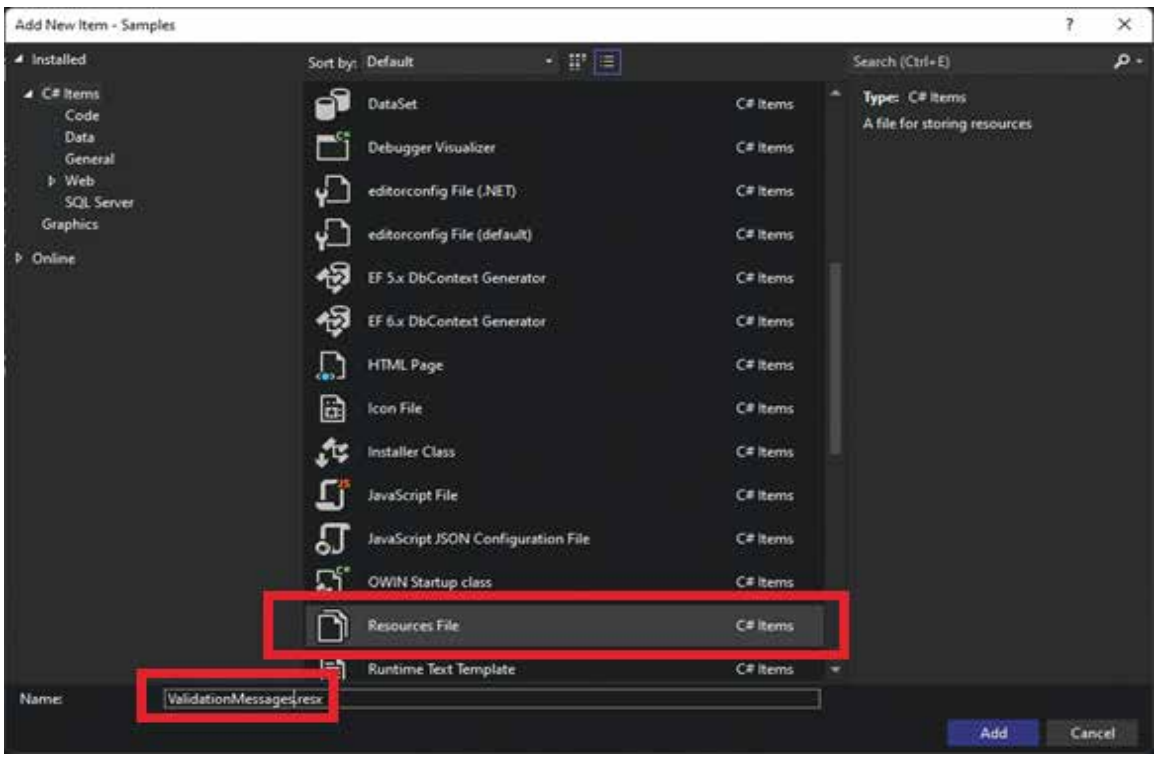


Figure 11: Add a new Resources File to add messages that you can use in your project.

Let's look at how to set up resources, add both English and Spanish error messages, and connect those messages to data annotations.

Create Resources

Add one resource file per language that you wish to support in your application. Right mouse-click on your project and create a new folder named **Resources**. Right mouse-click on the **Resources** folder and select **Add > New Item** from the menu. Locate the Resources File template and set the name to **ValidationMessages.resx**. Click the Add button to add the new resource file to the project, as shown in **Figure 11**.

After adding the resources file, you need to change the Access Modifier to Public, as shown in **Figure 12**. Once you set this modifier, a **ValidationMessages** class is generated by Visual Studio so you can access each resource as a property of that class.

Table 2 is a list of the resources you need to add to the ValidationMessages file. Each of these resources correspond to the similarly named data annotation attribute. These resource names show up as properties in the ValidateMessage class.

Create a Spanish Version of Error Messages

You're going to learn how to assign the resource names to the data annotation attributes in your Product class, but before you do that, let's create the Spanish version of these error messages. Right mouse-click on the **Resources** folder and select **Add > New Item** from the menu. Locate the Resources File template and set the name to **ValidationMessages.es-MX.resx**. Click the Add button to add the new resource file to the project. Add the same names as you did in the first resource file you added. The **Value** property is what changes for each language. In **Figure 13**, you see the values you should enter for the Spanish error messages. Please excuse any bad Spanish grammar as I used "Google Translate."

Why did you add the suffix of ".es-MX" on this file, but didn't use one on the other resource file? The resource file selected is based on two things; the culture running on the computer and the culture set on the current thread. If the two cultures match, the resource file without a suffix is chosen, otherwise the resource file that matches the culture on the current thread is selected.

Modify Product Class to Work with Resources

Let's now modify the Product class to use the values from the resource files instead of the hard-coded messages you've used throughout this article. Open the **Product.cs** file and add a Using statement at the top of this file. This Using statement is the namespace where the **ValidationMessages** class has been generated.

```
using Samples.Resources;
```

Locate the Name property and modify the [Required] and [StringLength] attributes. Remove the **ErrorMessage** property from each of these attributes and add the **ErrorMessageResourceName** and **ErrorMessageResourceType** properties, as shown in the following code:

```
[Required(ErrorMessageResourceName =
    nameof(ValidationMessages.Required),
  ErrorMessageResourceType =
    typeof(ValidationMessages))]

[StringLength(50, MinimumLength = 4,
  ErrorMessageResourceName =
    nameof(ValidationMessages.StringLength),
```

Name	Value
Required	{0} Must Be Filled In.
StringLength	{0} Must Have {1} Characters or Less.
MinLength	{0} Must Have {1} Characters or More.
MaxLength	{0} Must Have {1} Characters or Less.

Table 2: Add a resource name and value for each message you wish to display in your application

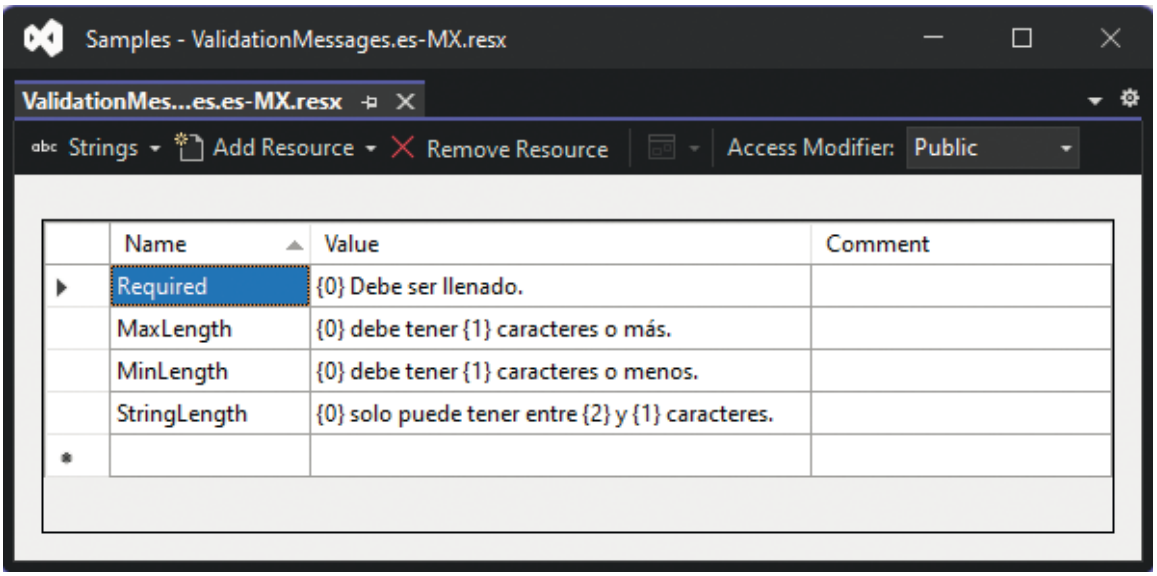


Figure 13: Create a resource file for each language you need to support in your project.

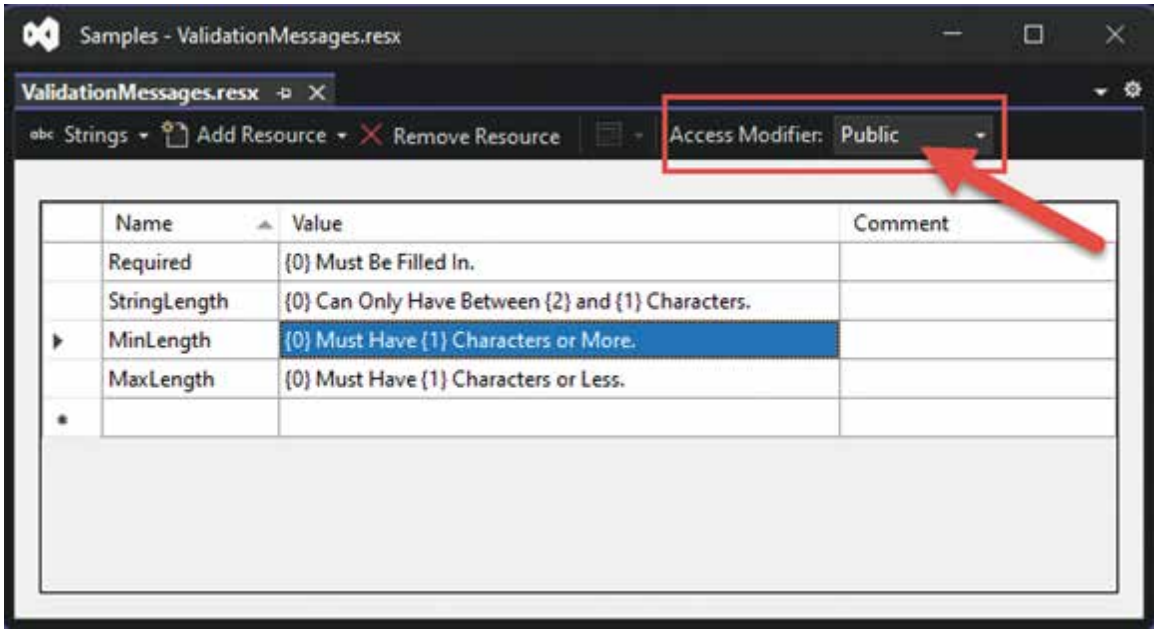


Figure 12: Change the Access Modifier on the resources file to Public to ensure that you can use the messages.

```
  ErrorMessageResourceType =
    typeof(ValidationMessages))]
public string Name { get; set; }
```

Next, modify the [MaxLength] attribute on the **ProductNumber** property to look like the following code:

```
[MaxLength(25, ErrorMessageResourceName =
  nameof(ValidationMessages.MaxLength),
  ErrorMessageResourceType =
    typeof(ValidationMessages))]
public string ProductNumber { get; set; }
```

Finally, modify the [MinLength] property on the **Color** property to look like the following:

```
[MinLength(3, ErrorMessageResourceName =
  nameof(ValidationMessages.MinLength),
  ErrorMessageResourceType =
  typeof(ValidationMessages))]
public string Color { get; set; }
```

Try It Out

Open the **Program.cs** file and instantiate the ProductViewModel class to look like the following:

```
ProductViewModel vm = new() {
  Entity = new() {
    ProductID = 1,
    Name = "",
    ProductNumber = "A very long product
      number to show[MaxLength] Attribute",
    Color = "it",
    StandardCost = 2,
    ListPrice = 5,
    SellStartDate = DateTime.Today,
    SellEndDate = DateTime.Today.AddDays(+1),
    DiscontinuedDate =
      Convert.ToDateTime("1/1/2023")
  }
};
```

Run the application to see the English language messages from the resource file appear.

Modify the Culture on the Current Thread

Open the **Program.cs** file and add a using statement at the top of this file.

```
using System.Globalization;
```

Add a new variable named **culture** to set the current language.

```
string culture = "en-US";
```

Just before the call to the vm.Validate() method, set the current UI culture to the string contained in the **culture** variable. The culture set on the CurrentUICulture thread is the one used to determine which resource file to use.

```
Thread.CurrentThread.CurrentUICulture =
  new CultureInfo(culture);
```

Run the application again and you should see the same English language messages appear. After viewing the error messages, modify the **culture** variable to "es-MX" as shown below:

```
string culture = "es-MX";
```

Run the application again to see the Spanish language messages appear.

Modify the User Class to Work with Resources

In the User class, you added the [EmailAddress] and the [Phone] attributes. You should add the appropriate error messages for those two attributes to each of your re-

Name	Value
Email	The {0} Field is not a Valid Email Address.
Phone	The {0} Field is not a Valid Phone Number.

Table 3: Add resources for the [EmailAddress] and [Phone] attributes to the ValidationMessages.resx file

Name	Value
Email	El campo {0} no es una dirección de correo electrónico válida.
Phone	El campo {0} no es un número de teléfono válido.

Table 4: Add resources for the [EmailAddress] and [Phone] attributes to the ValidationMessages.es-MX.resx file

source files. Open the **ValidationMessages.resx** file and add the name/value pairs shown in **Table 3**.

Open the **ValidationMessages.es-MX.resx** file and add the name/value pairs shown in **Table 4**.

Open the **User.cs** file and add a using statement at the top of the file to bring in the namespace where the **ValidationMessages** class is located.

```
using Samples.Resources;
```

Add a [Required] attribute to the **LoginId** property to look like the following code:

```
[Required(ErrorMessageResourceName =
    nameof(ValidationMessages.Required),
  ErrorMessageResourceType =
    typeof(ValidationMessages))]
public string LoginId { get; set; }
```

Modify the [EmailAddress] attribute on the **EmailAddress** property to look like the code shown below:

```
[EmailAddress(ErrorMessageResourceName =
    nameof(ValidationMessages.Email),
  ErrorMessageResourceType =
    typeof(ValidationMessages))]
public string EmailAddress { get; set; }
```

Modify the [Phone] attribute on the **Phone** property as shown in the code below:

```
[Phone(ErrorMessageResourceName =
    nameof(ValidationMessages.Phone),
  ErrorMessageResourceType =
    typeof(ValidationMessages))]
public string Phone { get; set; }
```

Try It Out

Open the **Program.cs** file and instantiate the UserViewModel to look like the following:

```
UserViewModel vm = new() {
  Entity = new() {
    UserId = 1,
    LoginId = "",
    Password = "Joe!Smith@2022",
    ConfirmPassword = "Joe!Smith@2022",
    EmailAddress = "test!test.com",
    Phone = "asfsadf"
  }
};
```

Be sure to set the **culture** variable back to "en-US" and run the application to see the English language messages from the resource file appear. After running the English version, change the **culture** variable to "es-MX" and run the application to see the Spanish language version of the error messages.

Summary

In this article, you learned about many of the data annotations available in .NET. There are more annotations than what I covered in this article, but I covered the ones you're going to use most often. There's great functionality that you get out of the data annotations available from Microsoft, but if they don't cover your needs, it's very easy to build your own validation attributes. The custom attributes illustrated in this article should provide you with a good start. Take advantage of resource files, even if you're not doing multilingual applications. They're great for ensuring that all your error messages stay consistent from one class to another.

Paul D. Sheriff

DEVELOPERWEEK™

2023
2023
2023

Feb 15-17 Oakland, CA **Feb 21-23** Virtual

8,000+ Developers at the World's Largest Developer Expo & Conference Series

250+ Speakers

Todd Sharp
Principal Developer Advocate
twitch
2023 Speaker

250+ Speakers across 15+ technical tracks, including:

JavaScript • AI/ML • Kubernetes • APIs • Microservices • Product • Python • Databases • Dev Tools • Dev Management/Leadership

Use code **MP108** for **$200** off any conference. Register at **developerweek.com**

DEVELOPERWEEK GLOBAL SERIES

7 events. **5,000+** companies. **25,000+** annual attendees.

DEVELOPERWEEK EUROPE
26-27 April, 2023

DEVELOPERWEEK LATIN AMERICA
June 21-22, 2023

Dev Innovation Summit
August 16-17, 2023

DEVELOPERWEEK MANAGEMENT
May 9-10, 2023

DEVELOPERWEEK CLOUD
August 16-17, 2023

DEVELOPERWEEK ENTERPRISE
November 15-16, 2023

Mastering Routing and Middleware in PHP Laravel

In this article, I'll cover the anatomy of handling an incoming HTTP request. I'll also uncover how Laravel internally runs all kinds of middleware to ensure that it generates a response for the request. Finally, I'll look at how it generates the response, down to the client that initially requested the resource. This new article on PHP Laravel is part of a series I launched a few issues back

Bilal Haidar
bhaidar@gmail.com
https://www.bhaidar.dev
@bhaidar

Bilal Haidar is an accomplished author, Microsoft MVP of 10 years, ASP.NET Insider, and has been writing for CODE Magazine since 2007.

With 15 years of extensive experience in Web development, Bilal is an expert in providing enterprise Web solutions.

He works at Consolidated Contractors Company in Athens, Greece as a full-stack senior developer.

Bilal offers technical consultancy for a variety of technologies including Nest JS, Angular, Vue JS, JavaScript and TypeScript.

to cover developing MVC applications with PHP Laravel. If you haven't had a chance to read the first three tutorials, you can access them through these links:

- Building MVC Applications in PHP Laravel: Part 1 (https://www.codemag.com/Article/2205071/Building-MVC-Applications-in-PHP-Laravel-Part-1)
- Building MVC Applications in PHP Laravel: Part 2 (https://www.codemag.com/Article/2207041/Building-MVC-Applications-in-PHP-Laravel-Part-2)
- Service Containers in PHP Laravel (https://www.codemag.com/Article/2212041/Dependency-Injection-and-Service-Container-in-Laravel)

One component often overlooked or perceived as part of the **C** (controller) in MVC is the routing engine. It's the brains behind the MVC pattern. It's one of the most important components in MVC that initially receives the request from the client and allocates which controller will handle it.

The brains behind routing in Laravel is the **app\Providers\RouteServiceProvider.php** file. This service provider reads and loads all route configuration files you've defined in your application and makes all the route configurations ready to serve the next HTTP Request.

Middleware, on the other hand, is as important as a routing engine. Through middleware, you can inspect and take action before and/or after a specific request is handled and processed by Laravel. For instance, you can have middleware that runs for a specific route, the members' area route, to check whether the user trying to access this route is indeed a member. Accordingly, the middleware can take action to either redirect the user to the membership page to become a member or allow the request to go through for a registered member.

The Anatomy of a Laravel Request Life Cycle

Let's go through the anatomy of a typical Laravel HTTP request and study all the major components involved in processing this request.

Figure 1 highlights the components that I'll be discussing.

Let's go through the stages involved in **Figure 1**:

- The browser (client) requests a resource (View or API endpoint).
- Laravel bootstraps the application.
- The HTTP kernel, a component managed by Laravel, passes the incoming request through a series of Global middleware. Global middleware applies to any incoming HTTP request. That middleware is defined inside the **app\Http\Kernel.php** file through the **$middleware** variable.
- The kernel collects all of the global middleware and runs them one by one. It runs the **Before Global Middleware** first. I'll cover the Before and After Middleware later in the article.
- The kernel then hands off the request to the Illuminate\Routing\Router class.
- The router collects all middleware defined on the route and runs them individually. It runs the **Before Middleware** first.
- Right after running all Before Middleware, the router matches the incoming request with the suitable Controller and Action function and hands off the execution of the request to the Controller to generate a proper response.
- The router then runs all the **After Middleware** on the resulting response from the previous step.
- The kernel collects all of the global middleware and runs them one by one. It runs the **After Global Middleware** now.
- Laravel finally sends down the response to the requesting client. This step ends the request on the server side.

> To process a Request, Laravel runs Global, Route Group, and Route-specific middleware.

This is, in brief, how Laravel handles an incoming request.

Middleware in Laravel

Anything that runs on a server between receiving an incoming HTTP request and sending out a response is technically middleware.

Middleware is the glue between requests and responses. When a new request hits your application, Laravel passes this request through a series of middleware before generating a response and sending it back to the browser.

Let's discuss the types of middleware you can have in Laravel.

Before Middleware

A middleware in Laravel can have two phases: before and after. The before-phase is the code Laravel runs against

the incoming request before it handles it inside a controller/action.

Here's an example of how to write a Before Middleware in Laravel.

```
class BeforeMiddleware implements Middleware {
    public function handle(
      $request, Closure $next
    )
    {
        // Do stuff here

        return $next($request);
    }
}
```

The **BeforeMiddleware** has a **handle()** function that includes your custom code and ends with a call to the next middleware in the chain of middeware that Laravel should run.

After Middleware

The middleware after phase is the code Laravel runs against the response generated by executing the matching controller/action. In this context, the after phase inspects the response rather than the request and adds additional headers, for example.

Here's an example of how to write an After middleware in Laravel.

```
class AfterMiddleware implements Middleware {
    public function handle(
      $request,
      Closure $next
    )
    {
        $response = $next($request);

        // Do stuff here

        return $response;
    }
}
```

The **AfterMiddleware** has a **handle()** function that lets Laravel first execute the incoming request and generate a response. The code stores the response in the **$response** variable. Then, the middleware can inspect the generated response and add more things to it before returning the final response to the next middleware in the chain.

Full Middleware

You can also combine before and after phases both in a single middleware as follows:

```
class AllPhasesMiddleware implements Middleware {
    public function handle(
      $request,
      Closure $next
    )
    {
        // Do stuff here
        $response = $next($request);
        // Do stuff here
        return $response;
    }
}
```

In this case, the **AllPhasesMiddleware** runs some code before generating a response from the incoming request. It then inspects the resulting response to add more things and returns this response to the next middleware in the chain.

> Middleware has two phases: a Before phase that occurs before processing the request and an After phase that occurs after running the controller/action.

Types of Middleware

In Laravel, there are three different types of middleware:

- Global middleware
- Route Group middleware
- Route middleware

Global middleware is defined inside the **app\Http\Kernel.php** file. Open this file and locate the $middleware array variable.

```
protected $middleware = [
    ...\TrustProxies::class,
    ...\HandleCors::class,
    ...\PreventRequestsDuringMaintenance::class,
```

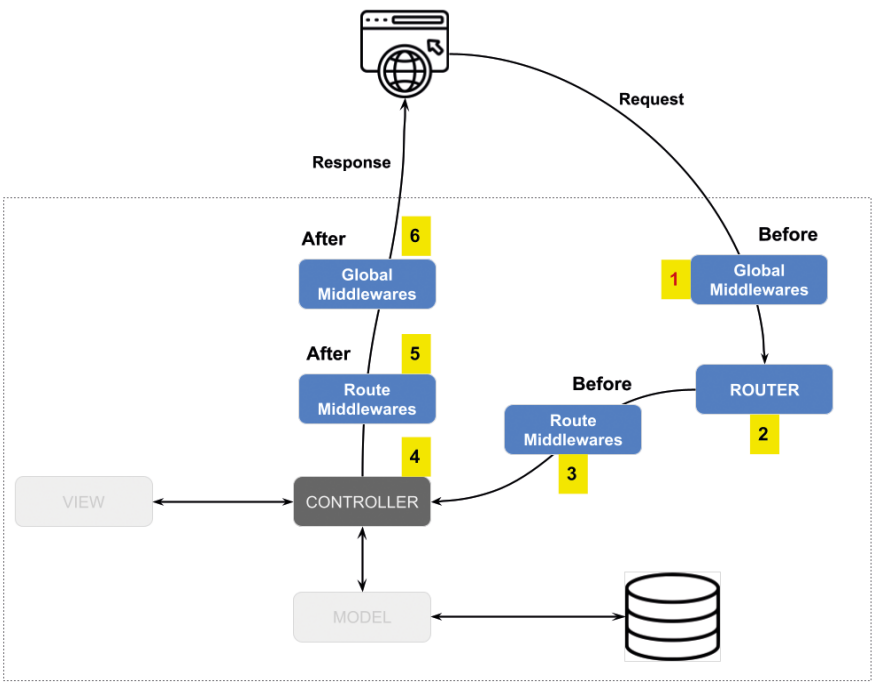


Figure 1: Laravel internal components serving a single HTTP request

```
    ...\ValidatePostSize::class,
    ...\TrimStrings::class,
    ...\ConvertEmptyStringsToNull::class,
];
```

In the next article of this series, I'll dig deeper and show you how the different types of midleware run inside the Laravel framework. For now, it's important to familiarize yourself with the different types of middleware available in Laravel.

The route group middleware is also defined inside the same kernel file. Open this file and locate the **$middlewareGroups** array variable.

Listing 1 shows the route middleware groups.

In the Router section later in this article, I'll discuss the route middleware groups in depth. As you'll see shortly, the route middleware is assigned to specific routes or a group of routes.

You can read more about Laravel middleware here (https://laravel.com/docs/9.x/middleware).

The Router Object

Routes are the stepping stone in the lifecycle of MVC applications. Whenever you open an existing Laravel application, you should first open its routes and study them. Learning about the application's routes makes it very clear what functionality it offers and how to access all of its features.

A route in Laravel is a bridge that connects the outside world to your source code by mapping an incoming HTTP request to either a Closure or a Controller Action to return a response to the client.

For those following this series on building MVC applications with PHP Laravel, open the sample Laravel application you built in the first two episodes.

Let's define a basic route in Laravel. Go to the **routes/web.php** file and add the following route definition:

```
Route::get('/', function () {
    return view('welcome');
});
```

In this code snippet, you're defining a GET request mapping. Whenever you visit the application's root path, represented by a forward slash, the Closure runs and returns the **welcome.blade.php** view to the browser.

Listing 1: Route middleware groups

```
protected $middlewareGroups = [
    'web' => [
        ...\EncryptCookies::class,
        ...\AddQueuedCookiesToResponse::class,
        ...\StartSession::class,
        ...\ShareErrorsFromSession::class,
        ...\VerifyCsrfToken::class,
        ...\SubstituteBindings::class,
    ],
    'api' => [
        'Throttle:api',
        ...\SubstituteBindings::class,
    ],
];
```

The Route object corresponds to the Illuminate\Support\Facades\Route facade. The Illuminate\Routing\Router object backs up this facade. The Router object, as I'll explore in a future episode of this series on Advanced Routing in Laravel, is responsible for running the incoming HTTP request through a pipeline of middleware until it generates a response and returns it to the browser.

Not only can you define GET HTTP requests, but all well-known HTTP methods are available, such as GET, POST, PATCH, DELETE, etc. Here's a full list of HTTP methods to study: (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

Let's explore routes in more depth now.

Defining Routes

To define a route, you need first to understand what functionality you're building. For example, in an application to manage real estate properties, one of the modules an admin needs is to manage real estate properties.

An admin needs to:

- View a list of all real estate properties
- View the details of single real estate property
- Add a new real estate property
- Update the details of existing real estate property
- Delete existing real estate property

Now that you understand what functionality you need, let's open a Laravel application and define all those routes the Laravel way.

First, you'll define the routes manually. In the next section, you'll learn about route resources.

Open the **routes/web.php** file once again and paste the following collection of route definitions:

Get a list of all properties:

```
Route::get(
    '/properties',
    [PropertyController::class, 'index']
)->name('properties.index');
```

Get a single property:

```
Route::get(
    '/properties/{property}',
    [PropertyController::class, 'show']
)->name('properties.show')->middlware('auth');
```

Get Property Create Form:

```
Route::get(
    '/properties/create',
    [PropertyController::class, 'create']
)->name('properties.create')->middlware('auth');
```

Post or create a new property:

```
Route::post(
   '/properties',
   [PropertyController::class, 'store']
)->name('properties.store')->middlware('auth');
```

Get Property Update Form:

```
Route::get(
   '/properties/{property}/edit',
   [PropertyController::class, 'edit']
)->name('properties.edit')->middlware('auth');
```

Put or update an existing property:

```
Route::put(
   '/properties/{property}',
   [PropertyController::class, 'update']
)->name('properties.update')->middlware('auth');
```

Delete an existing property:

```
Route::delete(
   '/properties/{property}',
   [PropertyController::class, 'destroy']
)->name('properties.destroy')->middlware('auth');
```

You've used other HTTP methods than the GET. You use POST to create a new property, the PUT to update an existing property, and the DELETE to delete or destroy an existing property.

Laravel defines those standard seven routes to manage a single Model. You can represent any route in Laravel using one of those seven routes.

Previously, you've seen how to map an HTTP method with a URL path to a PHP closure. In this list of routes, you map a route to a Controller and Action function to generate the response.

Implementing Controllers

Let's go through each route and define the corresponding controller/action function.

Something common you'll notice that applies to all routes is that the user needs to be authenticated to access those routes. This is done by specifying the **middleware('auth')** middleware.

Index Route

Let's take the first route:

```
Route::get(
   '/properties',
   [PropertyController::class, 'index']
)->name('properties.index');
```

The route defines the following: The PropertyController class, specifically the **index()** function, handles an HTTP GET request on the **/properties** path; The **PropertyController::index** function is responsible for generating a response for this request; Also, you're giving the name of **properties.index** for this specific route. You can use the route name to redirect the user to a specific route instead of using the actual URL. This URL might change in the future for any number of reasons. However, the name stays the same.

```
public function index()
{
   $properties = RealEstateProperty::all();
   return view(
      'property.index',
      ['properties' => 'properties']
   );
}
```

The index() function retrieves all the real estate properties in the database and returns the **resources/views/property/index.blade.php**.

You may return a JSON response, a file, or any other data from your controller actions.

As a refresher, you can create the **RealEstateProperty** model and the **PropertyController** class by running the following commands consecutively.

```
php artisan make:model RealEstateProperty -m
php artisan make:controller PropertyController
```

The **-m** flag generates a migration file to create the corresponding real_estate_properties database table.

Show Route

Now let's implement the action functions for the next route definition.

```
Route::get(
   '/properties/{property}',
   [PropertyController::class, 'show']
)->name('properties.show')->middlware('auth');
```

The **properties.show** named route defines a mapping between a GET request for the URL **/properties/{property}** and the handler of that request, the **PropertyController::show action** function.

However, this specific route defines the **URL: /properties/{property}**. The Router is capable of handling URL placeholders. Those placeholders are called **Route Parameters**. In this case, the route expects the **property** route parameter. This parameter corresponds to the **Real Estate Property ID** field. To view the details of a specific Real Estate Property, you must add the property ID on the URL.

Let's check together the **show()** action function.

```
public function show(
   RealEstateProperty $property
)
{
   return view(
      'properties.show',
      compact('property')
   );
}
```

The show() function accepts a Real Estate Property object as an input parameter. Wait—you pass a Real Estate Property ID on the URL and receive an object? Yes! This is another feature offered by the Router object called **Implicit Binding** (https://laravel.com/docs/9.x/routing#route-model-binding). The Router assumes that the router parameter passed

represents the **Key** of a Model object. Then, based on the type of input parameter you assign in the show() function, it queries the database for an object of that type where the object's ID equals the value of the route parameter.

You're not limited to this default behavior. For instance, you may want the route parameter to represent the Model Name. Laravel's got you covered. You can instruct the Model to use a different route Key for implicit binding data retrieval.

Open the **app\Models\RealEstateProperty.php** and add this function:

```
/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'name';
}
```

That's all!

You can read more about route parameters in Laravel here (https://laravel.com/docs/9.x/routing#route-parameters).

Create Route

Now let's implement the action functions for the next route definition.

```
Route::get(
   '/properties/create',
   [PropertyController::class, 'create']
)->name('properties.create')->middlware('auth');
```

The **properties.create** named route defines a mapping between a GET request for the URL **/properties/create** and the handler of that request, the **PropertyController::create action** function.

The admin uses this route to create a new Real Estate Property entry in the database. Let's check the backing **create()** function behind it.

```
public function create()
{
   return view('properties.create');
}
```

Listing 2: store() function source code

```
public function store(Request $request)
{
   $validator = Validator::make($request->all(), [
      'name' => 'required|max:255',
      'location' => 'required',
      'price' => 'required|'integer',
      // …
   ])->validate();

   RealEstateProperty::create($validator->validated());

   return redirect()
      ->route('properties.index')
      ->with('success','Property created successfully.');
}
```

The **create()** function returns a view that allows the admin to fill in the details of a new Real Estate Property.

Store Route

Now let's implement the action functions for the next route definition.

```
Route::post(
   '/properties',
   [PropertyController::class, 'store']
)->name('properties.store')->middlware('auth');
```

The **properties.store** named route defines a mapping between a POST request for the URL **/properties** and the handler of that request, the **PropertyController::store action** function.

This route handles the creation of a new Real Estate Property record in the database. It expects a request payload containing all the record details to create. Let's check the backing **store()** function behind it.

Listing 2 shows the entire source of this function.

It receives the Request object. This object contains the request payload and other useful information.

The function starts by validating the incoming HTTP request by calling the **validated()** function on the Validator object. Using the Validator object, you can define rules to validate the request. For instance, the **name** field should be present with a maximum of 255 characters. You can define whatever validation rules you want to at this stage.

After validating the request payload, you use the model static **create()** function on the RealEstateProperty model object. This function represents mass assignment in Laravel Eloquent and accepts an associative array of the columns you want to populate on the newly created record. Remember, this function works closely with the **$fillable** array property you define on the RealEstateProperty model object. Whatever you allow inside this array will pass through the static create() function.

```
protected $fillable = [
   'name',
   'location',
   'price'
];
```

Based on the current definition of the $fillable array, only name, location, and price fields can be passed through the static create() function.

You can read more about validation in Laravel here: https://laravel.com/docs/9.x/validation.

After creating the new record in the database, the store() function redirects the admin back to the index page to list all the Real Estate Property records, passing along a **success** message to notify the admin that the operation has succeeded. Notice how the function redirects to a **named route** rather than specifying the URL. This is why adding names to the routes makes your application flexible and maintainable.

Another way of doing validation is by using **FormRequests** in Laravel. This object extends the HTTP request object. I can't dig deeper into FormRequests right now, but, in brief, you can define a new FormRequest that contains all the validation rules. Then, instead of receiving an instance of the Request object, you'll receive an instance of the FormRequest object. Laravel automatically runs the validation code for you before even calling the action function.

You can read more about FormRequests in Laravel here: https://laravel.com/docs/9.x/validation#form-request-validation.

Edit Route

Now let's implement the action functions for the next route definition.

```
Route::get(
   '/properties/{property}/edit',
   [PropertyController::class, 'edit']
)->name('properties.edit')->middlware('auth');
```

The **properties.edit** named route defines a mapping between a GET request for the URL **/properties/{property}/edit** and the handler of that request, the **PropertyController::edit action** function. It has the property route parameter to identify which property to edit.

The admin uses this route to edit an existing Real Estate Property entry in the database. Let's check the backing **edit()** function behind it.

```
public function edit(
   RealEstateProperty $property
)
{
   return view(
      'properties.edit',
      ['property' => $property]
   );
}
```

The edit() action function is fairly simple. It receives a Real Estate Property object via the Router's implicit binding feature. It returns the Edit View together with the Real Estate Property object. The view uses this object to populate the Form for the admin to edit.

Update Route

Now let's implement the action functions for the next route definition.

```
Route::put(
   '/properties/{property}',
   [PropertyController::class, 'update']
)->name('properties.update')->middlware('auth');
```

The **properties.update** named route defines a mapping between a PUT request for the URL **/properties/{property}** and the handler of that request, the **PropertyController::update action** function.

This route handles updating an existing Real Estate Property record in the database. It defines the Real Estate Property ID and expects a request payload containing the updated record details. Let's check the backing **update()** function behind it.

Listing 3 shows the entire source of this function. It receives the Request object and the implicitly bound Real Estate Property object.

The function starts by validating the incoming HTTP request by calling the **validated()** function on the Validator object. It then uses the **update()** function to perform the underlying update on the database.

After updating the record in the database, the update() function redirects the admin back to the index page to list all the Real Estate Property records, passing along a **success** message to notify the admin that the operation has succeeded.

Delete Route

Now let's implement the action functions for the last route definition.

```
Route::delete(
   '/properties/{property}',
   [PropertyController::class, 'destroy']
)->name('properties.destroy')->middlware('auth');
```

The **properties.destroy** named route defines a mapping between a DELETE request for the URL **/properties/{property}** and the handler of that request, the **PropertyController::destroy** action function.

This route handles deleting an existing Real Estate Property record from the database. It has the property route parameter to identify which property to delete. Let's check the backing **destroy()** function behind it.

```
public function destroy(
   RealEstateProperty $property
)
{
   $property->delete();

   return redirect()->route('properties.index')
     ->with(
          'success',
          'Property deleted successfully'
      );
}
```

Listing 3: update() function source code

```
public function update(
   Request $request,
   RealEstateProperty $property
)
{
   $validator = Validator::make($request->all(), [
      'name' => 'required|max:255',
      'location' => 'required',
      'price' => 'required|'integer',
      // …
   ])->validate();

   $property->update($validator->validated());

   return redirect()
      ->route('properties.index')
      ->with('success','Property updated successfully.');
}
```

Listing 4: RouteServiceProvider booth() method source code

```
public function boot()
{
   // …

   $this->routes(function () {
      Route::middleware('api')
         ->prefix('api')
         ->group(base_path('routes/api.php'));

     Route::middleware('web')
         ->group(base_path('routes/web.php'));
   });
}
```

It receives the implicitly bound Real Estate Property object as an input parameter.

The function starts by deleting the Real Estate Property object by calling the **delete()** function. It then redirects the admin to the index page to list all the Real Estate Property records, passing along a **success** message to notify the admin that the operation has succeeded.

Route Resources

The concept of a Resource stems from the REST architecture that positions a resource at the core of an API. All the operations you perform on the application are eventually performed against a Resource. You can read more about REST here: https://restfulapi.net/.

In Laravel, if you think of an Eloquent Model as a resource, you might perform the same set of operations on the resource. Therefore, Laravel has a shorthand representation that simplifies defining routes.

All of the routes defined under the section **Defining Routes** could be replaced by a single-line route resource, as follows:

```
Route::resource(
   'properties',
   PropertyController::class
);
```

That's it! The static **resource()** method internally defines the Laravel standard seven routes for you.

For this to work smoothly, you need to use the following command to create the PropertyController object.

```
php artisan make:controller \
   PropertyController --resource
```

I've added the **--resource** flag to signal to Laravel to generate a function for each resource operation.

In some cases, you don't need all of the seven routes. You can define whatever resource operation you need in two different ways:

```
Route::resource(
   'properties',
   PropertyController::class
)->only(['index', 'show']);
```

You use the **only()** function to specify what resource operations you want to define.

The other way around is to define what resource operations to exclude:

```
Route::resource(
   'properties',
   PropertyController::class
)->except(['create', 'store', 'edit', 'update']);
```

Finally, when using route resources and you want to define additional routes to the standard seven routes, you **must place the additional routes before the route resource definition**. Otherwise, you'd be overwriting the resource routes.

For instance, I want to define a mass deletion route for multiple real estate properties. The way I do it is as follows:

```
Route::delete(
   'posts/destroy',
   [PropertyController::class, 'massDestroy']
);

Route::resource(
   'properties',
   PropertyController::class
);
```

Another feature that you'll find useful to use in your projects is the nested resource controllers. You can check this example here: https://laraveldaily.com/post/nested-resource-controllers-and-routes-laravel-crud-example.

There are many more details related to route resources in Laravel that you can read about here: https://laravel.com/docs/9.x/controllers#resource-controllers.

Route Files and Groups

Now that you know about routing in Laravel, let's dig deeper and understand the **routes** folder in Laravel. If you navigate to the **/routes** folder, you'll notice a set of route files:

- **api.php**: This file contains the routes related to a Laravel API. With Laravel, you can build both a web application and an API application.
- **channels.php**: This file contains routes for broadcasting. Laravel supports broadcasting and real-time processing.
- **web.php**: This file contains the routes related to the web application.

Now that you have all those route files, how does Laravel load them into the application and make them available for the Router to match later on? The answer lies inside the **app\Providers\RouteServiceProvider.php** file.

Locate this file and check the boot() method. **Listing 4** shows the boot() method source code.

The method starts by calling the **$routes** method defined on the parent **Illuminate\Foundation\Support\Providers\RouteServiceProvider.php** file. It passes a Closure to

this method. Later on, Laravel executes this Closure and loads all of the routes in the application.

The first statement in the Closure loads the routes defined inside the **routes/api.php** file. The second one loads the routes defined inside the routes/web.php file.

In both statements, Laravel groups the routes under each file separately. It assigns a prefix for each group, and finally, it applies the route middleware group **api** and **web** consecutively.

Therefore, the routes defined inside the routes/api.php file are prefixed by the string **api**.

Let's open the routes/api.php file and see what default content it has:

```
Route::middleware('auth:sanctum')
   ->get('/user', function (Request $request) {
      return $request->user();
   });
```

It defines a single GET request for **/user**. The handle of this request is a Closure that returns the logged-in user. The **auth:sanctum** middleware is applied to this route, meaning only authenticated users can access this route.

The URL corresponding to this route is: **/api/user** rather than /user. That's because of the prefix applied to all routes inside this file.

Going back to the boot() function Closure, it assigns a route middleware group to both route files.

At the end of this article, you will learn more about middlewares in Laravel. Meanwhile, let's open the **app\Http\Kernel.php** file and locate the **$middlewareGroups** array variable shown in **Listing 1**.

This is an associative array with two keys only: api and web. Each key in this array represents an array of middleware classes. We call those routes middleware in Laravel. The $middlewareGroups we call the route middleware groups.

Laravel runs and executes the middleware defined under the group **web** for all the routes defined inside the **routes/web.php** file. Similarly, it executes the middleware defined under the group api for all the routes defined inside the **routes/api.php** file.

Custom Route File

In addition to the default route files, you can also have your route files. Sometimes, you need to group some routes separately so that you don't jam your web.php file. For instance, I usually keep all the admin route definitions in a separate admin.php file.

Follow the steps below to create a custom route file and register it successfully in your app.

1. Create a new route file inside the folder routes. For example, routes\admin.php.
2. Place your routes inside the new file.
3. Instruct Laravel to load this new file.

Listing 5: Load a custom route file

```
public function boot()
{
   // …

   $this ->routes(function () {
      Route::middleware('api')
         ->prefix('api')
         ->group(base_path('routes/api.php'));

      Route::middleware('web')
         ->group(base_path('routes/web.php'));

      Route::middleware('is_admin')
         ->namespace($this ->namespace)
         ->group(base_path('routes/admin.php'));
    });
}
```

Listing 6: IsAdminMiddleware.php source code

```
namespace App\Http\Middleware;

class IsAdminMiddleware
{
   public function handle(
      $request,
      Closure $next
   )
   {
      if( !Auth::check() ) {
         return redirect()->route('login');
      }

      if( Auth::user()->isAdmin == true ) {
         return $next($request);
      }

      return redirect()
         ->back()
         ->with(
            'unauthorized',
            'You are unauthorized to access this page'
         );
   }
}
```

To instruct Laravel to load the new route file, go to **app\Providers\RouteServiceProvider.php** file, and inside the boot() method, add the following line toward the end of the Closure, as shown in **Listing 5**.

Notice that I'm using a new route, the middleware group called is_admin. Let's create this new group. Open the app\Http\Kernel.php file and locate the $routeMiddlewareGroups array variable. Add the following line:

```
protected $middlewareGroups = [
   'web' => [...],
   'api' => [...],

   'is_admin' => [
      // … any middleware here
      App\Http\Middleware\IsAdminMiddleware::class,
   ],
];
```

Next, let's create the new middleware inside the **app\Http\Middleware** folder. **Listing 6** shows the source code for this new middleware.

The middleware's handler method checks whether the user is authenticated. If not, it redirects back to the log in page. Otherwise, it checks whether the user is an admin. Otherwise, it redirects back to the previous page with an error message.

Route Grouping

Let's revisit the list of routes you originally defined for the real estate property management module. **Listing 7** shows all the routes.

Listing 7: All routes

```
Route::get(
   '/properties',
   [PropertyController::class, 'index']
)->name('properties.index')->middlware('auth');

Route::get(
   '/properties/{property}',
   [PropertyController::class, 'show']
)->name('properties.show')->middlware('auth');

Route::get(
   '/properties/create',
   [PropertyController::class, 'create']
)->name('properties.create')->middlware('auth');

Route::post(
   '/properties',
   [PropertyController::class, 'store']
)->name('properties.store')->middlware('auth');

Route::get(
   '/properties/{property}/edit',
   [PropertyController::class, 'edit']
)->name('properties.edit')->middlware('auth');

Route::put(
   '/properties/{property}',
   [PropertyController::class, 'update']
)->name('properties.update')->middlware('auth');

Route::delete(
   '/properties/{property}',
   [PropertyController::class, 'destroy']
)->name('properties.destroy')->middlware('auth');
```

Listing 8: Routes with groups

```
Route::prefix('/properties')
   ->middleware(['auth'])
   ->controller(PropertiesController::class)
   ->name('properties.')
   ->group(function() {

      Route::get('/', 'index')->name('index');

      Route::get('/{property}', 'show')->name('show');

      Route::get('/create', 'create')->name('create');

      Route::post('/', 'store')->name('store');

      Route::get(
         '/{property}/edit',
         'edit'
      )->name('edit');

      Route::put(
         '/{property}',
         'update'
      )->name('update');

      Route::delete(
         '/{property}',
         'destroy'
      )->name('destroy');
});
```

You now understand that you can use Route::resource() whenever you use all seven standard routes. What if you don't want to use the resource routes? Can you improve the routes and minimize repetitions?

A few things to note about those routes:

- They all require the Auth middleware.
- They all share the same Controller.
- They all share the same prefix of **/properties.**

You could rewrite the above routes as shown in **Listing 8**.

Route Groups in Laravel are so powerful in beautifying the routes and making them more readable. Let's understand the different methods you have used:

- **prefix()**: Use this method to specify a prefix in the route URL.
- **middleware()**: Use this method to specify one or more middleware to apply on this current route.
- **controller()**: Use this method to specify a common Controller to use for all the route handlers.
- **name()**: Use this method to specify a common prefix route name. Notice the dot at the end of the string properties.
- **group()**: Use this method to group all routes with some common behavior and place them all inside a Closure.

I recommend reading this article on Laravel Route Grouping: https://laravel-news.com/laravel-route-organization-tips.

Routing Best Practices

In this section, I'd like to share a video session delivered by Adam Wathan (https://twitter.com/adamwathan), the creator of Tailwind CSS (https://tailwindcss.com/), for Laracon US 2017. In this video, Adam shares some concepts about how to think in routes, decide on the routes your application needs, divide your source code into smaller and more manageable controllers, and much more.

I recommend that you watch this video and take notes to help you decide on your application's routes: Cruddy by Design (https://www.youtube.com/watch?v=MF0jFKvS4SI).

Conclusion

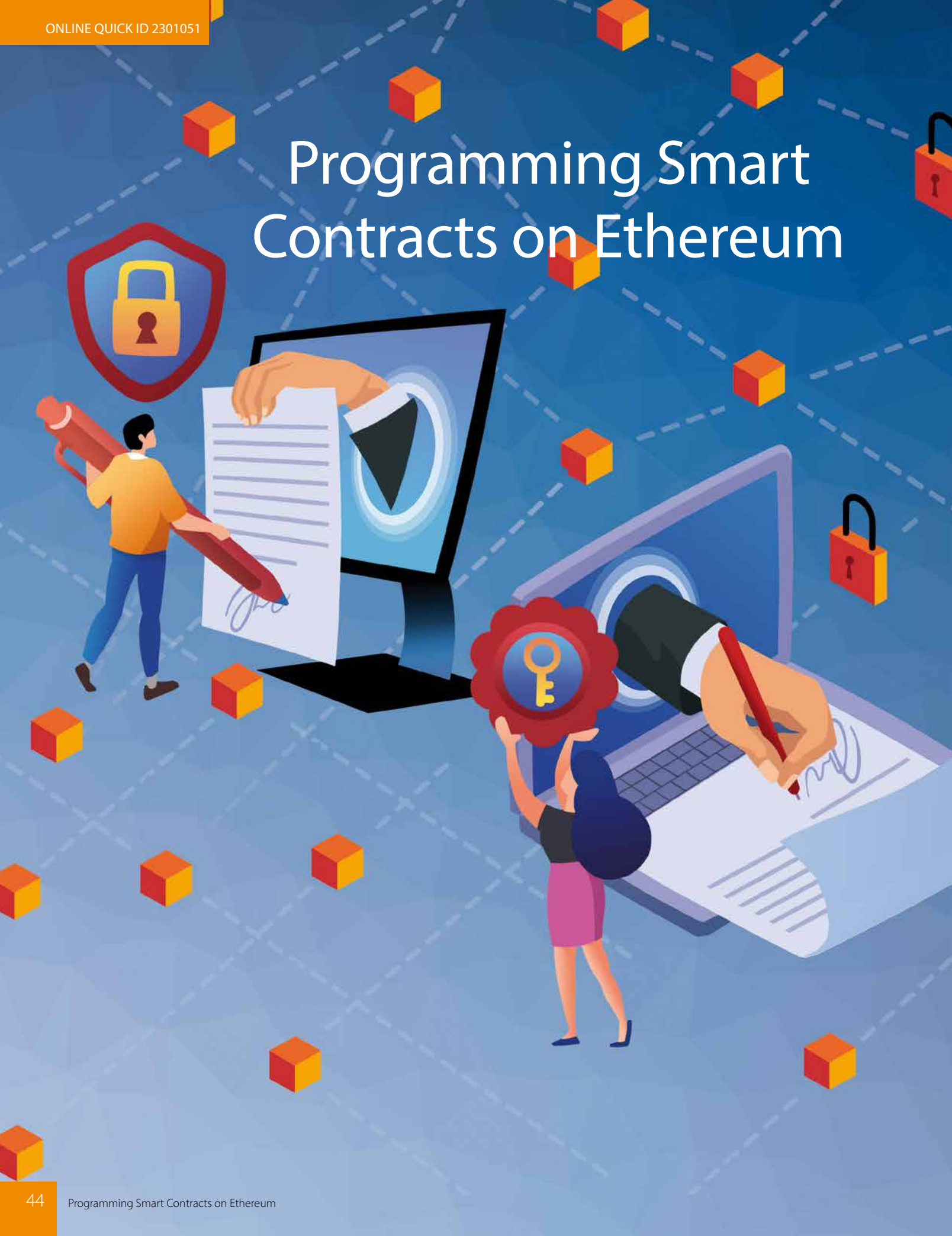
In this fourth article of my series on developing MVC applications with PHP Laravel, I discussed routing and middleware.

The topic of middleware and routing in Laravel is complicated and manyfold. That's why I'll be writing a more in-depth guide, showing how Laravel runs an incoming HTTP request from the beginning until a response is generated and sent back to the browser. During this next article, you'll learn how Laravel runs all kinds of middleware, how the router matches a route, and how it hands off the execution of the incoming request to the matching controller and action function.

Stay tuned and happy Laravelling!

Bilal Haidar

Programming Smart Contracts on Ethereum



Four years ago, I wrote an article entitled "Understanding Blockchain: A Beginners Guide to Ethereum Smart Contract Programming" (https://www.codemag.com/Article/1805061/Understanding-Blockchain-A-Beginners-Guide-to-Ethereum-Smart-Contract-Programming). Since then, a lot of new developments have happened in the blockchain world, and new start-ups touting new tokens are literally popping up every single day. Today, we are bombarded with terms like **DeFi** (Decentralized Finance), **DEX** (Decentralized Exchange), **DAO** (Decentralized Autonomous Organization), **Stable coins**, **tokens**, **ICO** (Initial Coin Offering), **Oracles**, and more. Explaining all these terms is going to take a while, and I'll try to do it in another article.

Behind all these key terms is a key enabling technology: **smart contracts**. My focus for this article is to explain smart contracts in detail so that once you have a good grasp of what smart contracts can do, understanding the other terms becomes a walk in the park.

> If you're not familiar with how blockchain works, I strongly suggest you read my earlier article before continuing with this one.

What's a Smart Contract?

A smart contract is an application residing on the blockchain that can write values onto the blockchain itself. As all the data recorded on the blockchain is immutable and traceable, smart contracts allow you to make use of the blockchain as a backing store for storing data that's permanent (such as student examination results) so that you can use it as a proof later on.

Figure 1 shows the idea behind smart contracts.

Besides storing transactions, a blockchain (such as Ethereum) can also store smart contracts. When a smart contract is invoked (every smart contract deployed onto the blockchain has an address), the smart contract can execute code much like an ordinary application can (although there are several restrictions on what a smart contract can and cannot do). Typically, a smart contract can write values (commonly referred to as **state variables**) onto the blockchain.

Besides writing values onto the blockchain, smart contracts can also transfer funds (such as tokens and cryptos) between accounts. This opens up a lot of possibilities for developers to write interesting applications that run on the blockchain. Decentralized lottery apps, anyone?

Creating a Smart Contract Using Remix IDE

The best way to understand how a smart contract works is to write one. Recall that data stored on the blockchain is immutable. Therefore, blockchain is a good **store-of-proofs**. A good use case is storing the educational credentials of students on the blockchain.

Recently, there have been many cases of job applicants submitting false degree qualifications to employers. Employers have no easy way to verify whether an applicant's qualification is genuine, and they must spend precious time verifying the authenticity of the qualifications. If an institution can record the educational credentials obtained by students onto the blockchain, it's easier for employers (or anyone) to verify the authenticity of potential employees' qualifications.

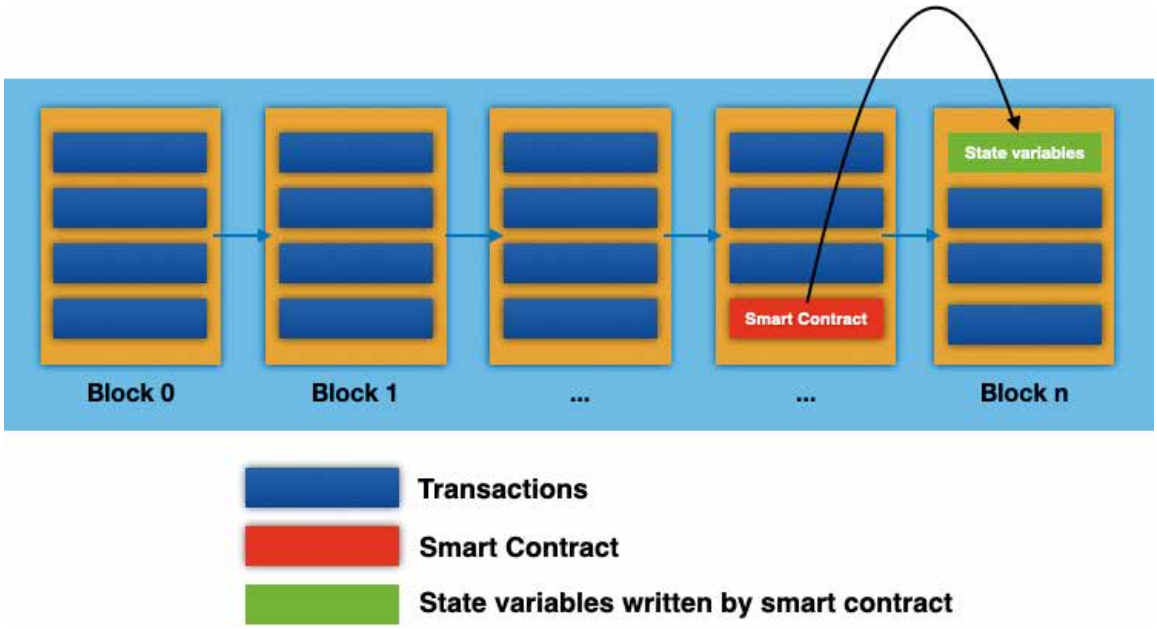


Figure 1: Besides storing transactions, a blockchain also contains smart contracts, which can write values onto the blockchain itself

Wei-Meng Lee
weimenglee@learn2develop.net
www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experiences and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

Consider the following simplified example of a student's examination result represented as a JSON string:

```
{
  "id": "1234567",
  "result": {
    "math": "A",
    "science": "B",
    "english": "A"
  }
}
```

The idea is to store the result onto the blockchain. However, it's not advisable to store the JSON string directly onto the blockchain for two key reasons:

- Storing the plain text onto the blockchain is expensive. Every additional character stored on the blockchain incurs additional gas fees. The aim is to only store only essential data.
- You should never store personally identifiable data on the blockchain. On public blockchains, all data is up for public scrutiny, and hence storing the result in plain text violates privacy regulations and laws.

Considering the privacy concerns, you should instead store the hash of the credentials onto the blockchain.

> Never store encrypted data on the blockchain as they're susceptible to hacking. Storing the hash of the data is much safer.

To verify that the education credentials are authentic, pass in the JSON string representing the result and check whether the hash exists on the blockchain. If it exists, the result is authentic.

> The Goerli testnet is the testnet you should use after the Merge (where Ethereum transitioned from Proof-of-Work to Proof-of-Stake). Other testnets, such as Ropsten, have since been deprecated.

For operational reasons, I'll first encode the JSON string using base64 encoding before obtaining its hash. To try base64 encoding, you can use the following site: https://codebeautify.org/json-to-base64-converter. The above JSON string yields the following base64 encoded output:

```
ewogICJpZCI6ICIxMjM0NTY3IiwKICAicmVzdWx0Ijogewog
ICAgIm1hdGgiOiAiQSIsCiAgICAic2NpZW5jZSI6ICJCIiwK
ICAgICJlbmdsaXNoIjogIkEiCiAgfQp9Cg==
```

For this example, I'm going to use the Ethereum blockchain. Rather than deal with the **mainnet** (the main Ethereum network where you have to use real **ethers**--the cryptocurrency for the Ethereum blockchain), I'm going to use the Goerli testnet.

Using the testnet allows you to test all the various features of the Ethereum blockchain without using real ethers. Also, for this article, I'll be using **Metamask** (an Ethereum crypto-wallet) on Chrome.

> If you're not familiar with Metamask, check out my article https://levelup.gitconnected.com/blockchain-series-getting-started-with-metamask-46fb51a52c3.

Once Metamask is set-up on Chrome, you need to obtain some test ethers. You can get some from https://goerli-faucet.com/.

> You need to setup a free Alchemy (https://www.alchemy.com/) account in order to request free Goerli test ethers.

To create the smart contract, I'll use the **Remix IDE** (https://remix.ethereum.org/). Be sure to load the Remix IDE using Chrome (or whatever browser has the Metamask extension installed). This is to ensure that later on, when you deploy and test your smart contracts, you have a way to pay for the gas (transaction) fees.

> Remix IDE, is a web-based development environment for developing smart contracts. Using Remix IDE, you can compile, deploy, and test your smart contracts, all within your web browser.

When Remix IDE is loaded, you should have something like that shown in **Figure 2**.

Right-click on the **contracts** item and select **New File** (see **Figure 3**).

Name the new file as **EduCredentialsStore.sol** and populate it with the statements as shown in **Listing 1**.

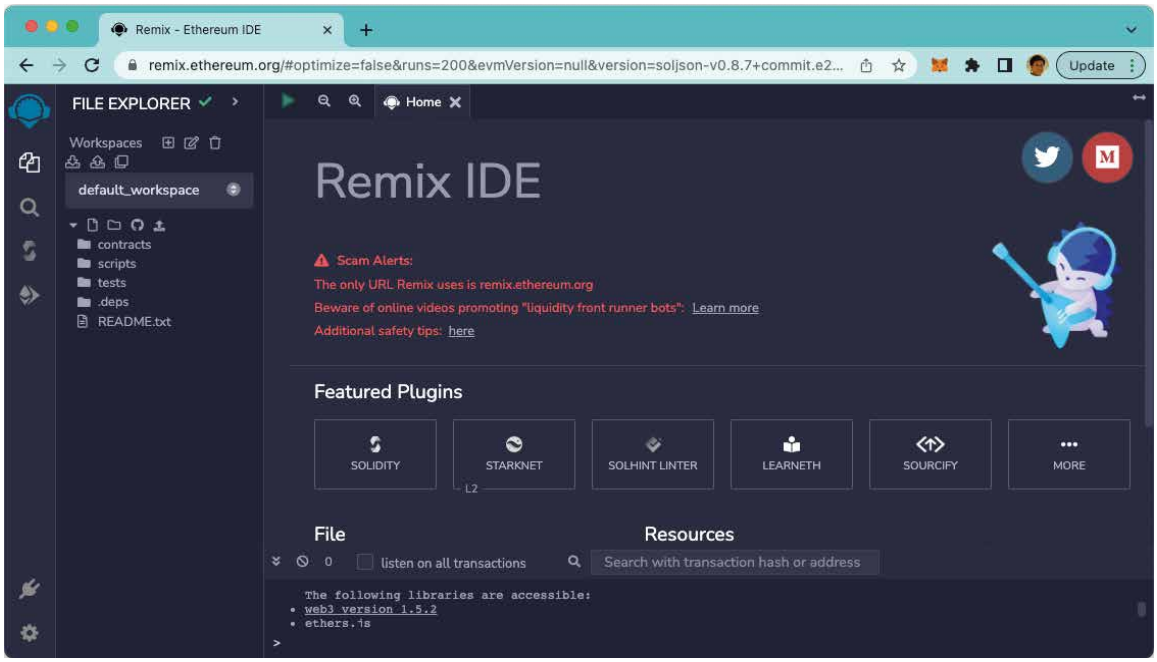


Figure 2: Loading up Remix IDE

Here's a summary of the smart contract (written in the Solidity programming language):

- The first line in the contract specifies the type of license the contract uses. This was introduced in Solidity 0.6.8. The MIT License is a permissive free software license originating at the Massachusetts Institute of Technology in the late 1980s. As a permissive license, it puts only very limited restriction on reuse and has, therefore, high license compatibility.
- The **pragma solidity** statement is a directive that tells the compiler that source code is written for a specific version of Solidity. In this case, it's compatible with version 0.8.x, e.g., versions 0.8.1, 0.8.17, etc. However, it's not compatible with versions such as 0.7 or 0.9.
- The contract has two private functions: **storeProof()** and **proofFor()**. Private functions are denoted with the **private** keyword. Private functions are only callable within the contract and not by users.
- The contract has one **external** function (**storeEduCredentials()**) and one **public** function (**checkEduCredentials()**). Public and external functions are both visible outside the contract and can be called directly by the user. The key difference between **public** and **external** is that public functions are also visible to subclasses of the contract and external functions are not visible to subclasses of the contract.
- You'll notice the use of the **calldata** keyword in the functions parameter declaration. Besides the **calldata** keyword, you can also use the **memory** keyword. Both the **memory** and **calldata** keywords indicate that the parameter holds temporary value, and they won't be persisted onto the blockchain. The key difference between the two keywords is that parameter prefixed with the **calldata** keyword is immutable and cannot be changed, and parameters prefixed with **memory** is mutable.

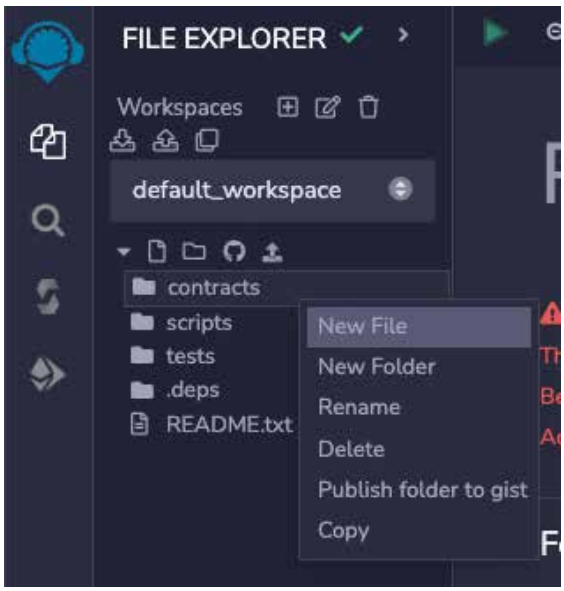


Figure 3: Creating a new contract file in Remix IDE

- You'll also notice that one function has the **pure** keyword and another has the **view** keyword. The **pure** keyword indicates that the function does not read or write values onto the blockchain. The **view** keyword, on the other hand, indicates that the function will read values from the blockchain.
- The contract has a **mapping** object named **proofs**. A **mapping** object in Solidity is like a dictionary object: it holds key/value pairs. The value of **Proofs** is persisted on the blockchain and is known as a **state variable**.

With all the different keywords for function access modifiers and parameters declaration, which should you use? Here's a quick guide:

Listing 1: The smart contract to save a person's educational credentials onto the blockchain

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;

contract EduCredentialsStore {
  //---store the hash of the strings and their
  // corresponding block number---

  // key is bytes32 and val is uint
  mapping (bytes32 => uint) private proofs;

  //-----------------------------------------------
  // Store a proof of existence in the contract state
  //-----------------------------------------------
  function storeProof(bytes32 proof) private {
    // use the hash as the key
    proofs[proof] = block.number;
  }

  //-----------------------------------------------
  // Calculate and store the proof for a document
  //-----------------------------------------------
  function storeEduCredentials(string calldata
  document) external {
    // call storeProof() with the hash of the string
    storeProof(proofFor(document));
  }

  //-----------------------------------------------
  // Helper function to get a document's sha256
  //-----------------------------------------------
  // Takes in a string and returns the hash of the
  // string
  function proofFor(string calldata document) private
  pure returns (bytes32) {
    // converts the string into bytes array and
    // then hash it
    return sha256(bytes(document));
  }

  //-----------------------------------------------
  // Check if a document has been saved previously
  //-----------------------------------------------
  function checkEduCredentials(string calldata
  document) public view returns (uint){
    // use the hash of the string and check the
    // proofs mapping object
    return proofs[proofFor(document)];
  }

}
```

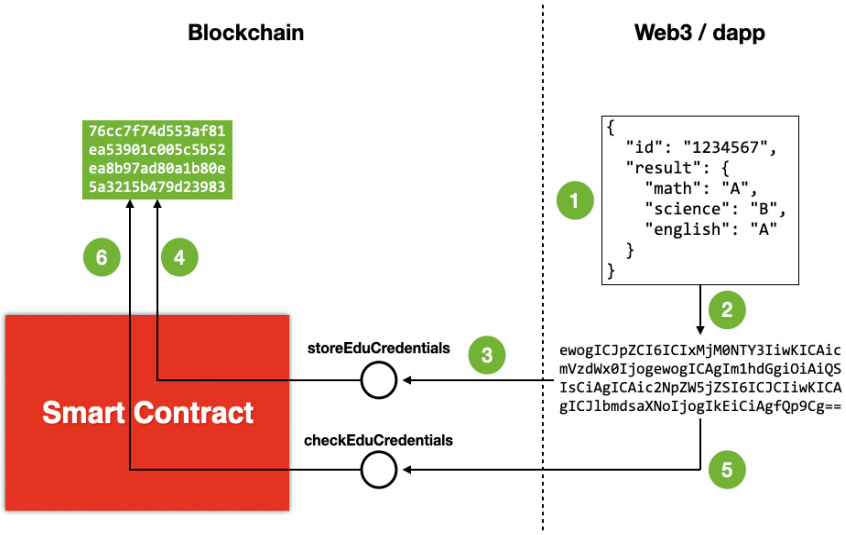


Figure 4: How the smart contract interacts with the user

- Use **calldata** for parameter declaration if the data passed into the function need not be modified. Declaring a function with **calldata** parameters will save on gas fees.
- Use **external** instead of **public** if there's no need for your functions to be called by subclasses of the contract. Due to the way arguments in **public** functions are accessed, declaring functions as **external** will incur lesser gas fees.

Figure 4 shows the conceptual flow of how your smart contract can be used.

Note that:

- The result of a student is represented as a JSON string.
- The JSON string is encoded using base64 encoding.
- The base64 encoded string is passed into the **storeEduCredentials()** function of the smart contract.
- The **storeEduCredentials()** function hashes the base64 encoded string and then stores the hash together with the block number (containing the transaction) onto the blockchain.
- To check whether an educational credential is authentic, the based64 encoded string of the student's result is passed into the **checkEduCredentials()** function.
- The **checkEduCredentials()** function hashes the base64 encoded string and then checks whether the hash already exists on the blockchain. The block number that contains the hash will be returned. If the hash is not found, a value of 0 is returned.

Figure 5 shows how the smart contract stores the hash of the credentials onto the blockchain. The state variable

When smart contracts are deployed to the blockchain, you have to pay for gas fees based on the size of the contract. When users call the functions in a smart contract after it has been deployed, they also need to pay transaction fees, but the fee payable depends on the complexity of the function. Hence when writing your smart contract, it's always good to optimize your code so that the caller can minimize the transaction fees.

is a **mapping** object containing key/value pairs. The keys are the hashes and the values are the block numbers in which the hashes are written onto the blockchain.

Compiling the Contract

With the smart contract written, it's now time to compile it. In Remix IDE, click the **Compiler** tab (see **Figure 6**, step 1) and check **Auto compile**. Doing so enables Remix IDE to automatically compile your code every time you make changes to it.

At the bottom, you'll see two items: **ABI** and **Bytecode**. Clicking on **ABI** copies the Application Binary Interface (ABI) to the clipboard. Paste it onto a text editor and it will look like this:

```
[
  {
    "inputs": [
      {
        "internalType": "string",
        "name": "document",
        "type": "string"
      }
    ],
    "name": "checkEduCredentials",
    "outputs": [
      {
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  },
  {
    "inputs": [
      {
        "internalType": "string",
        "name": "document",
        "type": "string"
      }
    ],
    "name": "storeEduCredentials",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
```

The use of the ABI is to provide details to front-end applications on how to call the smart contract. It contains a description of the external/public functions in the smart contracts, the parameters they accept, and the values they return. When you develop a Web3 dapp (Decentralized App), you need the ABI in order to invoke the contract.

Likewise, click the **Bytecode** button and paste it onto a text editor. It will look like **Figure 7**.

Specifically, the value of the **object** key contains the actual bytecode (think of it as machine language) of the contract that will be deployed onto the blockchain. **Figure 8** shows the steps performed by Remix IDE when it compiles your smart contract.

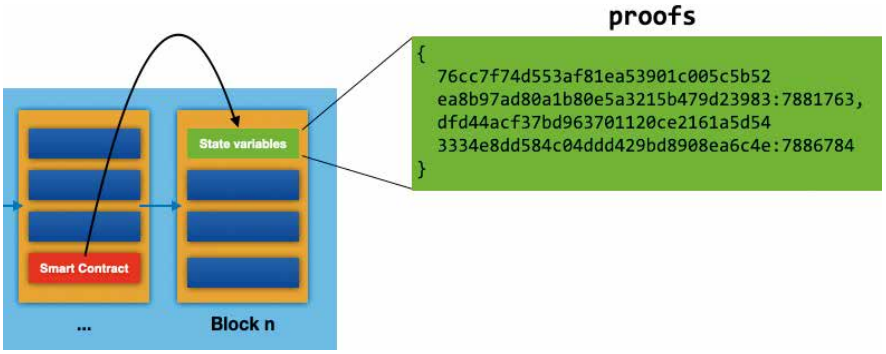


Figure 5: How the mapping object records the hashes and block numbers

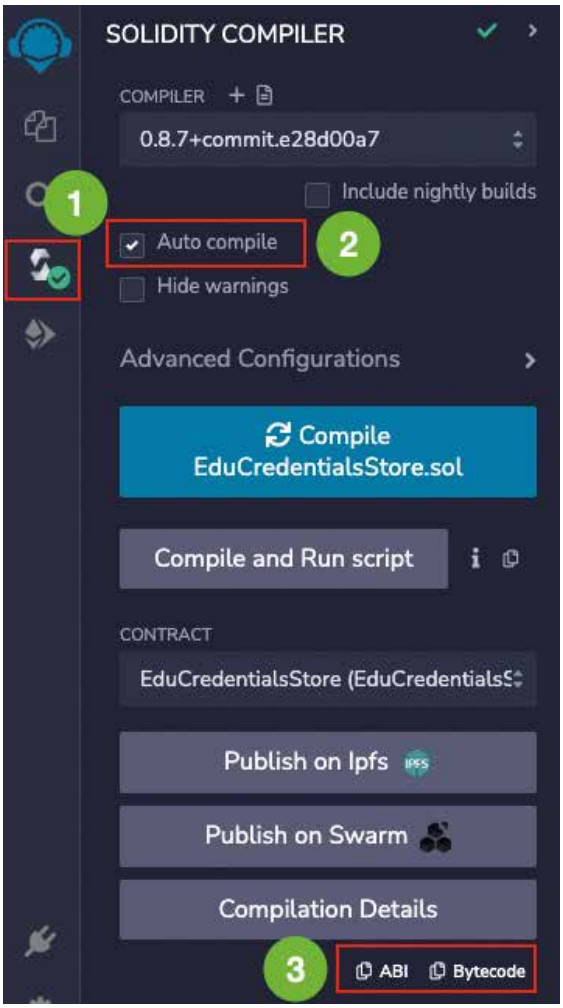


Figure 6: Enabling auto compile in Remix IDE

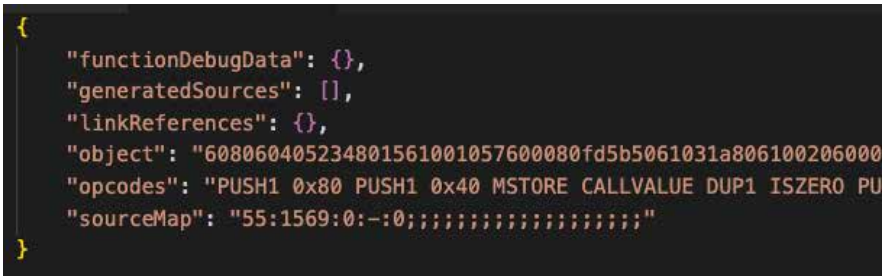


Figure 7: The content of the Bytecode

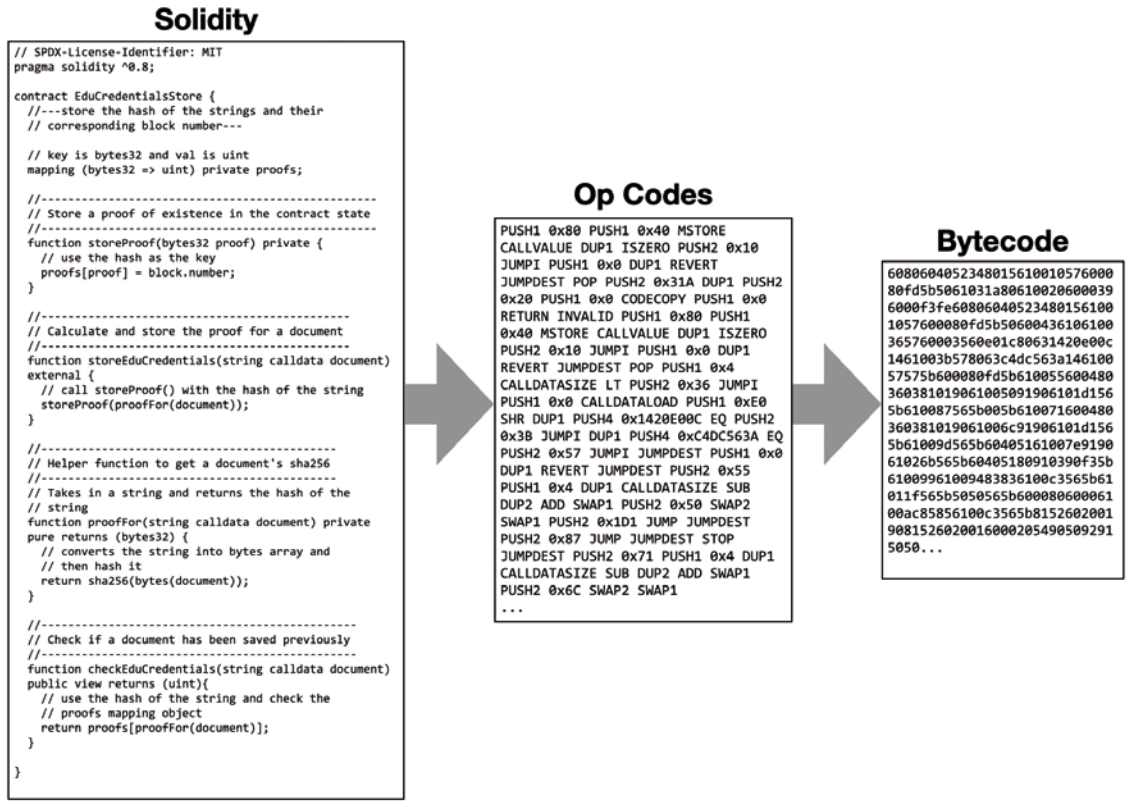


Figure 8: How Remix IDE compiles your smart contract

Deploying the Smart Contract

You're now ready to deploy the smart contract to the Goerli testnet. In Remix IDE, click on the **Deploy** icon (see **Figure 9**; step 1). Then, make sure the environment is **Injected Provider – Metamask**. By selecting Metamask, this means that the contract will deploy to whichever network the Metamask is connected to (make sure Metamask is connected to Goerli). Finally, click the **Deploy** button.

Metamask prompts you to pay for the transaction. Click **Confirm** and wait a while for the transaction to confirm.

Testing the Smart Contract

Once the smart contract is deployed, you'll see the contract under the **Deployed Contracts** section of the Remix IDE (see **Figure 10**). You'll also see the address of the contract displayed next to the contract name.

> To call a smart contract on the blockchain, your dapp needs the address of the contract as well as its ABI.

Let's now try to store the hash of the base64-encoded JSON string representing the result of a student onto the blockchain. Once the base64-encoded string is pasted into the textbox, click the **storeEduCredentials** button.

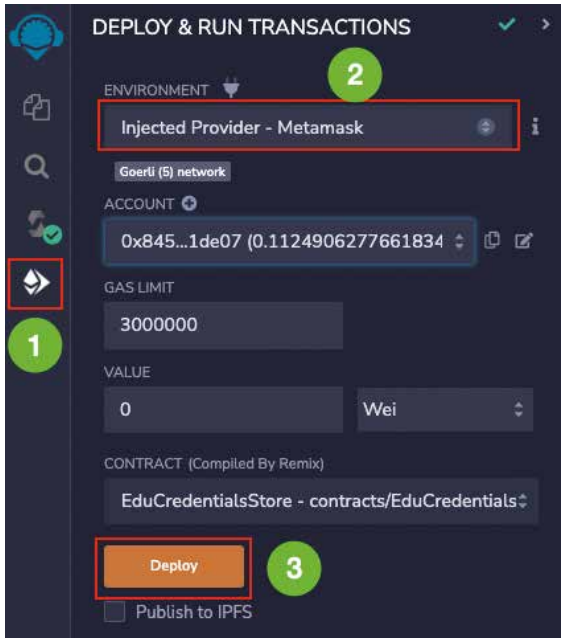


Figure 9: Deploying the contract in Remix IDE

Metamask will prompt you to pay for the transaction. Click **Confirm** (see **Figure 11**).

After a while, the transaction will be confirmed. You can now paste the same base64-encoded string into the textbox displayed next to the **checkEduCredentials** button (see **Figure 12**). Clicking the **checkEduCredentials**

button displays the block number (7881763, in this example) in which the hash of the base64-encoded string was stored on the blockchain. If you see a result of 0, this means that the hash was not found on the blockchain.

> What's the difference between the colors of the buttons? Orange buttons mean that you need to pay transaction fees (because you're modifying the state of the blockchain) and blue buttons mean that you don't have to (you're just reading data off the blockchain).

Restricting Access to Functions

Apparently, not everyone should be allowed to store the educational credentials of students on the blockchain. Ideally, only an educational institution (usually the one that deploys the smart contract) should be allowed to do that. You can make some changes to the smart contract ensuring that only the contract owner (the one that deploys the contract) is allowed to call the **storeEduCredentials()** function.

First, add in the following statements in green to the contract:

```
contract EduCredentialsStore {
  // store the owner of the contract
  address owner = msg.sender;
```

The **owner** variable automatically stores the address of the account (**msg.sender**) that deploys it. Then, in the **storeEduCredentials()** function, add in the **require()** function, as follows:

```
function storeEduCredentials(string
calldata document) external {

  require(msg.sender == owner,
    "Only the owner of contract can store
     the credentials");
```

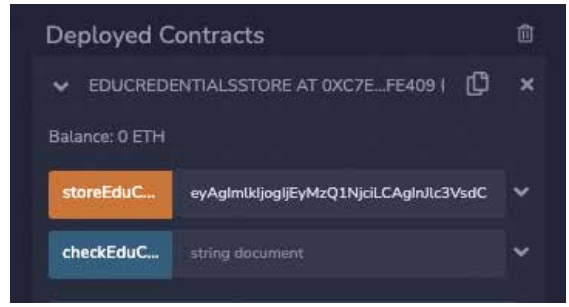


Figure 10: Viewing the deployed contract in Remix IDE

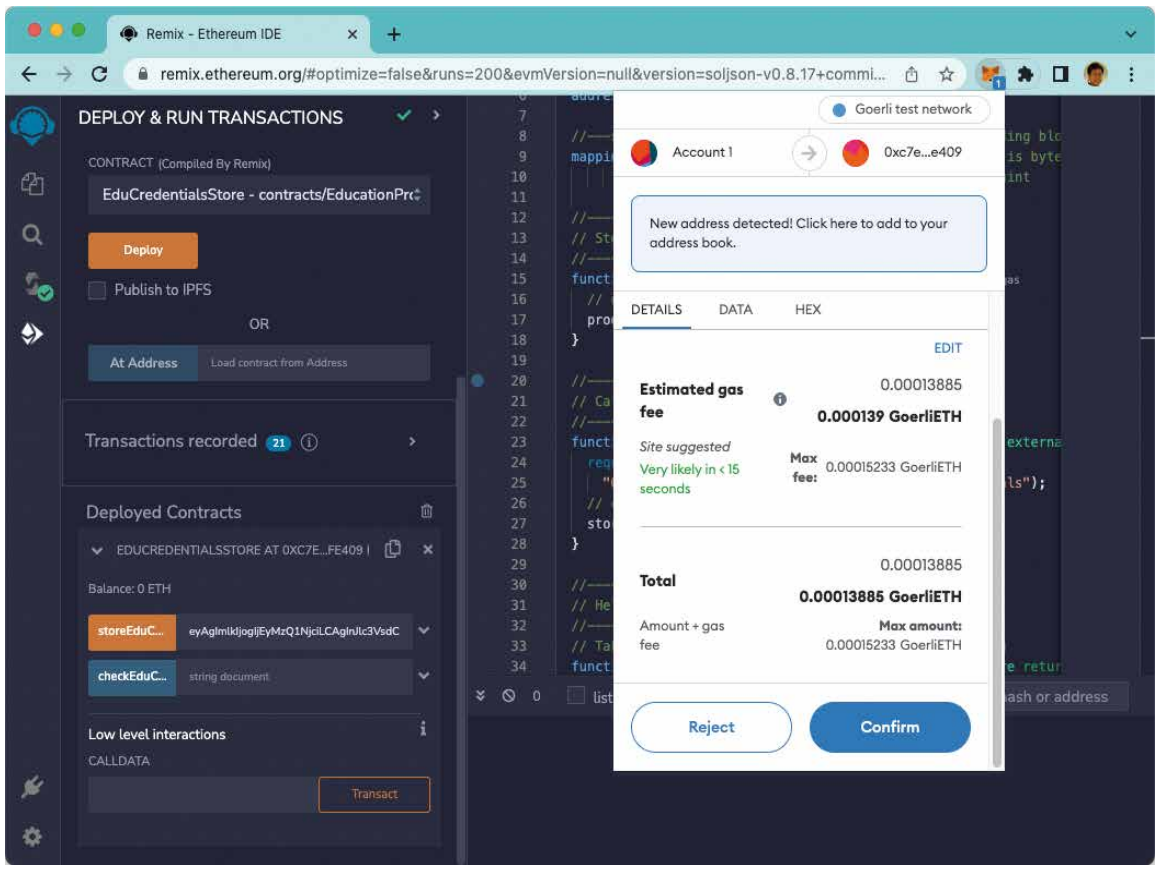


Figure 11: Paying for the transaction using Metamask

```
    // call storeProof() with the hash of
    // the string
    storeProof(proofFor(document));
}
```

The **require()** function first checks that whoever is calling this function (**msg.sender**) must be the **owner**, or else it returns an error message. If the condition is met, execution will continue, or else the execution halts.

In the Remix IDE, you can try out the above modifications by first deploying the contract using Account 1. After the contract is deployed, switch to another account in Metamask and try to call the **storeEduCredentials()** function. You'll see an error, as shown in **Figure 13**.

> Note that smart contracts are not alterable. Once they're deployed, you won't be able to make any changes to it. When you redeploy a contract, a new contract is stored on the blockchain—you won't be able to access any of the state variables of the old contract in the new one.

Accepting Payments in Smart Contracts

Now that you've managed to write a smart contract that allows educational credentials to be written to the blockchain, you might want to see how you can financially benefit from such a contract. How about earning some ethers whenever someone needs to verify the educational credential of a potential employee? Well, that's easy.

Let's add the **payable** keyword to the **checkEduCredentials()** function and delete the **view** keyword:

```
function checkEduCredentials(string
calldata document) public payable returns
(uint){

  require(msg.value == 1000 wei,
    "This call requires 1000 wei");

  // use the hash of the string and check
  // the proofs mapping object
  return proofs[proofFor(document)];
}
```

> One ether is equal to 1,000,000,000,000,000 wei (18 zeros). Wei is the smallest denomination of ether.

At the same time, add the **require()** function to indicate that the caller must send in 1000 wei (represented in **msg.value**).

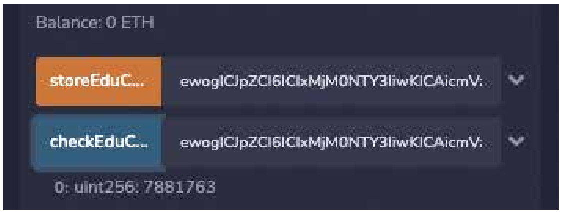


Figure 12: Verifying that the educational credential was previously stored on the blockchain

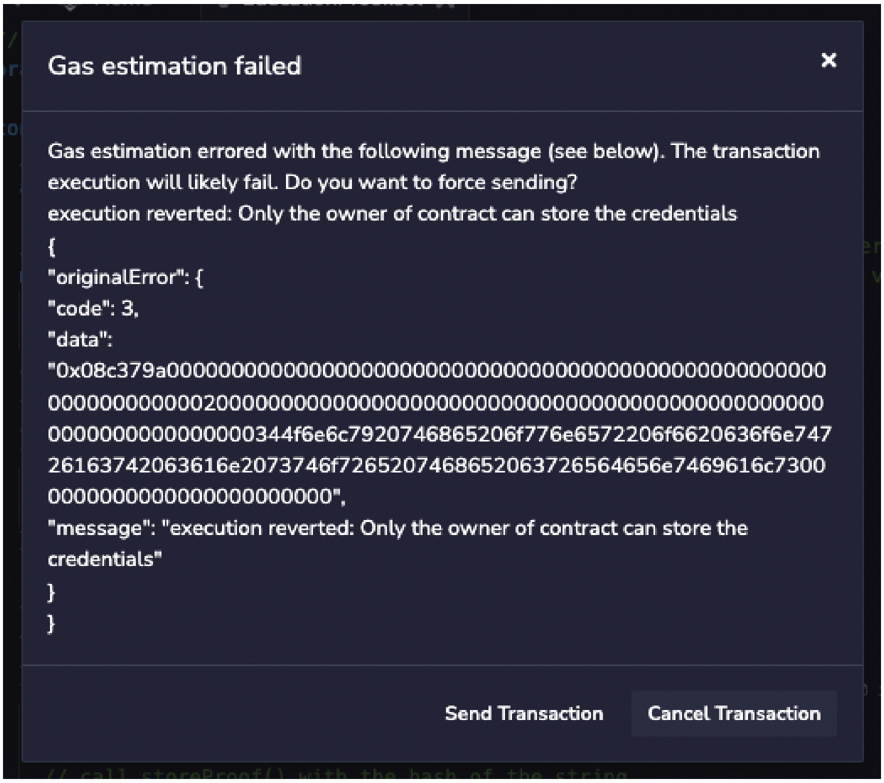


Figure 13: Remix IDE refusing the function call

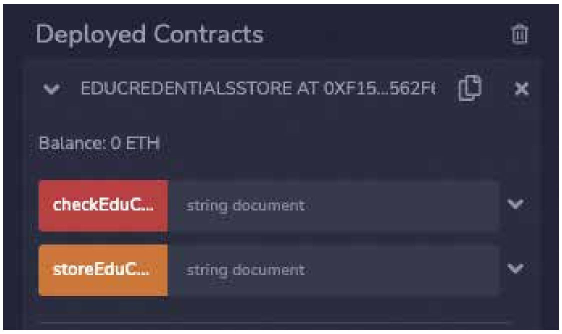


Figure 14: The checkEduCredentials button is now red

When you deploy this contract, observe that **checkEduCredentials** button is now red (see **Figure 14**).

Just like what you've done previously, go ahead and paste the base64-encoded string and then click the **storeEduCredentials** button. When you now try to click the **checkEduCredential** button, you'll see the error message shown in **Figure 15**.

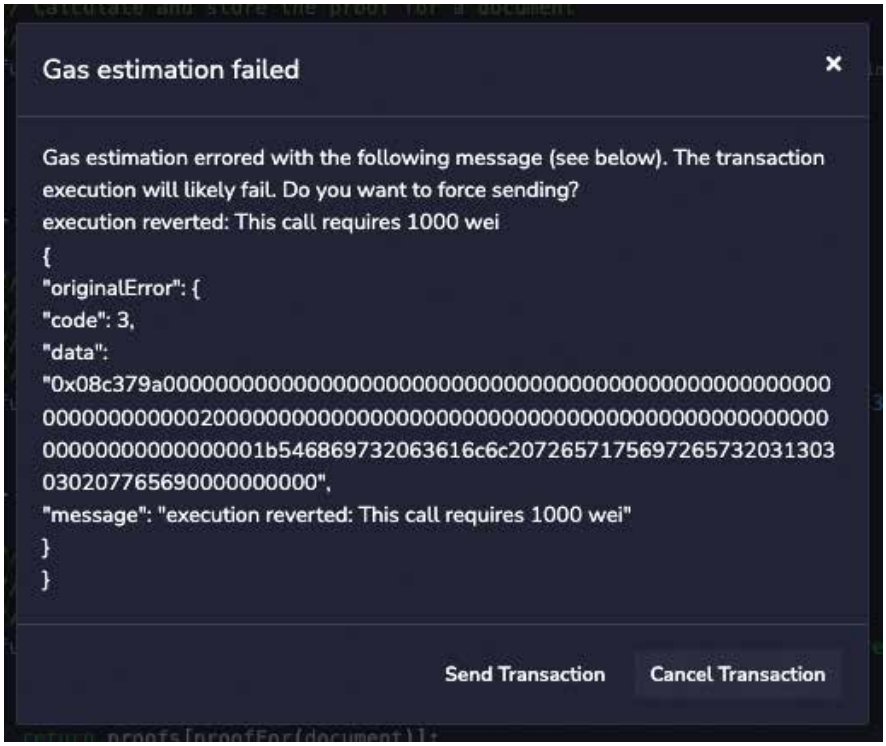


Figure 15: Remix refusing the transaction as the function expects ethers to be sent to it

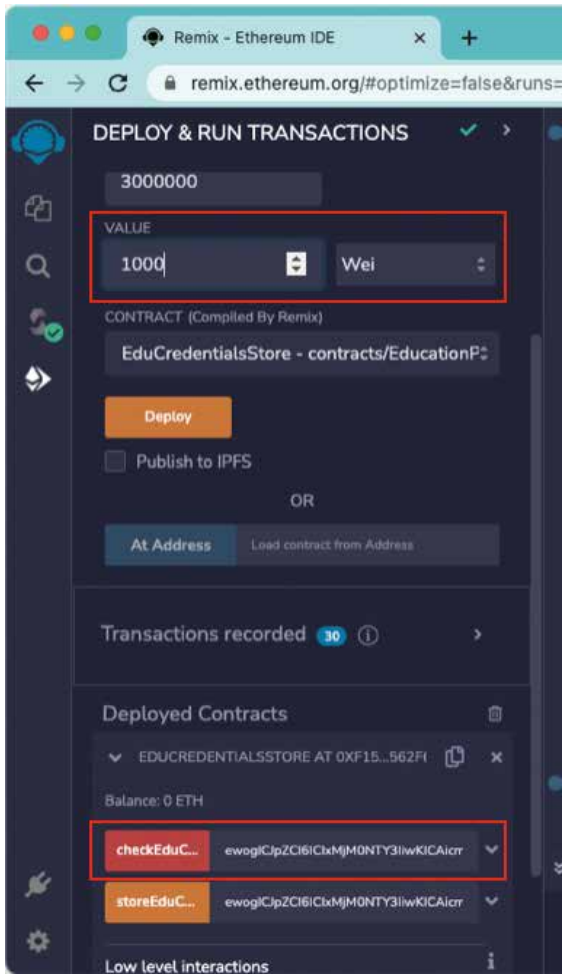


Figure 16: Sending ethers to the contract

Apparently, this is because you didn't send 1000 wei to the contract. To fix this in Remix IDE, specify 1000 wei before you click the **checkEduCredential** button, as shown in **Figure 16**.

When the transaction is confirmed, you can go to Etherscan and view the balance of the contract. For example, the address of my contract is 0xf15e70a24a50ef1b2c3bed0d3b033e35233562f6. Hence, the details of my contract on Etherscan (for the Goerli testnet) is: https://goerli.etherscan.io/address/0xf15e70a24a50ef1b2c3bed0d3b033e35233562f6.

> Etherscan is a blockchain explorer that allows you to view all the detailed transaction information that happened on the Ethereum blockchain (including the various testnets).

Figure 17 shows that the contract has a balance of 0.000000000000001 ether (which is 1000 wei). This proves that the contract has the ability to hold ethers.

Cashing Out

Now that your smart contracts holds ethers, you have a problem. The ethers are stuck forever in the contract as you didn't make any provisions to transfer them out. To be able to get ethers out of a contract, there are two main ways:

- Immediately transfer the ethers to another account the moment they're received in the **checkEduCredentials()** function.
- Add another function to transfer the ether to another account (such as the owner).

For this example, I'll use the second approach by adding a new **cashOut()** function to the contract:

```
function cashOut() public {

  require(msg.sender == owner,
    "Only the owner of contract can cash
     out!");

  payable(owner).transfer(
    address(this).balance);
}
```

In the **cashOut()** function, you first need to ensure that only the owner can call this function. Once this is verified, you'll transfer the entire balance of the contract to the owner using the **transfer()** function.

Once again, deploy the contract and then call the **checkEduCredentials()** function so that you can send 1000 wei to the contract. Then, click the **cashOut** button in the Remix IDE to transfer the ethers back to the owner. On Etherscan, you'll be able to see that there is a transfer of the balance to the owner of the contract (see **Figure 18**).

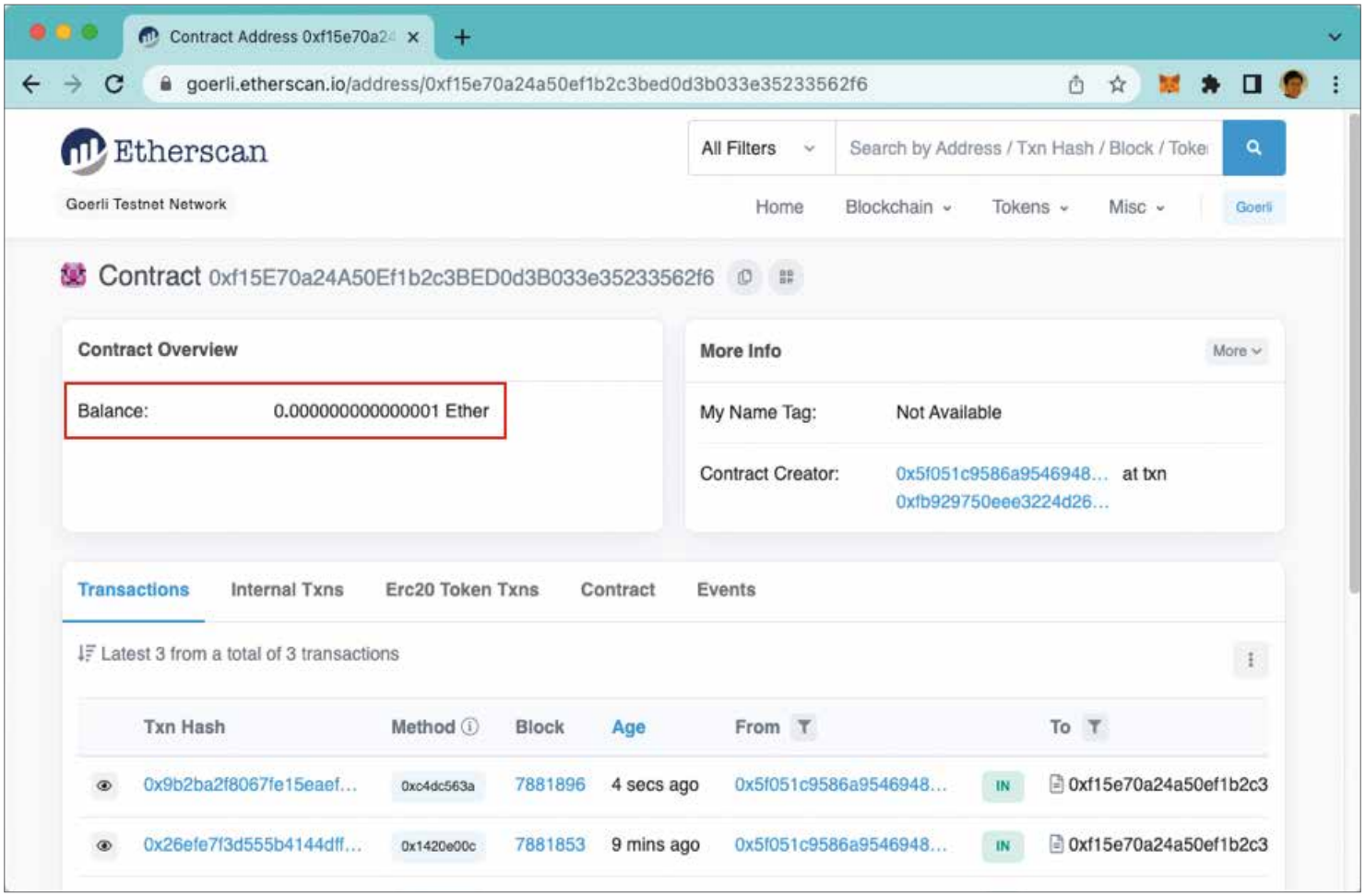


Figure 17: Smart contracts can also hold ethers

Events in Smart Contracts

A function in a smart contract can return a value back to the caller. Another way that a contract can return values back to the caller is through **events**. Events are usually used by smart contracts to keep front-end applications updated on what's happening to the smart contract.

For this example, let's define an event using the following statements in bold:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;

contract EduCredentialsStore {
  // store the owner of the contract
  address owner = msg.sender;

  //---store the hash of the strings and their
  // corresponding block number---
  // key is bytes32 and val is uint
  mapping (bytes32 => uint) private proofs;

  //---define an event---
  event Document(
    address from,
    bytes32 hash,
    uint blockNumber
  );
```

The above defined an event named **Document** with three parameters: from, hash, and blockNumber. To fire this event, use the **emit** keyword. For example, you could fire this event in the **storeProof()** function after you update the **proofs** state variable:

```
function storeProof(bytes32 proof)
private {
  // use the hash as the key
  proofs[proof] = block.number;

  // fire the event to inform the frontend
  emit Document(msg.sender, proof,
                block.number);
}
```

To try out the updated smart contract, deploy it and then call its **storeEduCredentials()** function. Remix IDE listens for the events and you can see it (see **Figure 19**).

ERC-20 Tokens

Now that you have a clear idea of how a smart contract works and its potential use case, let's move on to discuss another application of smart contracts. As you learned earlier, a smart contract allows you to store values on the blockchain, and in the case of the sample contract, the contract stores a mapping object on the blockchain. Let's

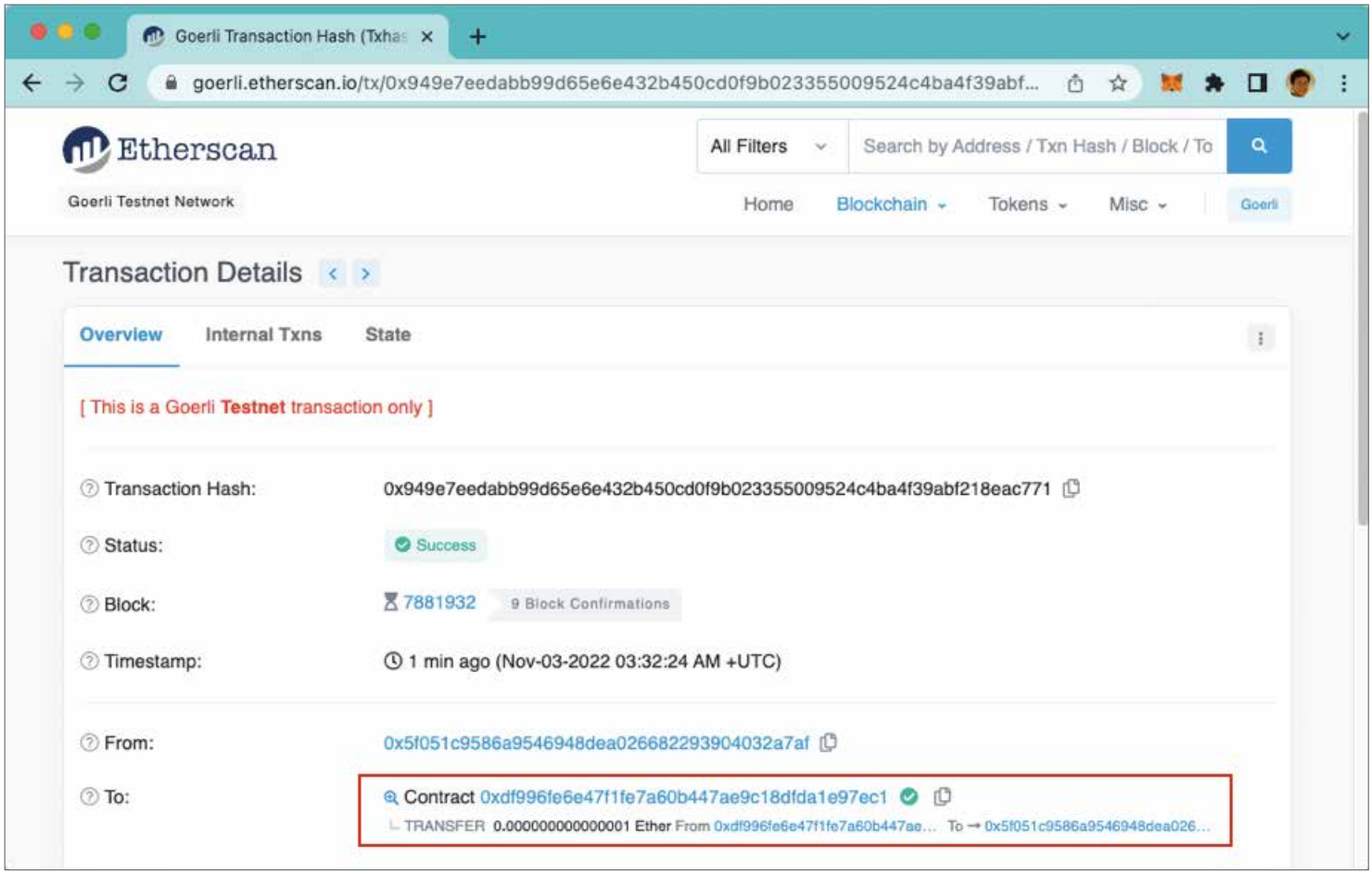


Figure 18: Etherscan records the internal transfers of ethers to another account.

take this idea one step further. Instead of storing key/value pairs of hashes and their associated block numbers, why not use the key/value pair to store account addresses and their respective balances (see **Figure 20**).

> ERC-20 (ERC stands for Ethereum Request for Comments) was proposed by Fabian Vogelsteller in November 2015. It implements an API for tokens within smart contracts so that tokens can be transferred between accounts, approved for use, balances queried, etc.

This is the idea behind **tokens**. A token is a representation of *something* (of value) on the blockchain. This *something* may represent assets, shares in a company, memberships in a club, or virtually anything you want to represent. In **Figure 20**, the number associated with each account address represents the quantity of tokens held by each account. As an example, say you're selling some items on the web. The only way customers could pay for your goods is through tokens, not cash. So in order to buy your items, your customers need to acquire tokens from you before they can buy from you.

How do you maintain the balance of each account token balance? You can do so through **token contracts**. A token contract writes the balance of tokens held by each account holder on the blockchain. To ensure that tokens can be easily exchanged with others on the Ethereum blockchain, create your tokens using the ERC-20 standard.

> What's the difference between a coin and a token? A coin is the native cryptocurrency on a blockchain, such as ether on Ethereum. A token is an asset that's created on top of an existing blockchain that already has a native cryptocurrency. A good example of a token is USDC (U.S. Dollar Coin) that exists on the Ethereum blockchain.

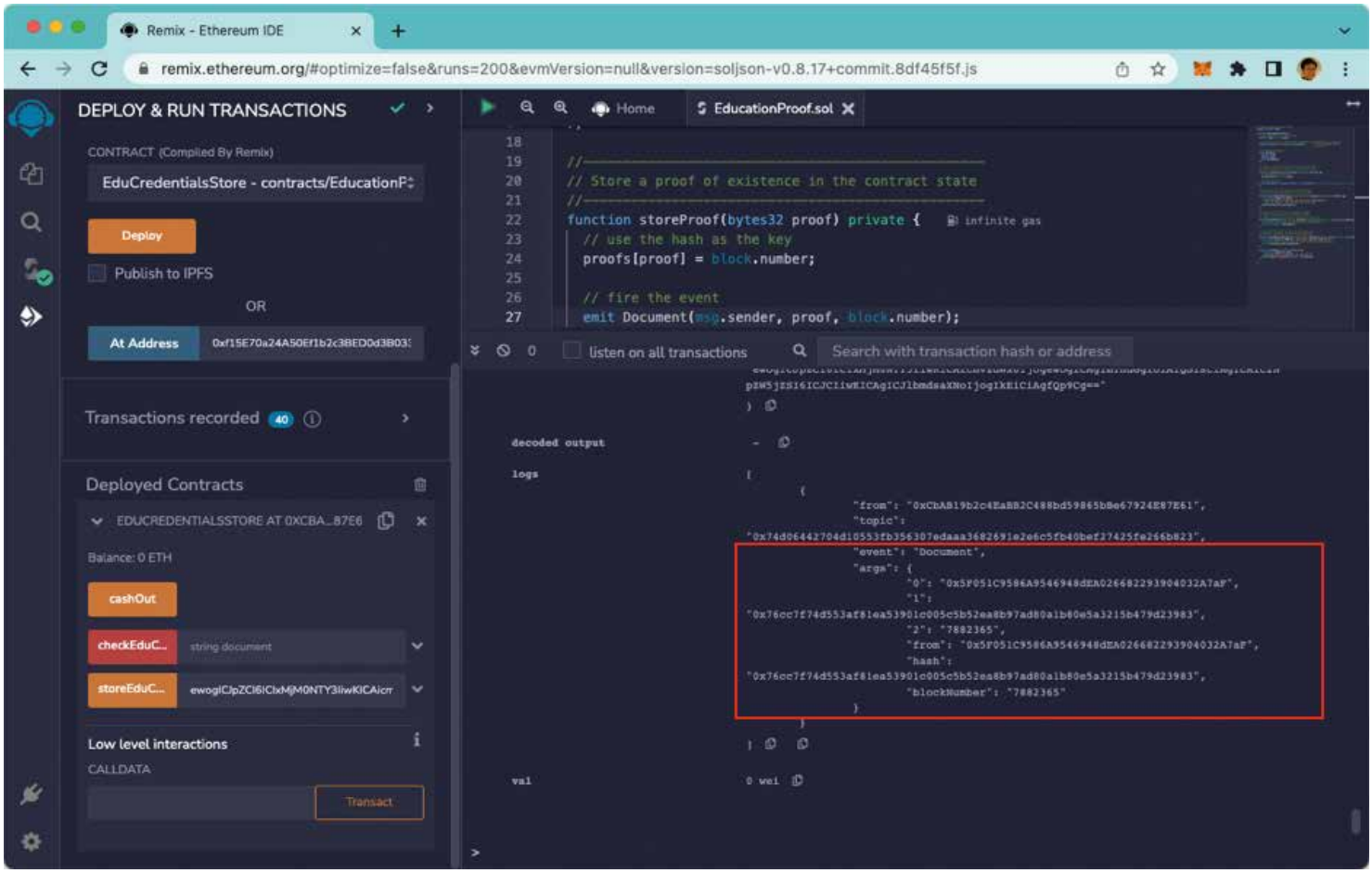


Figure 19: You can use Remix IDE to examine the events fired by smart contracts.

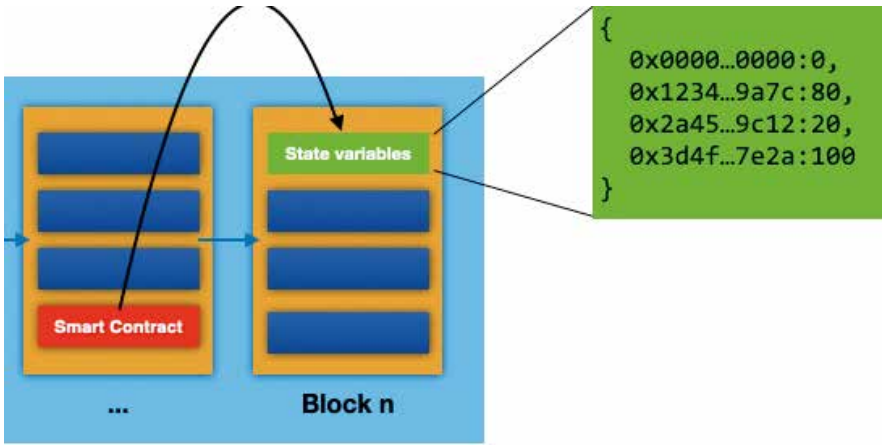


Figure 20: Storing account addresses and balance on the blockchain

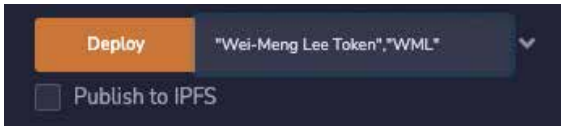


Figure 21: Deploying the token contract with the name and symbol

Listing 2: The ERC-20 Interface

```
contract ERC20Interface {
    function totalSupply() public constant
        returns (uint);
    function balanceOf(address tokenOwner) public
        constant returns (uint balance);
    function allowance(address tokenOwner, address
        spender) public constant returns
        (uint remaining);
    function transfer(address to, uint tokens) public
        returns (bool success);
    function approve(address spender, uint tokens)
        public returns (bool success);
    function transferFrom(address from, address to,
        uint tokens) public returns (bool success);

    event Transfer(address indexed from,
        address indexed to, uint tokens);
    event Approval(address indexed tokenOwner,
        address indexed spender, uint tokens);
}
```

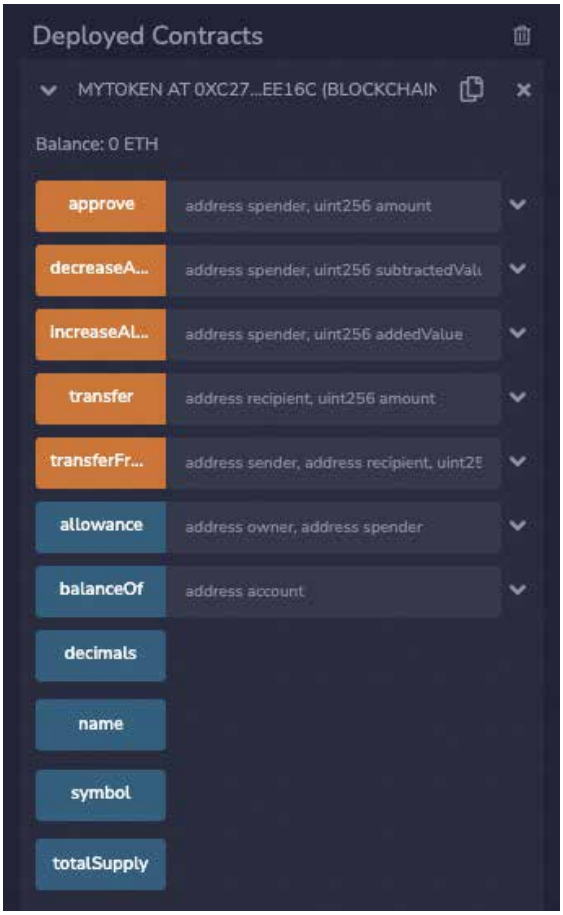


Figure 22: The various functions defined in an ERC-20 token contract

Note that ERC-20 tokens are also known as **fungible** tokens, which means that a, ERC-20 token is divisible (can be broken down into small units), interchangeable, and non-unique.

ERC-20 Token Contracts

Now that you know what a token is, it's time to create one to see specifically how it works. **Listing 2** shows the ERC-20 interface, which means that if you want to create an ERC-20 token, your contract must implement the set of functions and events as defined in the ERC-20 interface.

Specifically, your ERC-20 token contract must implement the following functions:

- **totalSupply()**: Returns the total token supply
- **balanceOf(address _owner)**: Returns the account balance of **_owner**
- **transfer(address _to, uint256 _value)**: Transfers **_value** to **_to** and fires the **Transfer** event. The function should revert if the **_from** account doesn't have enough tokens to spend
- **approve(address _spender, uint256 _value)**: Allows **_spender** to withdraw from the account several times, up to the **_value** amount

Figure 23: The deployed token contract (copy its address to the clipboard)

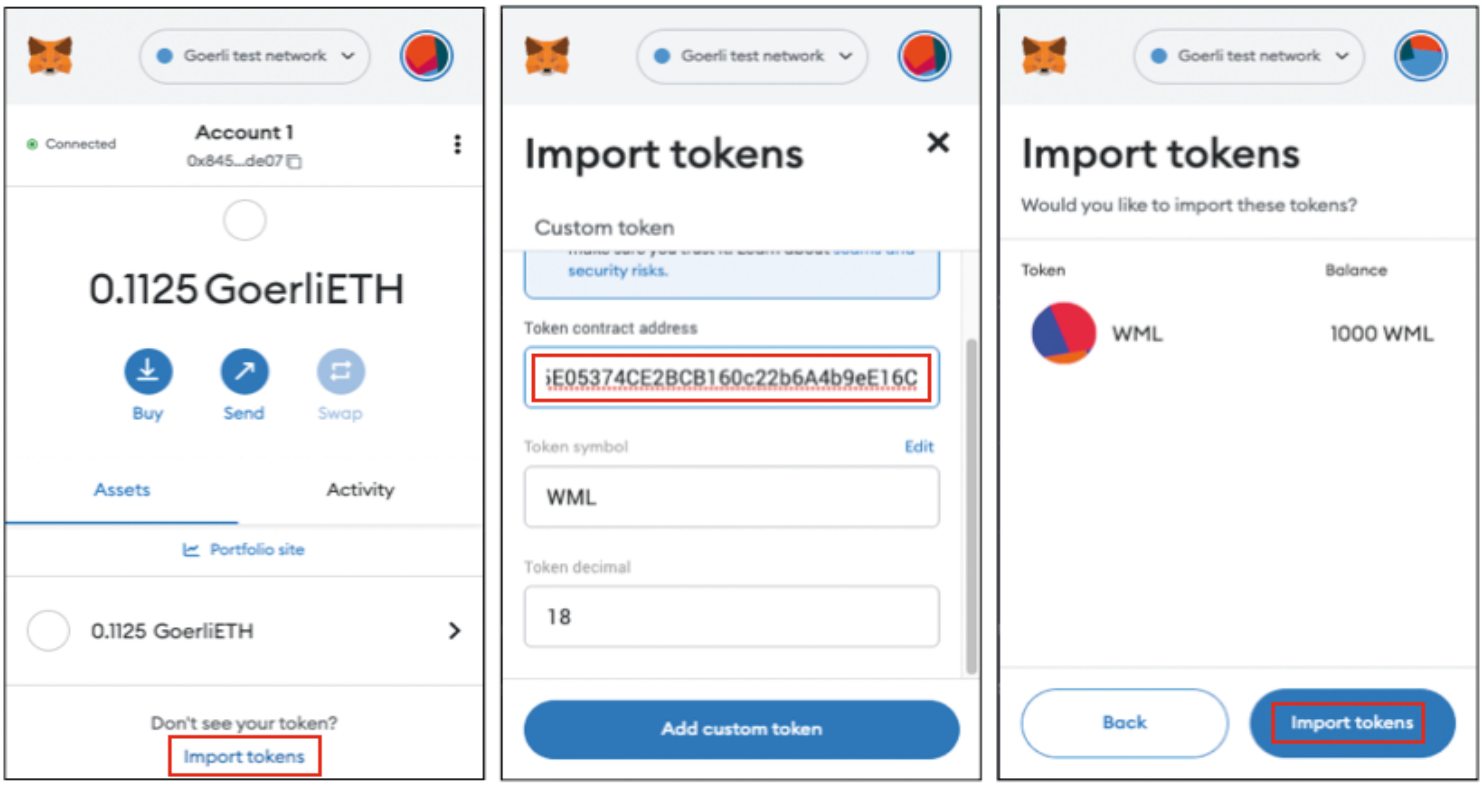


Figure 24: Adding the token to Metamask

- **transferFrom(address _from, address _to, uint256 _value)**: Transfers **_value** from **_from** to **_to** and fires the **Transfer** event. The function should revert unless the **_from** account has deliberately authorized the sender of the message via some mechanism.
- **allowance(address _owner, address _spender)**: Returns the amount that the **_spender** is still allowed to withdraw from the **_owner.**

In addition, the token contract must also emit the following events:

- **Transfer(address indexed _from, address indexed _to, uint256 _value)**: Must trigger when tokens are transferred, including zero-value transfers
- **Approval(address indexed _owner, address indexed _spender, uint256 _value)**: Must trigger on any successful call to **approve(address _spender, uint256 _value)**

Rather than implement all these functions and events yourself, **OpenZeppelin** has provided a base implementation for ERC-20. You can find this implementation at: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.0.0/contracts/token/ERC20/ERC20.sol/

If you're writing an ERC-20 token contract, you just need to import the base implementation from OpenZeppelin and inherit from it.

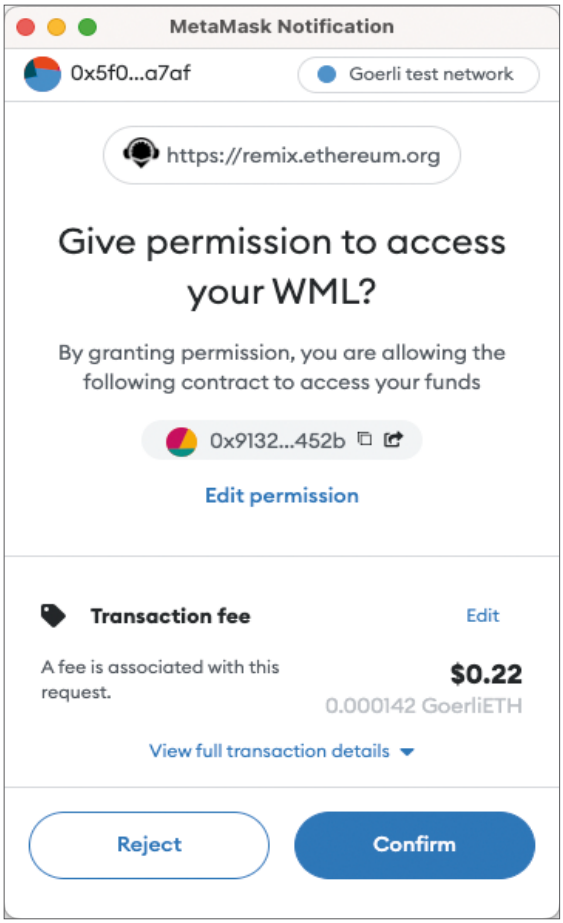


Figure 25: Redeploy the modified smart contract to accept tokens for payment.

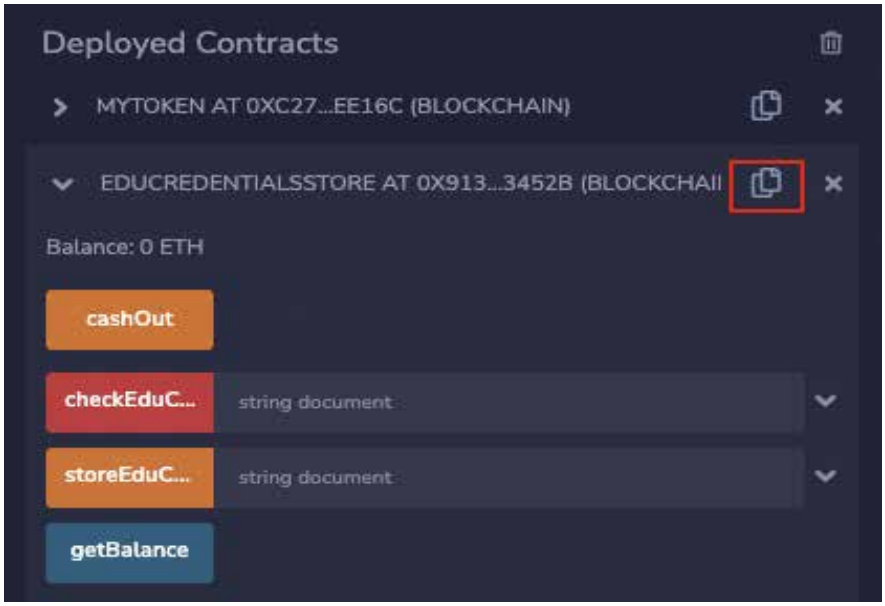


Figure 26: Approve the use of the tokens on a smart contract.

For this example, let's create a new contract in Remix IDE and name it **token.sol**. Populate it with the statements shown below:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;
import "https://github.com/OpenZeppelin/
openzeppelin-contracts/blob/v4.0.0/
contracts/token/ERC20/ERC20.sol";
```

> OpenZeppelin is an open-source framework to build secure smart contracts. OpenZeppelin provides a complete suite of security products and audit services to build, manage, and inspect all aspects of software development and operations for decentralized applications.

```
contract MyToken is ERC20 {
  constructor(string memory name,
  string memory symbol)
    ERC20(name, symbol) {

      // ERC20 tokens have 18 decimals
      // number of tokens minted = n * 10^18
      uint256 n = 1000;
      _mint(msg.sender,
        n * 10**uint(decimals()));
    }
}
```

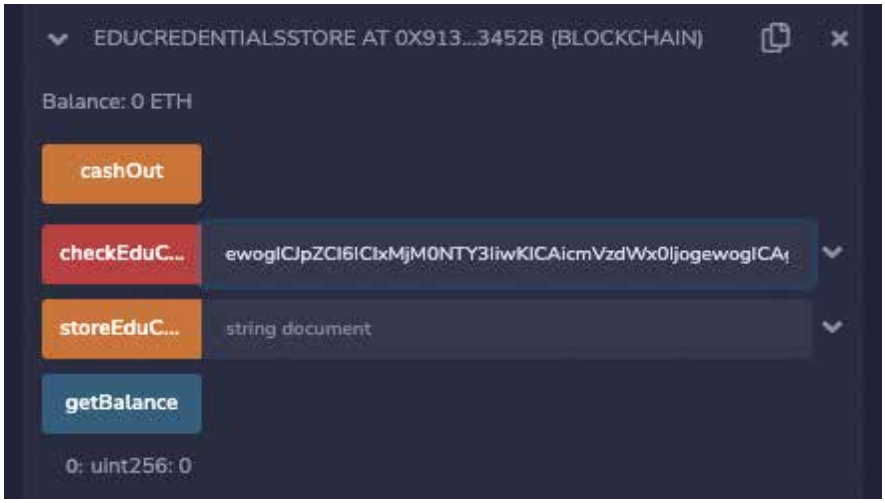


Figure 28: Call the checkEduCredentials() function.

> All operations involving tokens are based on the base units. For example, if I want to send one token to another account, I have to transfer 1,000,000,000,000,000,000 base units of my tokens to that account.

In the contract above, I'm creating 1000 tokens (value of **n**), and each token can go up to 18 decimal places of precision (because the **decimals()** function returns 18). Internally within the contract, the number of tokens minted is dependent on **n** and the precision. In this example, although the total supply is 1000 tokens, the total base units of tokens minted is equal to: **1000 x 10^{18}** or **1000,000,000,000,000,000,000**.

Deploying the Token Contract

To deploy the token contract, you must specify two arguments:

- The name (description) of the token
- The symbol of the token

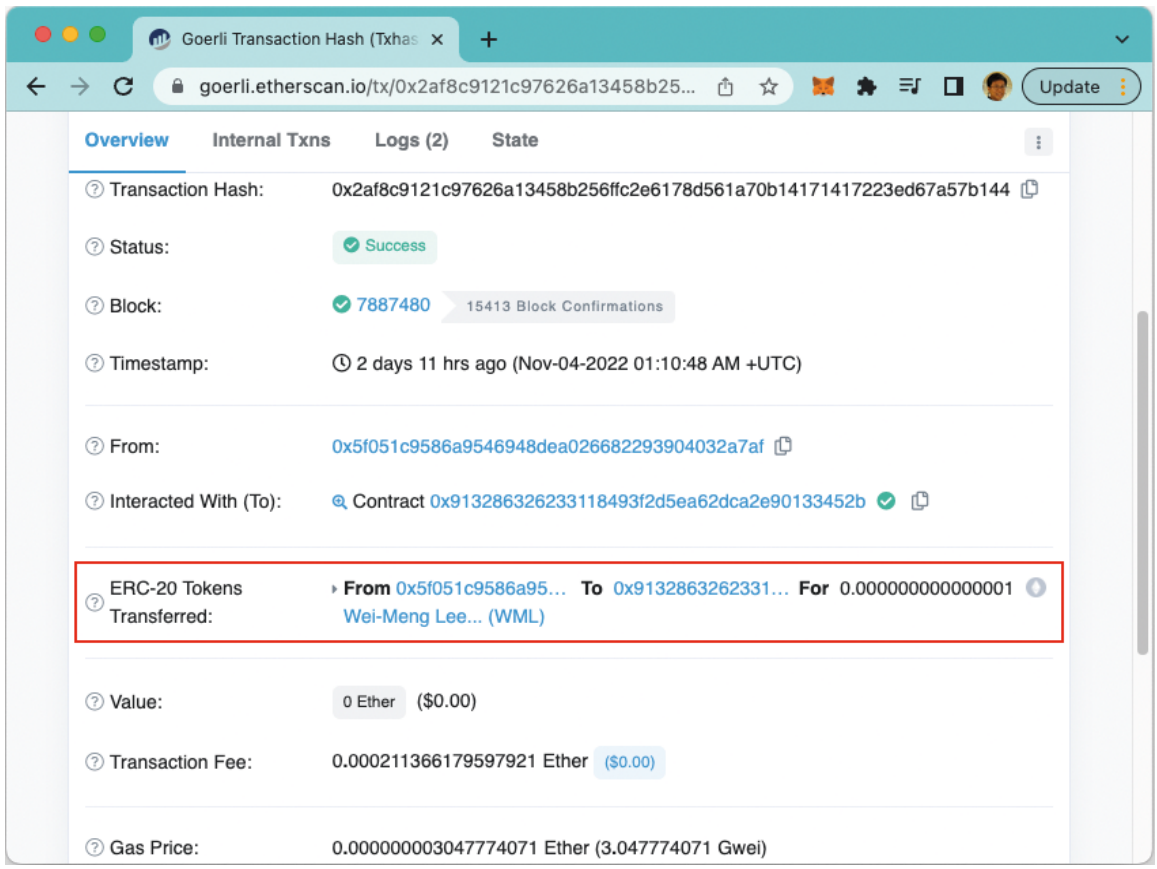


Figure 29: Etherscan records the transfer of token from an account to the smart contract.

Listing 3: The modified smart contract now accepts payment using tokens

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;

import "./token.sol";

contract EduCredentialsStore {
  // store the owner of the contract
  address owner = msg.sender;

  MyToken token = MyToken(address(
    0xc276658A795E05374CE2BCB160c22b6A4b9eE16C));

  //---store the hash of the strings and their
  // corresponding block number---
  // key is bytes32 and val is uint
  mapping (bytes32 => uint) private proofs;

  //---define an event---
  event Document(
    address from,
    bytes32 hash,
    uint blockNumber
  );

  //=======================================
  // return the token balance in the contract
  //=======================================
  function getBalance() public view returns (uint256) {
    return token.balanceOf(address(this));
  }

  //-----------------------------------------------
  // Store a proof of existence in the contract state
  //-----------------------------------------------
  function storeProof(bytes32 proof) private {
    // use the hash as the key
    proofs[proof] = block.number;

    // fire the event
    emit Document(msg.sender, proof, block.number);
  }

  //-------------------------------------------
  // Calculate and store the proof for a document
  //-------------------------------------------
  function storeEduCredentials(string calldata
  document) external {
    require(msg.sender == owner,
      "Only the owner of contract can store the
       credentials");
    // call storeProof() with the hash of the string
    storeProof(proofFor(document));
  }

  //-------------------------------------------
  // Helper function to get a document's sha256
  //-------------------------------------------
  // Takes in a string and returns the hash of the
  // string
  function proofFor(string calldata document) private
  pure returns (bytes32) {
    // converts the string into bytes array and then
    // hash it
    return sha256(bytes(document));
  }

  //-----------------------------------------------
  // Check if a document has been saved previously
  //-----------------------------------------------
  function checkEduCredentials(string calldata
  document) public payable returns (uint){

    // require(msg.value == 1000 wei,
    //   "This call requires 1000 wei");

    // msg.sender is the account that calls the
    // token contract
    // go and check the allowance set by the caller
    uint256 approvedAmt =
      token.allowance(msg.sender, address(this));

    // the amount is based on the base unit in the
    // token
    uint requiredAmt = 1000;

    // ensure the caller has enough tokens approved
    // to pay to the contract
    require(approvedAmt >= requiredAmt,
      "Token allowance approved is less than what you
       need to pay");

    // transfer the tokens from sender to token contract
    token.transferFrom(msg.sender,
        payable(address(this)), requiredAmt);

    // use the hash of the string and check the proofs
    // mapping object
    return proofs[proofFor(document)];
  }

  function cashOut() public {
    require(msg.sender == owner,
      "Only the owner of contract can cash out!");
    payable(owner).transfer(address(this).balance);
  }

}
```

Figure 21 shows an example name and symbol for the token. Click **Deploy** to deploy the token contract.

Once the contract is deployed, you can expand on the contract name to review the various functions (see **Figure 22**). These are the functions you need to implement in your ERC-20 token contract (which was implemented by the OpenZeppelin base contract).

Copy the address of the deployed token contract (see **Figure 23**).

Adding the Token to Metamask

In Metamask, click the **Assets** tab and click the **Import tokens** link at the bottom of the screen (see **Figure 24**). In the next screen, paste the token contract that you have copied from Remix IDE. The token symbol is automatically displayed. Click the **Add custom token** button. You can see that your account has 1000 WML tokens. Make sure that the account you're adding the token to is the same account that deployed the token contract.

What Can You Do with the Token?

Now that you've created the token, what can you do with it? It's really up to you to create a utility for the token. You could promote your token as a form of investment, or a representation of an asset that you're selling (such as properties or securities).

To sell a token, you can ask the recipient to pay you in fiat currency and then transfer the token to him/her. Alternatively, you can also code your token contract to receive ethers and then programmatically transfer the token to the sender of the ether (this is beyond the scope of this article).

Using Tokens for Smart Contract Payments

In an earlier section of this article, you wrote a contract that accepts ether for a service (the **checkEduCreden-**

Figure 30: Finding the balance of token in a smart contract

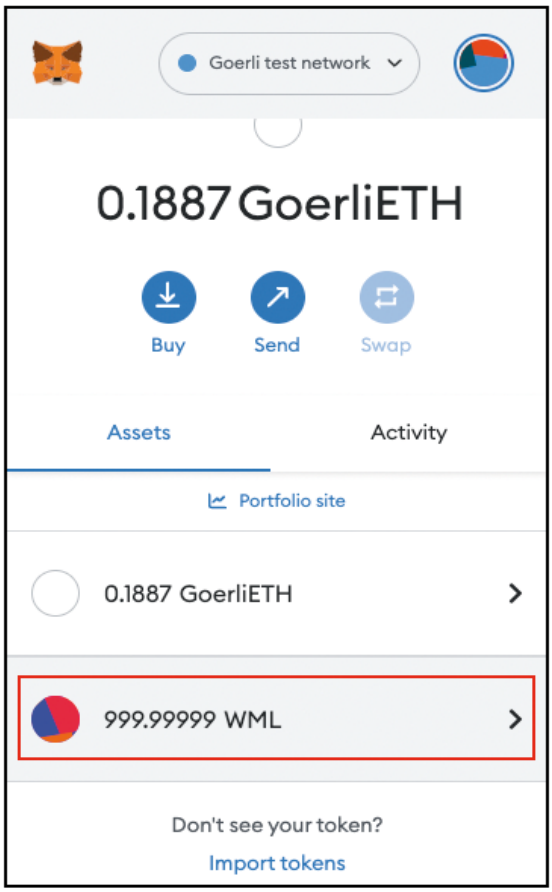


Figure 31: The token balance of the account is also reduced

tials() function). What if instead of accepting ether, you accept tokens for payment? This can be done quite easily.

In **Listing 3**, you made the following changes to the **EduCredentialsStore** contract:

- Imported the **token.sol** token contract. This is the token contract that you deployed earlier.
- Created an instance of the **MyToken** contract. You need to specify the address of the deployed token contract.
- Added a function called **getBalance()**. This allows you to know the token balance held by the smart contract.
- Modified the **checkEduCredentials()** function so that now, instead of accepting ethers for payment, you only accept tokens. To do so, you first check how many tokens the caller of the contract has approved for use in this contract. Then you check to make sure that the amount approved to be used is at least 1000 base units (this is the amount that you decide to charge for calling this service). If the amount approved is sufficient, you transfer the tokens to this contract.

Re-deploy this **EduCredentialsStore** contract and take note of its address (see **Figure 25**). For my example, the deployed contract address is **0x913286326233118493F2D5eA62dCA2E90133452B**.

Before you can call the **checkEduCredentials()** function (which now requires a payment of 1000 base units of the tokens instead of 1000 wei), the token owner (you) needs to approve 1000 base units of the token to be paid to the smart contract. To do that, you need to call the **approve()** function of the token contract with the following values (see also **Figure 26**):

```
0x913286326233118493F2D5eA62dCA2E90133452B,
1000
```

The above value indicates that you want to approve 1000 base units of the token to be spent on the smart contract whose address is specified (0x913286326233118493F2D5eA62dCA2E90133452B).

Click on the **approve** button and Metamask shows the prompt, as shown in **Figure 27**. Click **Confirm** to grant permission for your tokens to be used on the smart contract.

You will now be able to call the **checkEduCredentials()** function (see **Figure 28**).

Once the transaction is confirmed, 1000 base units of the token will be transferred to the smart contract. If you examine the transaction on Etherscan, you can observe that there's a transfer of ERC-20 tokens (see **Figure 29**).

To verify that the contract did indeed receive the tokens, click the **getBalance** button (see **Figure 30**). You should see a value of 1000.

In Metamask, you'll also see that the account holding on to the token has its balance reduced (see **Figure 31**). This is because 1000 base units of the tokens have been transferred to the smart contract as payment.

Summary

In this article, I've walked you through what a smart contract is, how it works, and some of the use cases for it. I also demonstrated one good use case of smart contracts—creating token contracts. Token contracts are what makes the blockchain world so exciting. A lot of tokens have since been created and this is the key technology that created stablecoins like USDC, USDT, and more. I'm excited to see what types of tokens you'll create after reading this article!

Wei-Meng Lee

An Introduction to Distributed Tracing with OpenTelemetry in .NET 7

Although enterprises are gathering more data than ever, they may not be using it to their advantage. In most cases, companies don't even know what kind of information they possess or how to use it. This is where OpenTelemetry can help. OpenTelemetry facilitates the collection and analysis of data from multiple sources simultaneously, enabling businesses to make more informed decisions about their operations. With OpenTelemetry, enterprises have transformed their approach to observability. OpenTelemetry is adept at troubleshooting, alerting, and debugging applications and is well poised to be the future of instrumentation.

OpenTelemetry includes a set of tools and libraries for distributed tracing and monitoring. It was created by the Cloud Native Computing Foundation (CNCF) to provide a standard way of instrumenting code for tracing. OpenTelemetry allows you to collect data about the flow of requests through your system and it can be used with any programming language. It has particularly good support for .NET.

The data collected can be used to generate a trace of how the requests were handled. This is useful for diagnosing performance issues or errors. In addition to .NET, there are libraries available for ASP.NET Core and other frameworks. These libraries make it easy to instrument your code and collect trace data. Overall, OpenTelemetry is a great tool for distributed tracing and monitoring. It's easy to use with .NET and it has good support for various frameworks.

This article talks about the concepts related to OpenTelemetry, why it's useful, and how you can work with it in ASP.NET 7 Core applications.

If you're to work with the code examples discussed in this article, you need the following installed in your system:

- Visual Studio 2022 Preview
- .NET 7.0
- ASP.NET 7.0 Runtime

If you don't already have Visual Studio 2022 Preview installed in your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

In this article, you'll:

- Learn OpenTelemetry and its benefits
- Build a simple application in ASP.NET 7 Core
- Configure the application to provide support for OpenTelemetry
- Extend OpenTelemetry
  - Build a custom processor
  - Build a custom exporter
- Export telemetry data
- Understand the future of OpenTelemetry

What Is Distributed Tracing?

Distributed tracing is a method used to capture events across multiple services in order to troubleshoot issues and optimize performance. It's used to understand the execution of a distributed system and involves tracking the execution of each component in the system and recording information about each step. Using this information, the path taken by the system during execution can be recreated.

Distributed tracing can be used to identify bottlenecks in a system, or to understand the behavior of a complex distributed system. It can also be used to diagnose errors in a distributed system. This can help you understand how your application is performing and find root causes for any performance issues.

What Is Observability? Why Is It Important?

Observability refers to a system's ability to measure its current state from the data it generates, such as log files, metrics, and traces. Observability can improve operational visibility by capturing data and proactively observing distributed systems. It can identify relevant data based on how an application performs in a production environment over time.

Metrics, logging, and tracing are the three foundation elements of observability. I'll discuss each of these shortly.

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.

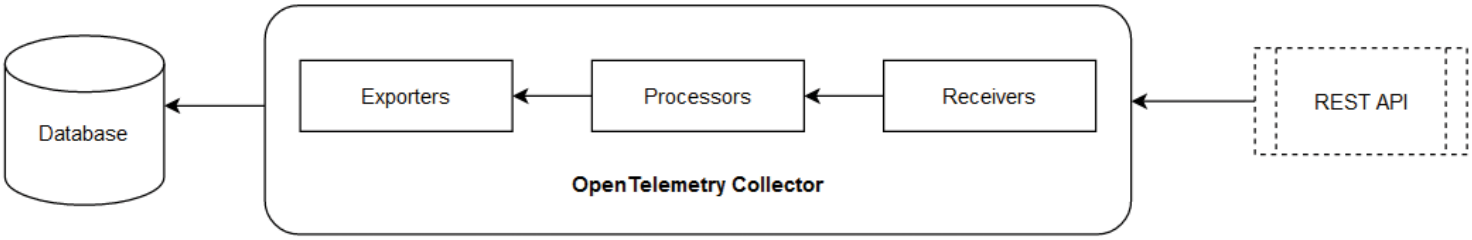


Figure 1: An OpenTelemetry Collector at work

What Is Open Telemetry? Why Should I Use It?

OpenTelemetry is an open-source distributed tracing framework for .NET 7. It uses the open-source framework for collecting, storing, and analyzing telemetry data (metrics, logs, and traces) and provides a high-level API for distributed tracing and metrics that can be used by applications in different environments, including monoliths, microservices, and serverless applications.

OpenTelemetry is a .NET library for distributed tracing and metrics that was originally introduced in 2016 by Microsoft as part of the OpenTracing standard adopted by the OpenTracing Working Group at CNCF.

Since then, it's been widely adopted by various projects and companies across multiple industries, including Microsoft's own Azure services and Azure Functions. OpenTelemetry is a community-driven project, released under the Apache 2.0 license. It's available on GitHub and NuGet, and supports .NET Core 1.1 or higher, ASP.NET 5+, or any other .NET framework that implements the OpenTracing API.

OpenTelemetry comprises a collection of APIs, SDKs, tooling, and integrations for collecting and storing telemetry data that includes traces, metrics, and logs. In OpenTelemetry, a collector is a component that receives, processes, and exports telemetry data, as shown in **Figure 1**.

OpenTelemetry can be used to correlate events that occur when your application is in execution. You can correlate events based on:

- **Execution context:** You can correlate logs and traces based on the traceId of an event.
- **Execution time:** You can correlate logs and traces based on the time of the event, i.e., when the event occurred.
- **Telemetry Origin:** You can correlate events based on the service name, the service instance, or the version of the service.

Why Open Telemetry?

OpenTelemetry is a new distributed tracing system designed to be used in a wide variety of programming languages and platforms. There are many benefits to using OpenTelemetry for distributed tracing, including:

- It is open source and well-supported by the community.
- It has good integration with other popular open-source tracing systems like Jaeger.
- It offers many features that make it simple to instrument your code and collect trace data.
- It has strong support for .NET and other Microsoft technologies.

If you're looking for a distributed tracing system to use in your .NET applications, OpenTelemetry is a great option to consider.

Components of OpenTelemetry

The OpenTelemetry collector has three components: receivers, processers, and exporters.

Receivers

An OpenTelemetry receiver collects data and sends it to a collector. A receiver can support one or more data sources and may be push- or pull-based. In the OpenTelemetry pipeline, receivers receive data in a specified format, convert it to an internal format, and pass it on to processors and exporters.

Processors

A processor is an optional component in the collector pipeline that processes data before sending it to an exporter. Using processors, you can batch-process, sample, transform, and enrich the telemetry data you receive from the collector before exporting it.

Exporters

OpenTelemetry can export telemetry data, such as metrics and traces, to several back-ends. An exporter can be push- or pull-based and is responsible for exporting data to one or more back-ends or destinations, such as Azure Monitor, Jaeger, Splunk, Prometheus, etc.

What Are Traces, Metrics, and Logs?

There are three main types of data that are important for understanding the performance of a system: tracing data, metrics, and logs.

Traces

Tracing is a process that records the details of all events. It captures data about the flow of a particular request or transaction across multiple components in your system, including information such as the event, event type, method calls, and exceptions raised. Tracing data is very useful for understanding how a system works and for finding bottlenecks.

Metrics

Metrics are numerical data that provide insights on the performance of an application, such as the number of requests per second, the average response time, and so on. For example, you could collect the average number of milliseconds it took to render a web page over time.

Metrics can be used to evaluate performance, capacity planning, or other aspects of an application's health. Metrics are very useful for understanding the overall performance of a system. The data captured includes specific events within certain boundaries of time, such as average response times or throughput rates.

Logs

Logs record events that occur when an application is in execution or information about how long a snippet of code or a method took to complete. This includes information about errors, warnings, and more. Logs are typically used for debugging and troubleshooting problems in an application.

These historical records describe events in an application during some period (i.e., user log-in and authentication). You can use logs to debug problems because they allow you to see what went wrong when an issue occurs, as shown in **Figure 2**.

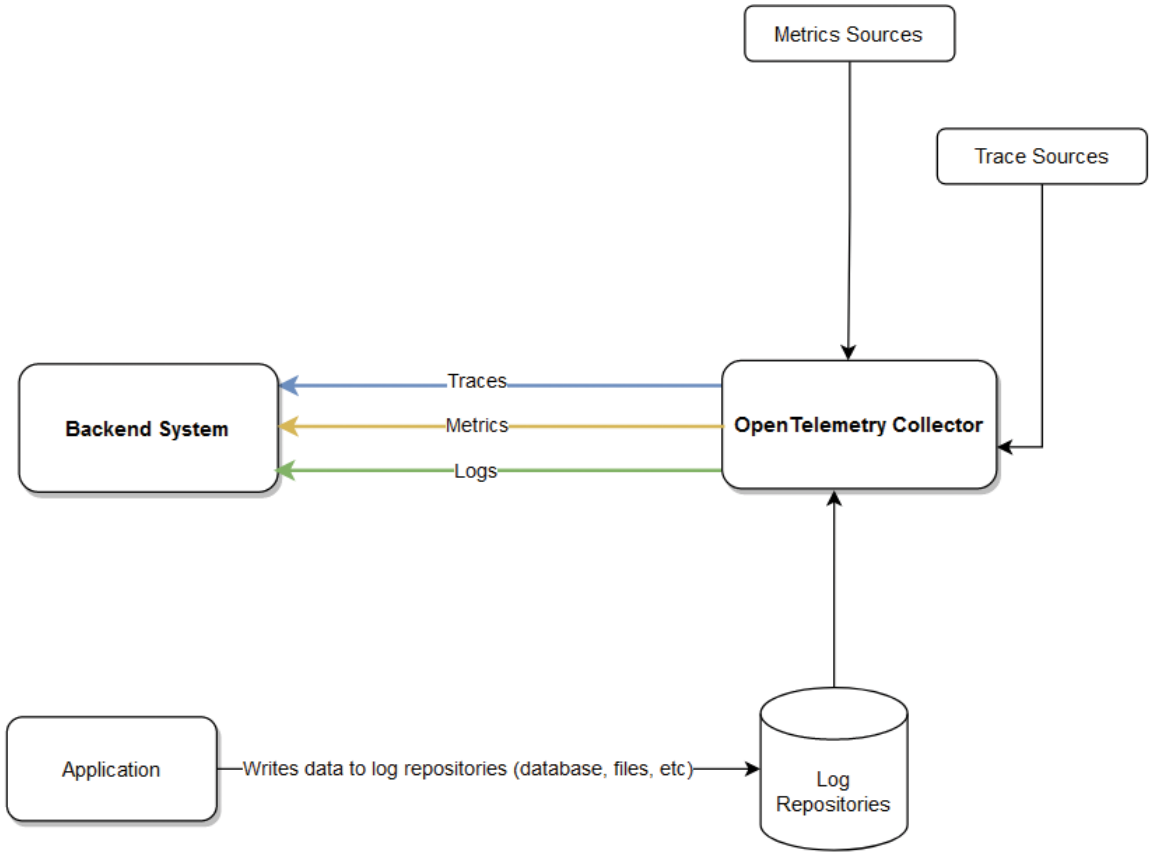


Figure 2: The OpenTelemetry ollcector at work

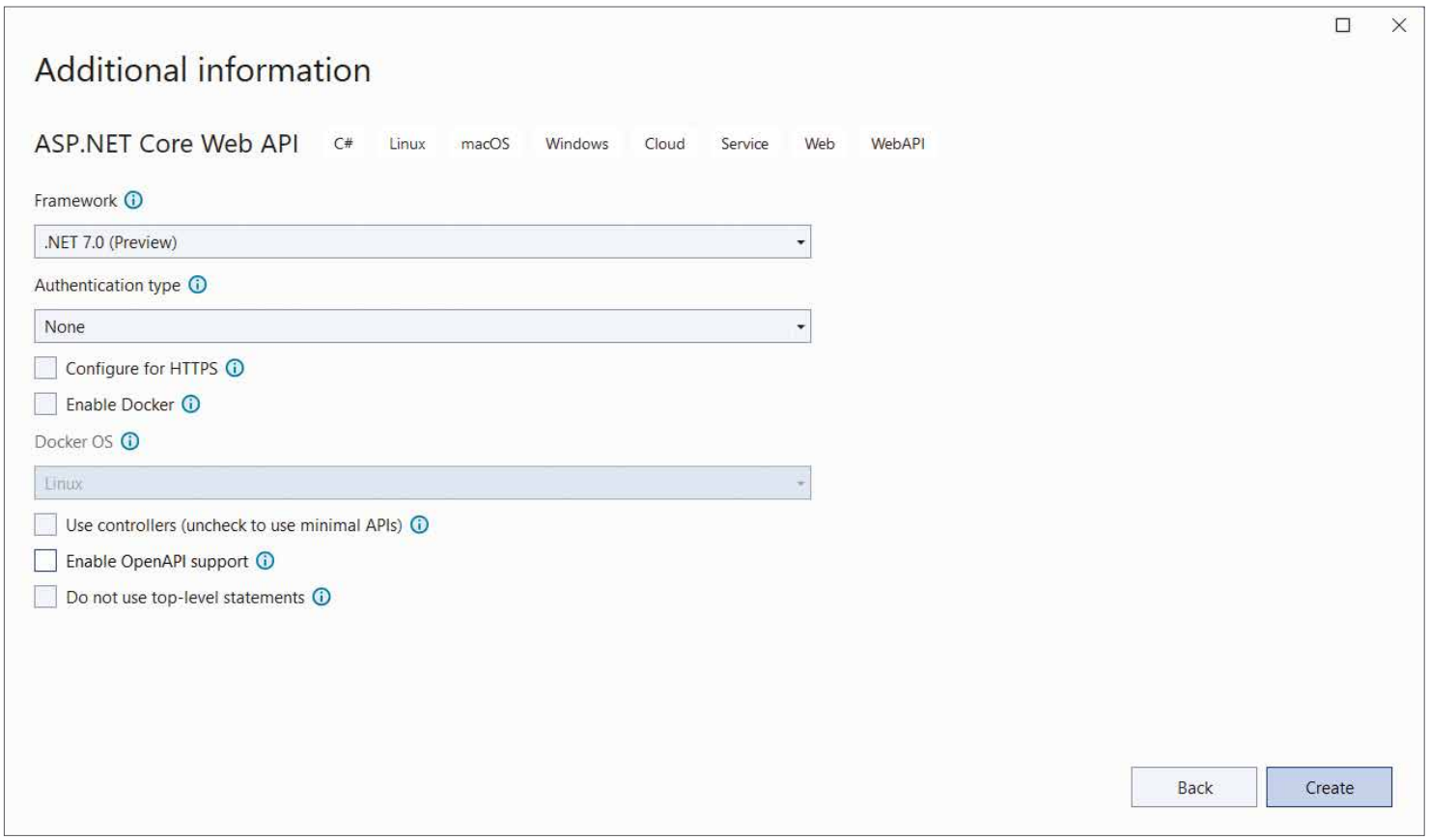


Figure 3: Specify the framework version and other metadata for your project.

Distributed Tracing Using Open Telemetry in ASP.NET 7

In this section, I'll examine how you can work with OpenTelemetry in an ASP.NET 7 application.

Create a New ASP.NET 7 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 7 Project in Visual Studio 2022 Preview:

1. Start the Visual Studio 2022 Preview 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 OpenTelemetryDemo 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 as .NET 7 (Preview) and the 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, as shown in **Figure 3**.

1. Because you'll be using minimal APIs in this example, remember to uncheck the Use controllers (uncheck to use minimal APIs) checkbox.
2. Click Create to complete the process.

You'll use this application in the subsequent sections in this article.

As the name suggests, a prerelease package is one that is still in development, is not yet fully tested, and has yet to be officially available as a stable version. For supporting the release lifecycle of a software product, NuGet provides a feature that allows it to work with prerelease packages. These packages are typically available between major releases.

You use prerelease packages when working with OpenTelemetry because these packages are constantly updated, and you can get the latest updates.

Install NuGet Package(s)

So far so good. The next step is to install the necessary NuGet Package(s) onto your project. You can install the required packages from the Developer Command Prompt for VS 2022 Preview by executing the following commands:

```
dotnet add package
--prerelease
OpenTelemetry.
Exporter.Console
dotnet add package
--prerelease
OpenTelemetry.
Extensions.Hosting
dotnet add package
--prerelease
OpenTelemetry.
Instrumentation.AspNetCore
```

The OpenTelemetry.Exporter.Console package is used for printing telemetry data at the developer console in your local computer. The OpenTelemetry.Extensions.Hosting package contains the extension methods to register OpenTelemetry into the applications using Microsoft.Extensions.DependencyInjection and Microsoft.Extensions.Hosting. The OpenTelemetry.Instrumentation.AspNetCore package is an instrumentation library that tracks the inbound web requests and collects telemetry data.

You can also install the packages inside the Visual Studio 2022 IDE by running the following commands at the NuGet Package Manager Console:

```
Install-Package
--prerelease
OpenTelemetry.
Exporter.Console
Install-Package
--prerelease
OpenTelemetry.
Extensions.Hosting
Install-Package
--prerelease
OpenTelemetry.
Instrumentation.AspNetCore
```

You'll also be able to install these packages using the NuGet Package Manager window inside the Visual Studio 2022 Preview IDE. To install the required packages into your project, right-click on the solution and then select **Manage NuGet Packages for Solution...**. Now search for these packages one at a time in the search box and install them.

Implementing OpenTelemetry in ASP.NET 7 Core

In this section, I'll examine how you can take advantage of OpenTelemetry in ASP.NET 7 Core. You'll use the application you created earlier to add support for telemetry data.

Add Support for Tracing

To add support for tracing in your application, you can write the following code in the Program.cs file:

```
builder.Services
.AddOpenTelemetryTracing
((builder) => builder
.SetResourceBuilder
(ResourceBuilder.
CreateDefault().
AddService("MyDemoService"))
.AddAspNetCoreInstrumentation()
.AddHttpClientInstrumentation()
.AddConsoleExporter());
```

Add Support for Metrics

The following code snippet illustrates how you can add metrics to your application:

```
builder.Services.
AddOpenTelemetryMetrics(builder =>
```

Listing 1: The Complete Source Code of the Program.cs file

```
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using OpenTelemetryDemo;

var builder = WebApplication.
CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddOpenTelemetryTracing
((builder) => builder
            .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyServiceName"))
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddMyConsoleExporter()
);

// Add support for tracing
builder.Services
    .AddOpenTelemetryTracing((builder) =>
builder
        .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyDemoService"))
        .AddAspNetCoreInstrumentation()
         .AddHttpClientInstrumentation()
        .AddConsoleExporter()
    );

// Add support for metrics
builder.Services
    .AddOpenTelemetryMetrics(builder =>
builder
        .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyDemoService"))
        .AddAspNetCoreInstrumentation()
        .AddConsoleExporter()
    );
// Add support for logging
builder.Host
    .ConfigureLogging(builder => builder
        .ClearProviders()
        .AddOpenTelemetry(options =>
        {
            options.IncludeFormattedMessage = true;
            options.SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyDemoService"));
            options.AddConsoleExporter();
        }));

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

```
builder.SetResourceBuilder
(ResourceBuilder.
CreateDefault().AddService
("MyDemoService"))
.AddAspNetCoreInstrumentation()
.AddConsoleExporter()
);
```

Add Support for Logging

To add support for logging in your application, you can use the code snippet given below:

```
builder.Host
.ConfigureLogging(builder => builder
.ClearProviders()
.AddOpenTelemetry(options =>
{
  options.
  IncludeFormattedMessage = true;
  options.SetResourceBuilder
  (ResourceBuilder.CreateDefault().
  AddService("MyDemoService"));
  options.AddConsoleExporter();
}));
```

The complete source code of the Program.cs file is given in **Listing 1** for your reference.

A Real-World Use Case

In this section, you'll implement a simple OrderProcessing application. To keep things simple, this application only displays one or more order records. The source code of this application contains the following classes and interfaces:

- Order class
- IOrderRepository interface
- OrderRepository class
- OrdersController class

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 Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public int ProductId { get; set; }
    public decimal UnitPrice { get; set; }
    public int OrderQuantity { get; set; }
    public decimal Amount { get; set; }
    public DateTime OrderDate { get; set; }
}
```

The Product and Customer classes aren't being shown here for brevity and also because this is a minimal implementation to illustrate how you can work with OpenTelemetry in ASP.NET 7 Core.

Create the OrderRepository Class

Now, create a new class named OrderRepository in a file having the same name with a .cs extension. Now write the following code in there:

Listing 2: The OrderRepository Class

```
public class OrderRepository : IOrderRepository
{
    private readonly List<Order> orders = new List<Order>
    {
        new Order
        {
            Id = 1,
            ProductId = 1,
            CustomerId = 2,
            OrderQuantity = 10,
            UnitPrice = 12500.00m,
            Amount = 125000.00m,
            OrderDate = DateTime.Now
        },
        new Order
        {
            Id = 2,
            ProductId = 2,
            CustomerId = 1,
            OrderQuantity = 20,
            UnitPrice = 10000.00m,
            Amount = 200000.00m,
            OrderDate = DateTime.Now
        },
        new Order
        {
            Id = 3,
            ProductId = 3,
            CustomerId = 3,
            OrderQuantity = 50,
            UnitPrice = 15000.00m,
            Amount = 750000.00m,
            OrderDate = DateTime.Now
        }
    };
    public async Task<List<Order>> GetOrders()
    {
        return await Task.FromResult(orders);
    }
    public async Task<Order> GetOrder(int Id)
    {
        return await Task.FromResult
        (orders.FirstOrDefault(x => x.Id == Id));
    }
}
```

```
public class OrderRepository :
IOrderRepository
{

}
```

The OrderRepository class illustrated in the code snippet below implements the methods of the IorderRepository interface. Here is how the IorderRepository interface should look:

```
public interface IorderRepository
{
    public Task<List<Order>> GetAllOrders();
    public Task<Order> GetOrder(int Id);
}
```

The OrderRepository class implements the two methods of the IorderRepository interface:

```
public async Task<List<Order>> GetOrders()
{
    return await Task.FromResult(orders);
}

public async Task<Order> GetOrder(int Id)
{
  return await Task.FromResult(orders.
  FirstOrDefault(x => x.Id == Id));
}
```

The complete source code of the OrderRepository class is given is **Listing 2**.

Register the OrderRepository Instance with IserviceCollection

The following code snippet illustrates how an instance of type IorderRepository is added as a scoped service to the IserviceCollection.

```
Builder.Services.
AddScoped<IorderRepository,
OrderRepository>();
```

The complete source code of the Program.cs file is given in **Listing 3**.

The OrderController Class

Lastly, create a new API controller class named OrderController in your project with the code given in **Listing 4** in there. The OrdersController class contains two action methods. Although the GetOrders action method returns a list of Order instances, the GetOrder action method returns one Order based on the Order ID passed to the action method as a parameter. The GetOrders and GetOrder action methods call the GetOrders and GetOrder methods of the OrderRepository class respectively. An instance of type IorderRepository is injected into the OrdersController using constructor injection.

Executing the Application

When you run the application, you'll be able to see telemetry data displayed at the console window, as shown in **Figure 4**:

Extending the OpenTelemetry .NET SDK

In this section, I'll examine how to extend the OpenTelemetry .NET SDK. You'll see how you can build a custom exporter and a custom processor.

Building Your Custom Exporter

You might often want to build custom exporters to send telemetry data to destinations not supported by built-in exporters.

Your custom exporter is a C# class that extends the BaseExporter class of the OpenTelemetry library, as shown in the code snippet given below:

```
class MyConsoleExporter :
BaseExporter<Activity>
{
    public override ExportResult
Export(in Batch<Activity> batch)
```

Listing 3: The Complete Source of Program.cs file

```
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using OpenTelemetryDemo;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddScoped
<IOrderRepository, OrderRepository>();

builder.Services.AddOpenTelemetryTracing(
    (builder) =>
        Builder
            .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyServiceName"))
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddMyConsoleExporter()
);

// Configure tracing

builder.Services.AddOpenTelemetryTracing(
    (builder) =>
        Builder
            .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyDemoService"))
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddConsoleExporter()
);

// Configure metrics

builder.Services.AddOpenTelemetryMetrics(
    builder =>
        builder
            .SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService("MyDemoService"))
            .AddAspNetCoreInstrumentation()
            .AddConsoleExporter()
);

// Configure logging
builder.Host.ConfigureLogging(
    builder =>
        builder
            .ClearProviders()
            .AddOpenTelemetry(
                options =>
                {
                    options.IncludeFormattedMessage = true;
                    options.SetResourceBuilder(  ResourceBuilder.
CreateDefault().
AddService("MyDemoService")
                    );
                    options.AddConsoleExporter();
                }
            )
);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

```
    {
        using var scope = SuppressInstrumentationScope.Begin();
        Console.WriteLine
("Displaying telemetry data:-");

        foreach (var activity in batch)
        {
            Console.WriteLine
($"Activity Id: {activity.Id}");
            Console.WriteLine
($"Trace Id:
{activity.TraceId.ToString()}");
            Console.WriteLine
($"Display Name:
{activity.DisplayName}");
        }

        return ExportResult.Success;
    }
}
```

Here's how you can add your custom exporter in the Program.cs file:

```
builder.Services.
AddOpenTelemetryTracing(
(builder) => builder
.SetResourceBuilder(ResourceBuilder.
CreateDefault().AddService
("MyServiceName"))
.AddAspNetCoreInstrumentation()
.AddHttpClientInstrumentation()
.AddMyConsoleExporter()
);
```

When you execute the application and hit the GetOrders endpoint, the activity details will be displayed at the developer console window, as shown in **Figure 5.**

Listing 4: The OrderController Class

```
[Route("api/[controller]")]
[ApiController]
public class OrdersController :
ControllerBase
{
    private IOrderRepository _orderRepository;
    public OrdersController(IOrderRepository
    orderRepository)
    {
        _orderRepository = orderRepository;
    }

    [HttpGet("GetOrders")]
    public async Task<List<Order>> GetOrders()
    {
        return await _orderRepository.GetOrders();
    }

    [HttpGet("{id}")]
    public async Task<Order> GetOrder(int id)
    {
        return await _orderRepository.GetOrder(id);
    }
}
```

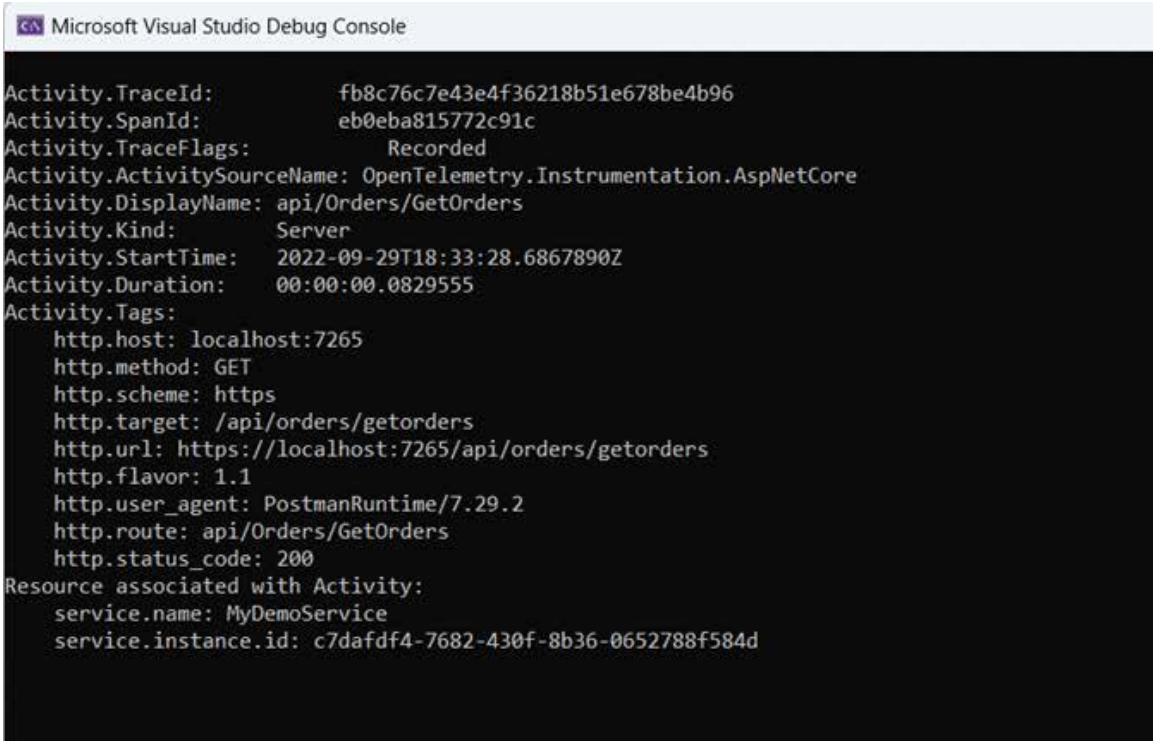


Figure 4: Telemetry data displayed at the Developer Console Window

```
Displaying telemetry data:-
Activity Id: 00-d3d20d3401287868500e8fc6f9f2ed3a-ca855a53331548ed-01
Trace Id: d3d20d3401287868500e8fc6f9f2ed3a
Display Name: api/Orders/GetOrders
```

Figure 5: Displaying activity details using a custom exporter

Build a Custom Processor

To create a custom processor, create a C# class that extends the BaseProcessor class of the OpenTelemetry library, as shown in the code snippet given below:

```
class MyProcessor :
BaseProcessor<Activity>
{
   public override void OnStart
(Activity activity)

   {
      Console.WriteLine
      ($"OnStart called:
      {activity.DisplayName}");
   }

   public override void OnEnd
(Activity activity)
   {
      Console.WriteLine
      ($"OnEnd called:
      {activity.DisplayName}");
   }
}
```

Export Telemetry Data

OpenTelemetry data can be exported to back-ends using trace exporters. Some of the popular trace exporters available are:

- Jaeger
- Zipkin
- OLTP gRPC
- OLTP HTTP

In this example, you'll examine how you can export telemetry data using Jaeger. To export your telemetry data using Jaeger, here's what you'll need to do:

1. Install Docker Desktop from here: https://www.docker.com/products/docker-desktop/.
2. Install the OpenTelemetry.Exporter.Jaeger NuGet package.
3. Configure Jaeger Exporter in the Program.cs file.
4. Use Docker-Compose to Start Jaeger.

Install the OpenTelemetry.Exporter.Jaeger NuGet Package

You can install the OpenTelemetry.Exporter.Jaeger package from the Developer Command Prompt for VS 2022 Preview by executing the following commands:

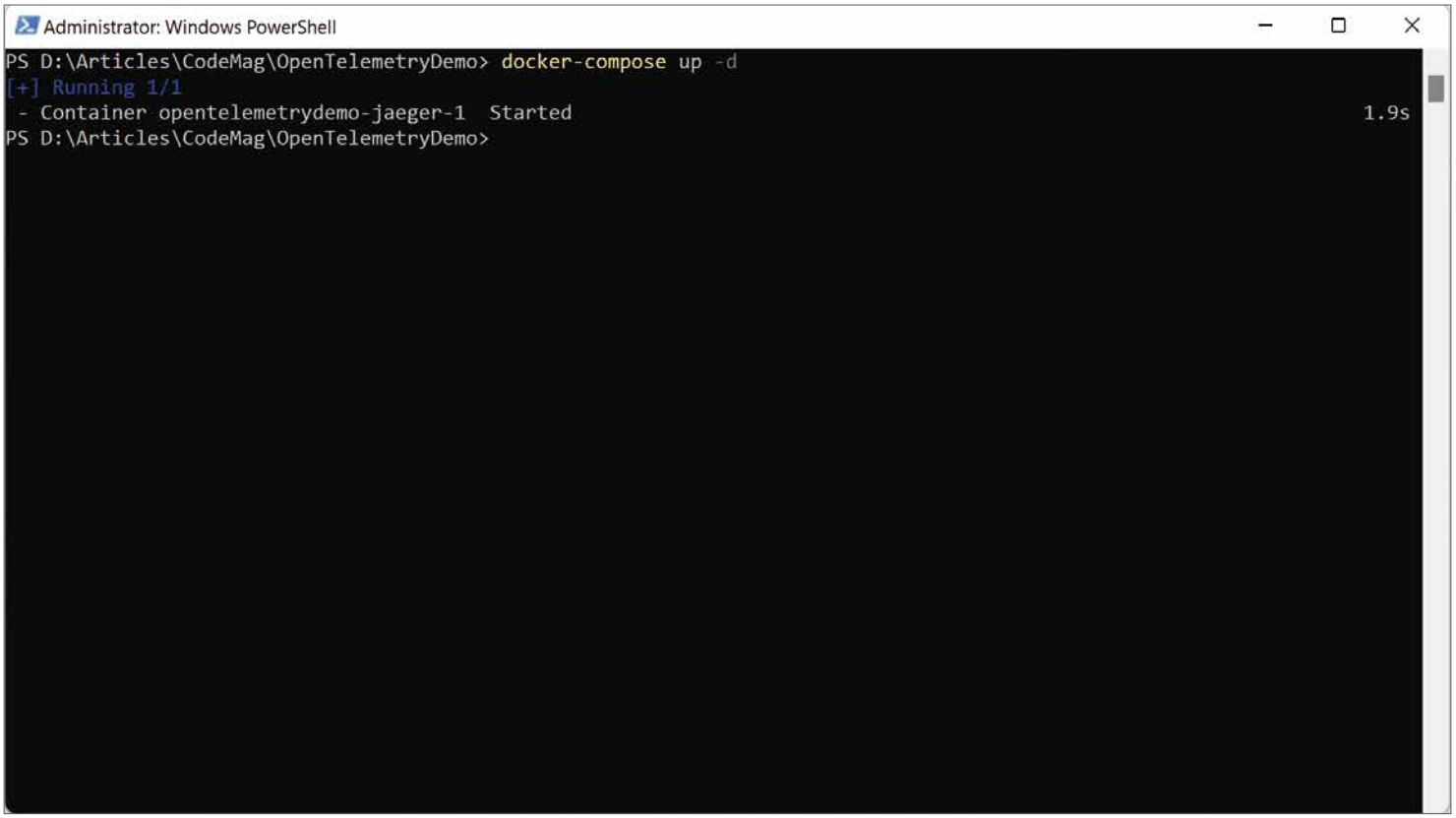


Figure 6: The docker-compose command starts a container

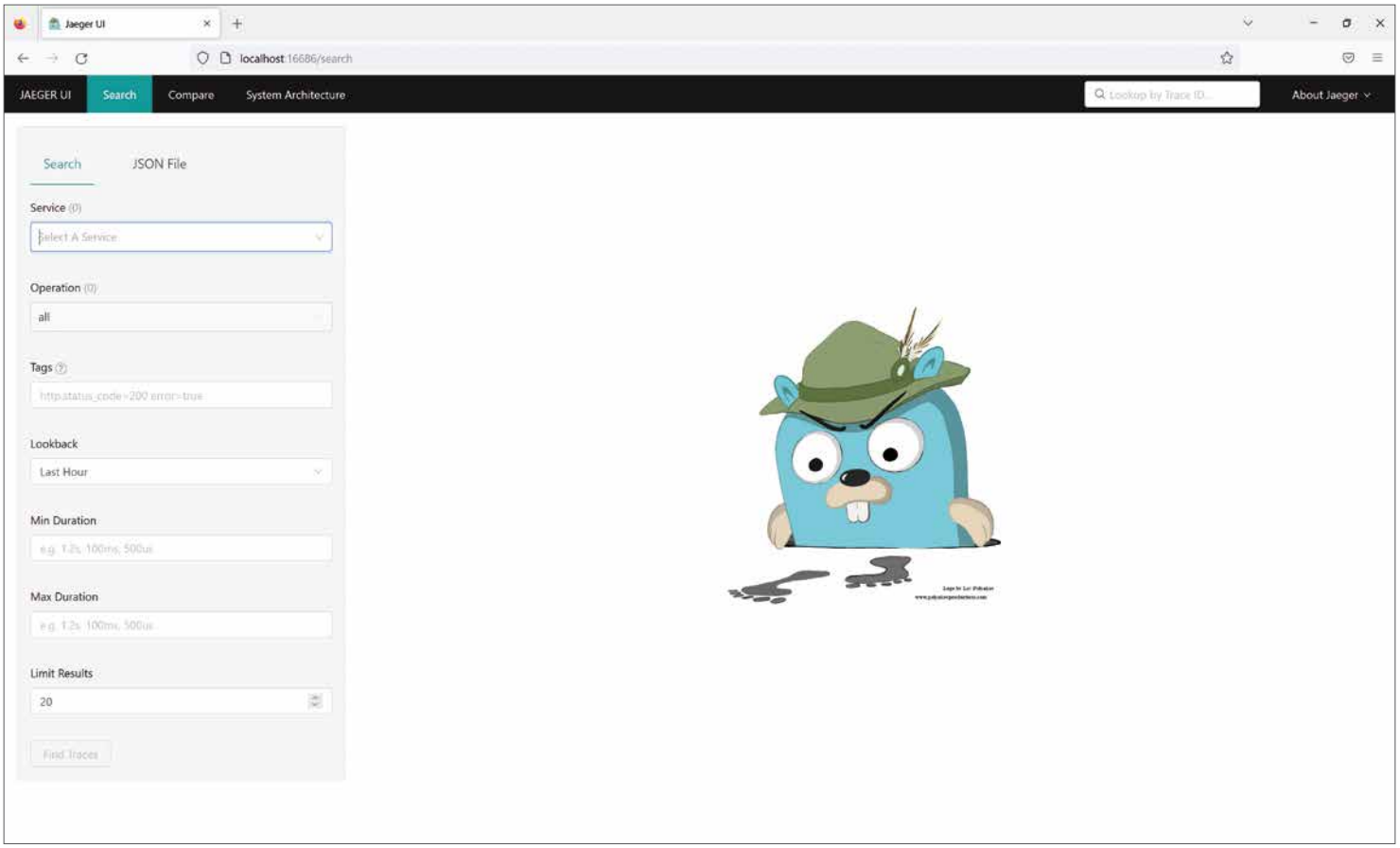


Figure 7: The Jaeger UI

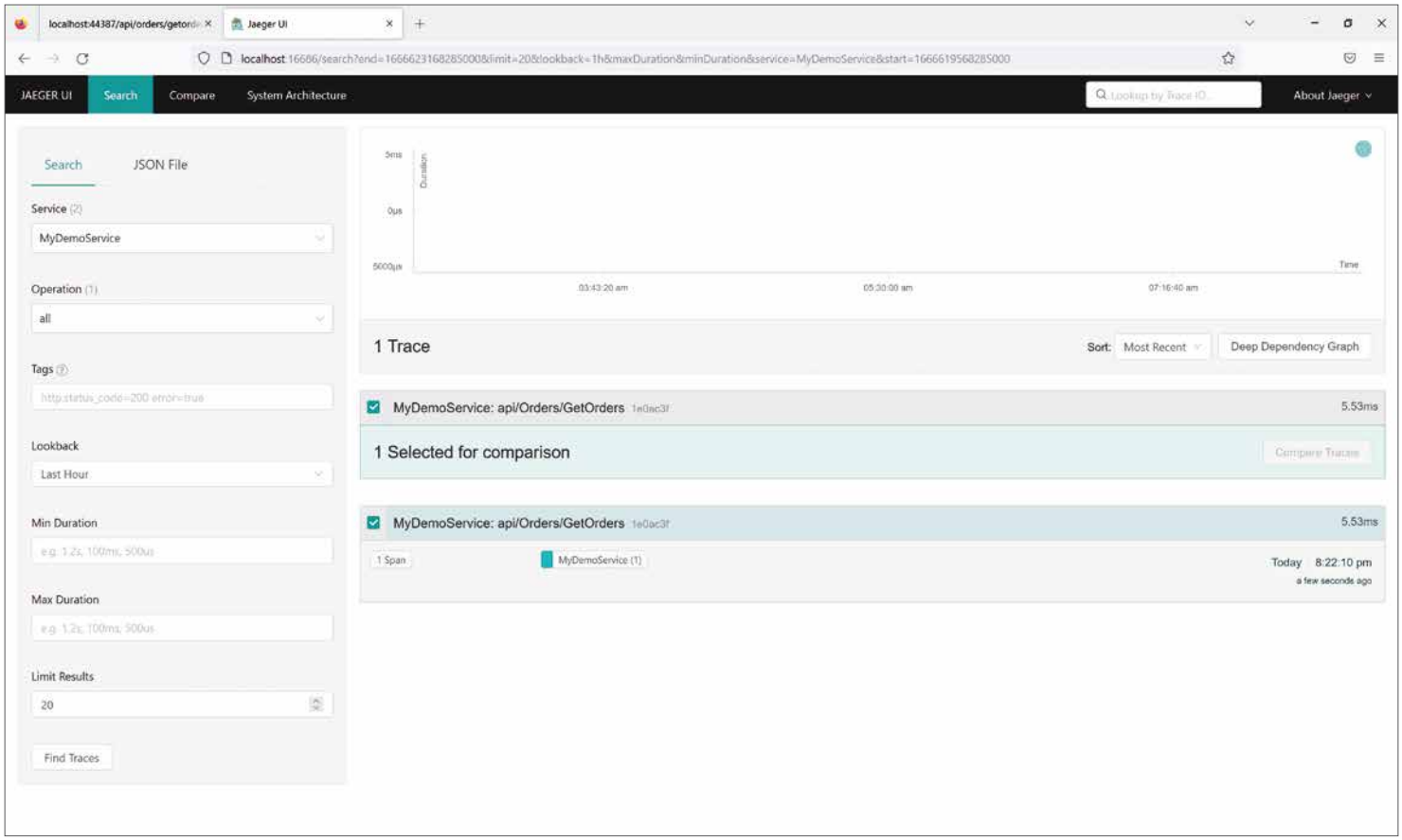


Figure 8: Viewing MyDemoService traces in the Jaeger UI


```
dotnet add package
--prerelease
OpenTelemetry.Exporter.Jaeger
```

You can also install the package from the NuGet Package Manager windows inside the Visual Studio 2022 Preview IDE.

Configure Jaeger Exporter in the Program.cs file

Now, replace the code you wrote earlier to enable OpenTelemetry tracing using the following piece of code:

```
builder.Services.
AddOpenTelemetryTracing
((builder) => builder
    .SetResourceBuilder
(ResourceBuilder.
CreateDefault().AddService
("MyDemoService"))
    .AddAspNetCoreInstrumentation()
    .AddJaegerExporter()
);
```

This allows you to export your telemetry trace data using Jaeger. The call to the AddAspNetCoreInstrumentation method enables ASP.NET Core instrumentation for your application at application startup.

Use Docker-Compose to Start Jaeger

Assuming Docker Desktop is already running in your system, you can use the following command to start Jaeger:

```
docker-compose up -d
```

You can write the above command in a PowerShell window or a Command window. In either case, ensure that you open the windows in administrator mode, as shown in **Figure 6**.

The Jaeger UI

Assuming that Jaeger started successfully, you can browse the default URL of the Jaeger UI, as shown in **Figure 7**. By default, the Jaeger UI can be viewed at http://localhost:16686.

Finally, execute your application in a web browser. You can now see the traces in the Jaeger UI for your service, as shown in **Figure 8**:

Conclusion

OpenTelemetry is an open-source project that provides tools for distributed tracing. It includes libraries for different programming languages, including .NET. OpenTelemetry can be used to prepare applications for distributed tracing. It also provides a set of APIs that can be used to collect trace data from a variety of sources.

Joydip Kanjilal

(Continued from 74)

etc. audits? Have you ever had to respond to such audit requests? Perhaps you've implemented tools related to Continuous Integration/Continuous Delivery (CI/CD), Agile, or some other aspect of DevOps. Assuming that some record exists, are its contents verifiably complete and accurate? Can you peel back the onion layers to get to the core? Can you engage in timely, efficient, and successful root-cause analysis such that you can make informed conclusions based on a verifiable context (data, environment, or any other relevant thing)?

Verifiability is precisely what an audit is all about. Is the published application based on the last published specification? If a sufficient record, premised on sufficient environment, exists, your deployment process should be able to quickly respond pursuant to a standard operating procedure: "Yes, the latest published application is based on the latest spec because**...."**

It's an easy question, and one that should be easy to answer with as little effort as possible. For anything audit related, despite its importance and necessity, strive to make asking and answering such questions as mundane as possible. Otherwise, the more effort and energy spent on the mundane, the fewer resources available for new feature development.

Here are some quick tips to improving your shop's ability to address those words that come after "because:"

- Meet as a team to get consensus on **how** decisions will be made. You're not pre-determining what the decisions will be. You're finding out what your rubrics for decision-making are. Set out clear rules for engagement in argument and debate, that such things should never be personal. Check egos at the door and hold each other accountable.
- Adopt a Definition of Done (DoD) and agree that it's inviolable. Just as there's a process for how decisions are made, among the many alternatives, there's a software development process that will work well for your shop. It's far better to hold something from deployment for quality reasons than to let it pass and ruin reputations, lose money, and create problems for yourselves and your customers. That's how technical debt unreasonably and unnecessarily accumulates. The entire point of stopping problems before they start is to better manage technical debt accumulation.
- Adopt a two-prong test when analyzing facts: relevance and probative value. Here's another borrowed item from the law: Relevance is about how "closely connected" one thing is to another. How close something gets to being relevant is a question for the entire team. Irrelevant facts are distractions. This is an important area because if there's disagreement about relevance, that means one of two things: somebody is creating a dependency relationship when there's none, or there's an unacknowledged dependency in the system. The second testing prong is probative value. In other words, can it be verified? Is the fact what it purports to be? Is it credible? Scrutinizing facts can be time-consuming and difficult, but facts are the foundations of your decisions. It's important to disregard facts as quickly as possible that are either not relevant or have no probative value.
- It's important to have and maintain a record of what happened. When it comes time to be able to demonstrate what happened to some third party, is there a system you can rely upon? A credible auditor can't just take someone's word for it. Remember, the auditor is assessing the facts you supply for relevance and probative value too. There are many tools that support processes concerning how we build software, such as Azure DevOps, GitHub, Atlassian, Jenkins, Team City, Rally, etc. As tools, they rely upon people and processes to work correctly and deliver the expected benefits. There's perhaps no other area so rife with debate than tools, their selection, which is best, how best used, etc. The point here is that it's not only important to log your application's activities. It's just as important to log how your team decides things, so you can demonstrate later what happened in the past. Without a discernable record, any response to the question of what happened is, at best, just an opinion of what individuals recall. They may be accurate or entirely off, you can't know.
- Embrace Dr. W. Edwards Deming's research and actively look to eliminate dependencies on manual inspections. The more that can be systemized and automated, the less it's subject to the variable of human intervention. With systemized consistency, the more resilient your processes can be and, perhaps more importantly, the more comparable they become through time because you've recorded how the automated processes have modified over time. These automations are at the heart of unit testing and continuous integration and deployment (CI/CD) pipelines today in modern DevOps implementations.
- If you aim to improve quality, you must understand the causal factors of success, which are learned through understanding the causal factors of failure—the "because" of it all. You need to be able answer "We succeeded because…" and "We failed because…" This can be a long and daunting journey but it's not only worth the time and effort, it's also necessary.

John V. Petersen

CODE

Jan/Feb 2023
Volume 24 Issue 1

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Contributing Editor
John V. Petersen

Content Editor
Melanie Spiller

Editorial Contributors
Otto Dobretsberger
Jim Duffy
Jeff Etter
Mike Yeager

Writers In This Issue
Bilal Haidar
Wei-Meng Lee
John Petersen
Joydip Kanjilal
Sahil Malik
Paul D. Sheriff

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@codemag.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: American News Company (ANC)

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $50.99. 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, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379
Phone: 832-717-4445

Why "Because" Matters

Some words have more importance and relevance than others. In the decision-making context, there may be no more important word than "because." The word "because" marks a line between a conclusion and a premise. For example, "We chose a specific messaging or database application because…." In this issue's CODA, I'm going to address one of my favorite topics: the process of how we go about deciding important matters and how we support those decisions.

In almost any technology decision of any importance, there'll be at least a few meetings, more than a few long email and chat threads, some arguments, more debate, and lots of going back and forth with the mix of facts. The team must sort through all of this as a group and as individuals in order to plot solutions and decide in the context of business risk, plus constraints and objectives, which alternatives are better than others, and how it's all ranked.

How do we go about that? How do we decide? And more specifically, how do we **objectively** make decisions when we're surrounded by so much data, some of which is presented in a subjective and skewed manner? Just like everything else in our world today, there are patterns, practices, frameworks, and rubrics for decision-making that can provide guidance. If you're thinking that subject matter areas like DevOps and Agile apply, you're correct.

But there's something more fundamental that we must rely upon for guidance. That fundamental thing is logic. And for illustrative purposes, I'm going to use the legal world as an analog.

In the law, basic guidance comes from the law itself in the form of statutes, ordinances, and regulations, and how those are applied in the context of facts, to determine liability or guilt. Which facts are more important and relevant than others? Which facts are more determinative to resolving the matter? Every legal proceeding is about resolving materially disputed facts in the context of applicable law. If there are no disputed facts, there's no need for a proceeding. We simply fast forward to applying the facts to the law and then decide. In fact, that's what a motion for summary judgment is all about. In the prelude to a trial or litigation, there's also discovery where we gather, assess, and analyze facts. We depose, submit written questions, subpoena documents, and assemble related information. If the issue needs to stand trial, we litigate the facts in the context of the law under the rules of procedure.

The commissioning, development, and delivery of a software project is substantively not that different from a trial. We march toward a solution, encountering facts to assess and analyze along the way. We ask for accommodations, some of which we may get in whole, in part, or not at all, based on the organization's policies and procedures. In a trial, motions for various kinds of relief are made often. The most common one is when an attorney objects to testimony or to the introduction of evidence. Everything stops at that point, pending the judge resolving the matter. Typically, the judge addresses the other attorney, asking for a rebuttal. This is what **argument** is all about. It's important to understand what argument is, what it isn't, and when it's appropriate.

Technicians and engineers tend to be people with very definite opinions about certain matters. For example, one team member may have an extreme preference for a certain open-source library. Others on the team may have an opposite opinion. Yet others may be completely indifferent. Hashing these differences out where the best alternatives for the organization may be further reviewed is a necessary activity. Depending on the personalities, some of this "hashing out" may be smoother and more conflict-free than others. This is where subjectivity and skewed facts can enter the process. Undue bias in decision-making often leads to an unintentional assumption of risk because facts aren't seen for what they are. Rather, they are seen for what some or all wish they'd be.

Words like argument and debate are often viewed as negative things because too often, technical debates turn into religious disagreements where each participant comes from a very personal space. Argument SHOULD be just an exchange of opposing views as to facts. Which facts? Which views? How are those views formulated? In other words, if one concludes and responds in kind that "Yes, our API can and will scale to the planned increased sales volume without modification," it begs the question of "Why?" Why is it capable of scaling? Was it designed to scale? Has it been tested and verified?

It's those additional things that stand apart from the conclusions that can be objectively reviewed. In other words, a conclusion without a sufficient premise is just a guess, out of thin air. The guess may indeed end up with success. What's the likely percentage of success? We can guess, but there is no objective basis for others to assess and analyze the facts. The point of the entire exercise is to objectively assess facts that are likely to lead to the best outcome for the organization. It may very well be that what was thought to be the best alternative isn't, in the end. That would be regarded as a failure. Therefore, to improve and better support the organization, it becomes necessary to understand why failure occurred. In other words, we must be able to say, "The process failed because…."

The critical piece to making everything work is a platform where people can collaborate on analyzing and assessing the facts. The primary constituent in this case is the business and its operations, not our individual egos. Cultures of success not only tolerate dissenting points of view, but dissent is openly encouraged. Any person making a conclusion or offering a suggestion has the burden of proof to reasonably support their position in a way that facilitates group understanding. It's unreasonable to expect others to know and experience what each participant is thinking. Success is all about the necessity of transparency.

Imagine there was no transparency in a legal proceeding. There'd be no understanding of why liability or guilt was assessed. In a typical legal proceeding, there are two important devices, the docket and the transcript. The docket is an enumeration of every document, motion, hearing, order, etc. The transcript is the record of what was said and what happened during the trial or other legal proceeding. In any appeal in support of an assertion of error, the transcript is essential because it's the only way the reviewing tribunal can know what happened, and, in light of the rules, assess whether the earlier decision reached was in error and should be reversed.

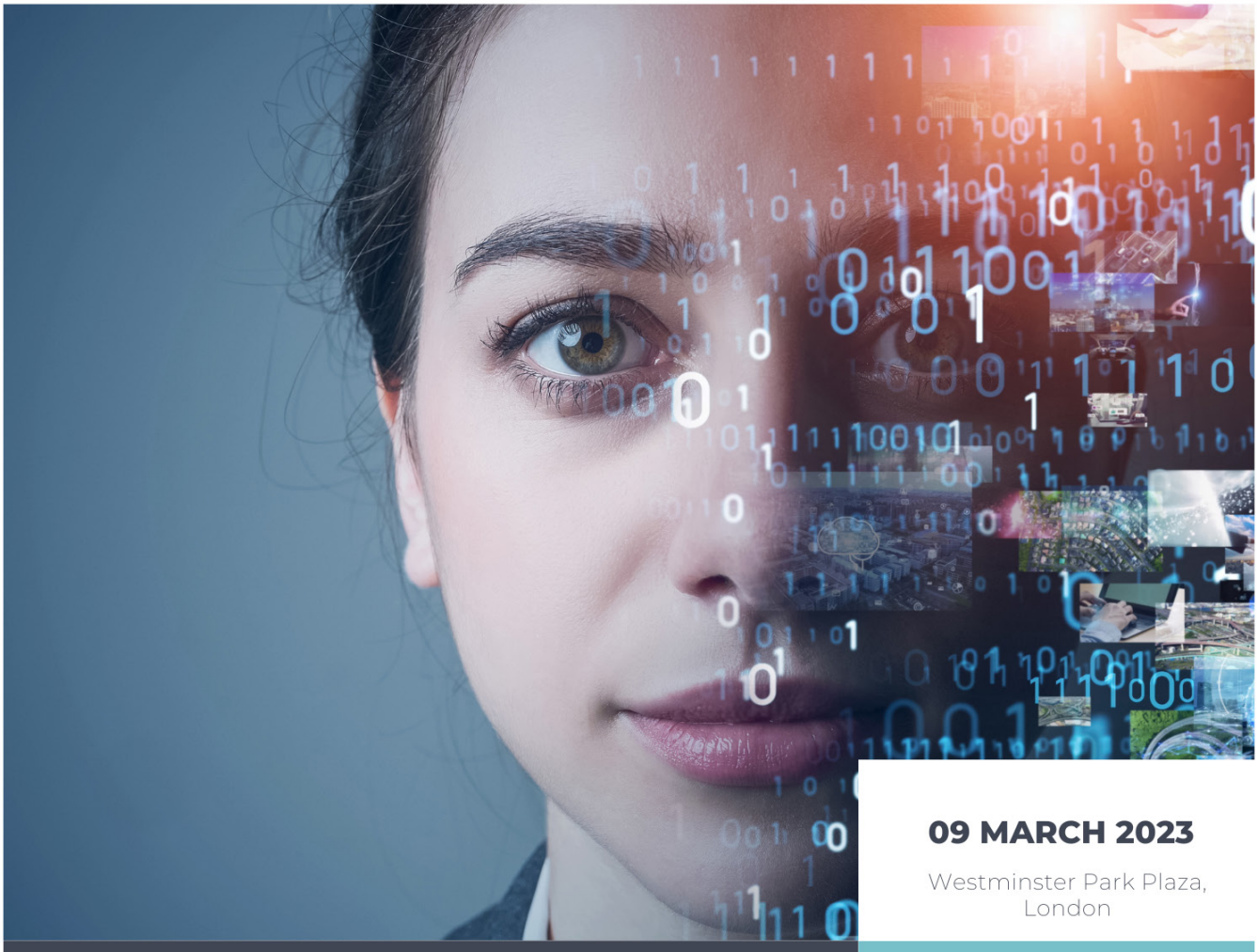
If you had to replay the tape to determine why a certain failure is being countered, could you?

The first thing you need is the tape itself. Is there any record? Or is it all tribal knowledge held in individual team members' recollections? Is your organization subject to SOX, SOC, HIPPA, PCI,

(Continued on page 73)

everywoman in Tech Forum

People.
Purpose.
Progress.



09 MARCH 2023

Westminster Park Plaza, London

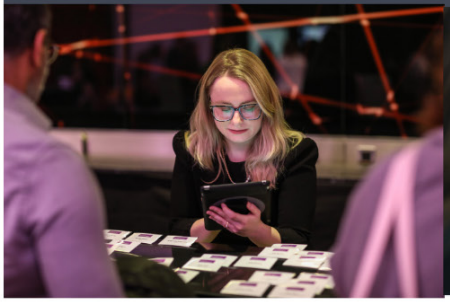
People. Purpose. Progress.

The everywoman in Tech Forum brings together a community of over 500 women working in Tech, from graduates through to the C-level. Women come to the Forum to be inspired as to what they can achieve, to be supported by their peers from around the world and to learn key skills and knowledge to support them in their careers.

GET YOUR DELEGATE PASS TODAY

Use 'CODE10' for a 10% discount, and book before January 30th to save £100 on a combi ticket to join us at our everywoman in Technology Awards.

Head to www.everywoman.com/techforum for more information.



CODE
LEGACY

NEED MORE OF THIS?

Is slow outdated software stealing way too much of your free time? We can help.
We specialize in updating legacy business applications to modern technologies.
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/modernize
832-717-4445 ext. 9 • info@codemag.com