Text, Text, Text, Text, Text, Text, Text

CODE

JUL
AUG
2022

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

EVENT SOURCING

Text Text Text
Text Text

Text Text Text
Text Text

Text Text Text
Text Text

DevI

ntersection

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.

LEAD

Rod Paddock

ONLINE QUICK ID 2209021

FIDO2 and WebAuthn

Authentication has been an essential part of applications for some time now because applications need to know some information about the user who's using the application. For the longest time, the solution to this has been username and passwords. Username passwords are popular because they're convenient to implement. But they aren't secure. There are many issues with passwords. First, there's the problem of transmitting this password securely. If you send the password over the wire, a man-in-the-middle could sniff it. That pretty much necessitated SSL over such communication or the equivalent of creating a hash of the password that's sent over the wire instead of the actual password. But even those techniques didn't solve the problem of the server securing the password, or a secure hash of the password. Or, for that matter, keeping you safe from replay attacks. Increasingly complex versions of this protocol were created, to the point where you could, with some degree of confidence, say that you were safe from man-in-the-middle attacks or replay attacks.

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.

Users created a simple, easy to remember password, and brute force techniques guessed those passwords. So we came up with complex requirements for passwords, such as your password must contain an upper case, lower case, special character, and minimum length—and yet people still picked poor passwords. When they didn't pick poor passwords that were easy to remember, they would reuse passwords across different systems. Or they would use password managers to store their passwords, until the password manager itself got compromised.

But even then, you're not safe from passwords being leaked. Worse, leaked passwords are not detected—you don't know if your password has been leaked until the leak is discovered. And these leaks could occur on a poorly implemented service. This means, no matter what you do, you're still insecure.

Don't Despair

There are solutions. There are concepts like MFA or one-time passwords that can be used in addition to your usual password. This is what you've experienced when you enter a credential, but in addition, you have to enter a code sent to you via SMS or from an authenticator app on your phone.

MFA and one-time passwords are great. In fact, I'd go to the extent of saying that if there's a service you're using that uses only username password, just assume it's insecure, and don't use it for anything critical. Additionally, pair it with common-sense practices like own your domain name, and a separate email address from your normal use email address for account recovery. Secret questions and answers that aren't easy to guess, and answers that don't make sense to anyone.

As great as MFA and one-time passwords are, they're still not a perfect picture. There are a few big issues with this approach.

First, they are cumbersome to manage for the end user. I work with this stuff on a daily basis, and I find it frustrating to manage 100s of accounts, multiple authenticator apps, and I worry that if I ever broke my phone accidentally, I'd be transported to neanderthal times immediately. I can't imagine how a common non-technology-friendly person deals with all this.

Second, MFAs and one-time passwords are both cumbersome and expensive for the service provider. All those SMS messages and push notifications cost money. This creates a barrier to entry for someone trying to get a service off the ground. Then there's the question of which authenticator app to trust and whether that app be trusted. Is SMS good enough?

Third, there's the issue of phishing. As great as MFA is, someone can set up a service that looks identical to a legit service, and unless you have very keen eyes watching every step, you may fall for it. Unfortunately, even the best of us is tired and stressed at times, and that's when you fall for this. In fact, the unscrupulous service that pretends to be a legit service could simply forward your requests to the legit service after authentication while stealing your session. So you may think everything is hunky dory but your session has effectively been stolen.

Finally, there is authentication fatigue. Hey, I just want to login and use a system. Zero trust dictates that you assume a breach, so it's common for services to over-authenticate. This creates authentication fatigue, and an already fatigued user could blindly approve an MFA request, especially if it's cleverly disguised. It only takes one mistake for a hacker to get in the house, then they can do plenty of damage, potentially remaining undetected for a long time.

What am I Trying to Solve?

I'm not trying to secure passwords or make a better MFA solution here. The fundamental problem I wish to solve here is how an application can securely trust a user's identity, such that the identity is not cumbersome to manage, is secure, convenient, and not stealable.

Let's refine this problem further. The problem I'm really trying to solve is that a user goes through a registration process, typically the first time you encounter the user. The next time the user shows up on the application, you want to make sure it's the same person behind the user ID. You want to do so with 100% confidence, and you want to do so with relative ease for users and the application.

If you had a clean slate to architect this with the technology available today, how would you do it?

Imagine if, during registration, the user generates a key pair. This is a typical certificate. There's a private key, and there's a public key. With the private key, you can sign stuff, you can encrypt stuff, but you never share that private key. You can keep the private key in your private possession forever. But the public key is public information. With the public key, you can only decrypt or verify the signature.

During registration, you generate a key pair that's unique for the service, and the public portion of that key pair is shared with the service. The server then stores it securely and connects it with this particular user (you).

Next time you wish to authenticate, the server generates a challenge that's just a random string. This challenge is communicated over to the user securely over HTTPS. This challenge is encrypted or signed by you using your private key that's unique to this service. This encrypted or signed string is now sent back to the service. The service can now validate the signature via the public key associated with the user.

This sounds like something that could work, but a few interesting things happened here. At no point was the private key communicated anywhere except on your device. As you'll see later in this article, there are plenty of hardware devices that allow for the storage of this private key securely.

There's also no need for a cumbersome MFA prompt or the maintenance of it. There's also no risk of SMS spoofing or your phone number rolling over to the next subscriber.

There's no additional cost for the service to send MFA prompts. The service just needs to remember a mapping of the public key with the user, so it's no worse than remembering a password. This can be paired with existing MFA techniques, if you choose to do so.

Sounds like we're on to something interesting here. Let's dig further.

Hardware Support

I've boiled the problem down to the much simpler problem of keeping your private keys secure. The good news is that there exists a lot of interesting hardware to help you do so. These look like USB keys, or even NFC keys, that securely store your private keys. An example of this key is a YubiKey, as shown in **Figure 1**.

These private keys require some interaction from the user to extract the private key briefly when it is needed. Additionally, you now see things like trusted platform module (TPM) requirements baked into Windows 11 and MacOS/iOS, supporting things like TouchID on Macs and FaceID on phones, which, paired with secure enclaves, give you a pretty neat solution around storing these keys securely in iCloud.

The Apple ecosystem moves faster because they have full control on the end-to-end story, and it also helps that they make a phone. I'm particularly excited about the possibility of roaming these keys using iCloud. What this means for the end user is that they just use their devices as they normally do. They don't need to carry a separate dongle or device or risk losing it. And yet they gain the convenience of never having to bother with a password while remaining secure. This is the holy grail of security—security and convenience, so users won't try to work around inconveniences. All this is pretty new at the time of writing this article; the support for this technology was introduced in iOS 15 and future versions of OSs will improve this and make it more accessible. I'm quite excited about where this is headed.

Hardware aside, you probably need a common understanding of protocols for this standard to be implemented, right? Let's talk about that next.

Protocols

The overall concept sounds great, but if various services don't speak a common language, this concept will never gain foothold. This is why this concept has been solidified as protocols. Like anything else in identity, protocols around this concept have been evolving.

The word "FIDO" comes from the FIDO alliance, which is the organization pushing for this standard. You can check them out at *https://fidoalliance.org*. If you check out their website, you'll see them describe specs on UX guidelines around strong authentication, but more interestingly, they talk of specific specs such as FIDO universal second factor (FIDO U2F), FIDO universal authentication framework (UAF), and FIDO2, which includes W3C's Web authentication (WebAuthn) spec, and FIDO client-to-authenticator protocol (CTAP).

All right, that was a lot of acronyms I just threw at you. Let's break it down in **Figure 2**. FIDO2 is the umbrella term of what I'm concerned with here. When the user needs to register or authenticate, they interact with an external authenticator or a platform authenticator. An external authenticator could be a USB key, such as a YubiKey. It looks just like USB flash storage but may have additional biometric protection on it. Or the user could use a platform authenticator, such as FaceID, TouchID, Windows Hello, etc. When the user interacts with a relying party (the service you are trying to access), it uses a protocol called WebAuthn.

Registration Process

When a user first lands on a site, they create an account. This is called the registration process. Here's how it would work if you were to do this under the FIDO2 protocol.

The user lands on the site, and says, "hey, I want to register a key." The server then generates a challenge, a random string, and passes it over TLS to the user along with a bunch of other information. A critical part of this information is the relying party ID. The relying party ID must match the TLD or top-level domain of the site the user is on. This is verified with the SSL cert being used by the server. Once the client has verified the identity of the server, the client then

Figure 1: YubiKeys

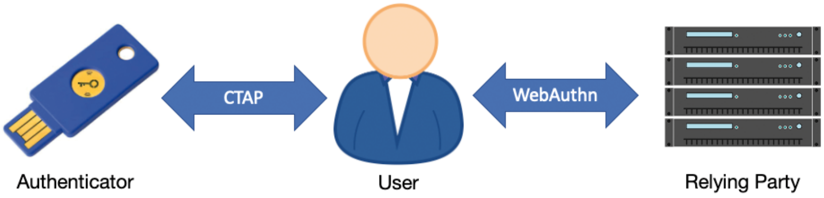


Figure 2: FIDO2 and its moving parts

generates a public-private key pair. The private key is never sent over the wire. But the public key, along with the signed challenge, is sent back to the server. Along with this, it also sends a credential ID generated by the security key.

The server then verifies the signed challenge with the public key. If it passes signature verification, the server then stores the credential ID and the public key, and sets the counter to zero. Every time an authentication is performed, this counter increments, to prevent the cloning of keys. There should be only one instance of the key in the wild, and if the counter isn't sequential, authentication is denied. This entire process can be seen in **Figure 3**.

Authentication Process

At a later time, the user lands on the site and wishes to authenticate themselves. The server communicates back to the user a randomly generated challenge, which is just a string, and a list of credential IDs for the user.

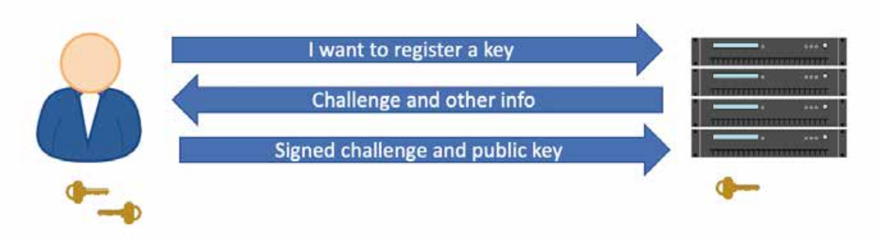


Figure 3: The registration process

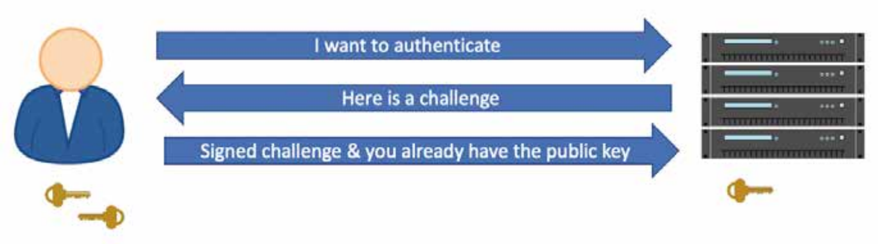


Figure 4: The authentication process

Why are there multiple credential IDs? It's because you want to support more than one key per user, just in case one key gets lost. Or perhaps one key lends you a greater level of access than the other.

The user now receives the challenge. At this point, the user's computer verifies the server identity, and uses the credential ID to find the appropriate key. It then increments the counter so it stays in synch with the server, and it signs the challenge using the private key.

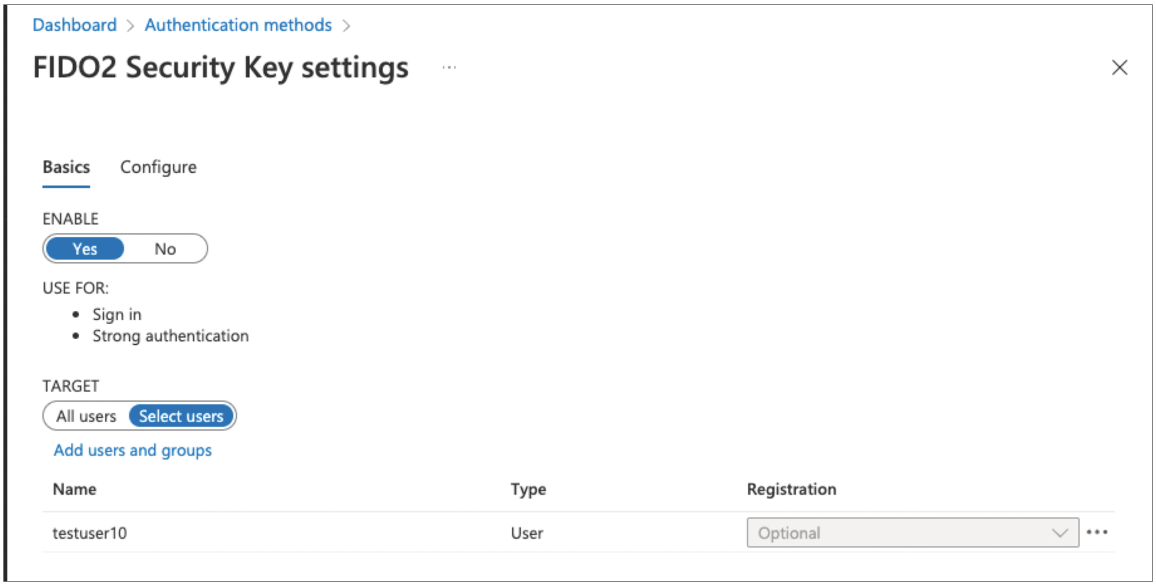
This challenge is then communicated back to the server, which then verifies the signed challenge with the public key, and increments the counter. This entire process can be seen in **Figure 4**.

Set Up FIDO2 Auth in Azure AD

Many websites support FIDO2. Microsoft has been a prominent participant in this ecosystem as well, along with Google, AWS and many others. Azure AD fully supports FIDO2 authentication for its users. This means that if your application uses Azure AD for authentication, you can make use of FIDO2 easily today. Additionally, you can also lock down access to critical resources such as the Azure Portal or Office 365 using FIDO2. Strong Auth that's convenient for users is a win for everyone. If users use FIDO2, they are less susceptible to MFA fatigue and accidentally completing the MFA challenge. They're also more secure as a result.

Let's see how you can go about setting up FIDO2 authentication in Azure AD. To follow through these steps, you'll need a physical FIDO2 key. I'm using a YubiKey.

Start by logging into the Azure Portal at portal.azure.com as a tenant admin, and then navigate to the Azure Active Directory blade. In that blade, look for Security, and navigate to Authentication Methods. Here, under policies, choose **FIDO2 security key**, and choose to enable it for select users. As you can see in **Figure 5**, I've chosen to enable it for testuser10.


Dashboard > Authentication methods >
FIDO2 Security Key settings
Basics
Configure

ENABLE
Yes ● No

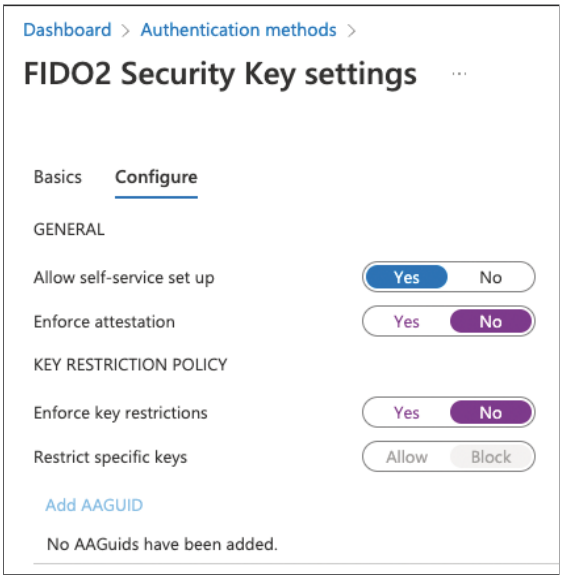
USE FOR:
- Sign in
- Strong authentication

TARGET
All users ● Select users

Name	Type	Registration
testuser10	User	Optional

Figure 5: Enabling testuser10 to use FIDO2 as an authentication method

Note that, optionally, you can configure FIDO2 at the tenant level under the Configure tab in the same area. This can be seen in **Figure 6.** There are a number of settings here, and you can allow users to set up FIDO keys themselves. The other option is for an admin to set these keys for the user. You can choose to enforce attestation or not. Usually, attestation is useful in enterprise scenarios where you want to disallow certain keys from being used. However, using any key is better than a username password, so it's okay to leave this set to "no". You can restrict keys to certain well-known keys, so users don't just buy their own keys and start registering them. You'd do this by adding AAGUIDs of the keys. This creates a huge management overhead, but it's incredibly secure.


Dashboard > Authentication methods >

FIDO2 Security Key settings ...

GENERAL

Allow self-service set up [✓ Yes] No

Enforce attestation Yes [✓ No]

KEY RESTRICTION POLICY

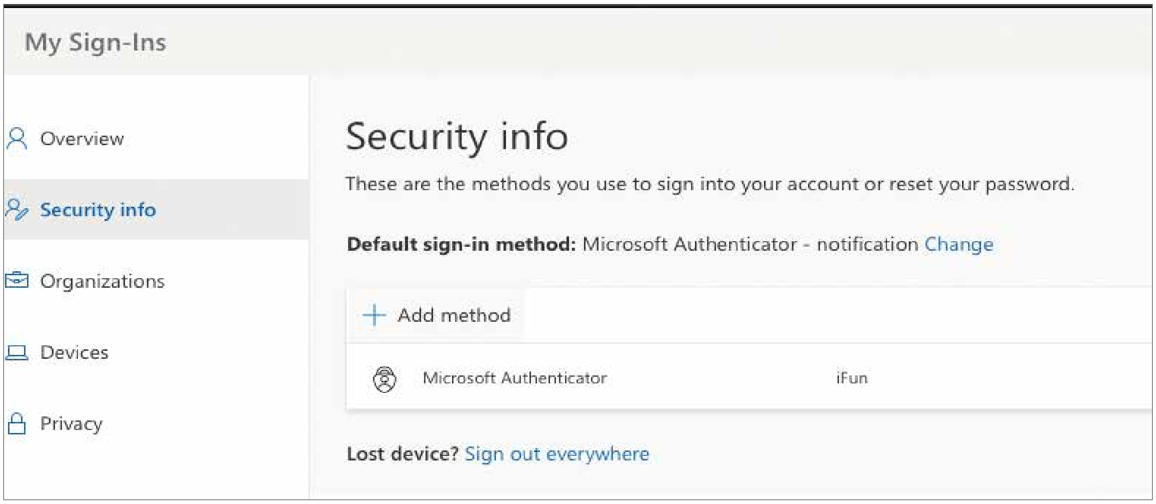
Enforce key restrictions Yes [✓ No]

Restrict specific keys Allow Block

No AAGuids have been added.

Figure 6: Configure FIDO2 settings at the tenant level.

As you can see from **Figure 6**, I've allowed self-service set up. Also, as you can see from **Figure 5**, I've enabled testuser10 to use FIDO2 as an authentication method. In a relatively modern browser (I'm using Chrome), in a non-private window, visit *https://myprofile.microsoft.com* and sign in as the user you've enabled FIDO2 authentication for. In my case, that's testuser10@sahilmalikgmail.onmicrosoft.com. Note that you may already have MFA enabled on this user, and that's okay—one user can have numerous authentication methods.


My Sign-Ins

Security info

These are the methods you use to sign into your account or reset your password.

Default sign-in method: Microsoft Authenticator - notification Change

Microsoft Authenticator iFun

Lost device? Sign out everywhere

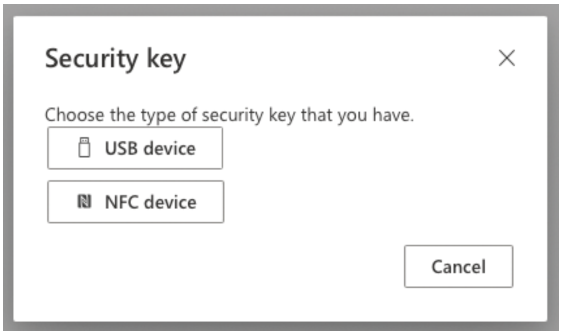
Figure 7: Add an authentication method for the user.

Once signed in, visit the "Security" section and click on **Add method**, as shown in **Figure 7**.

When prompted, choose **Security key** as the authentication method you'd like to use, and click **Add**. You'll then be prompted to pick what kind of device you wish to use. This can be seen in **Figure 8**.

I have a USB-C YubiKey, so I'll pick "USB device". Next, I'm shown a message saying that I should have my key ready, and when prompted, plug in the key and touch the key's sensor or button to finish setting it up. As soon as I click Next, I'm redirected to a new window to finish set up.

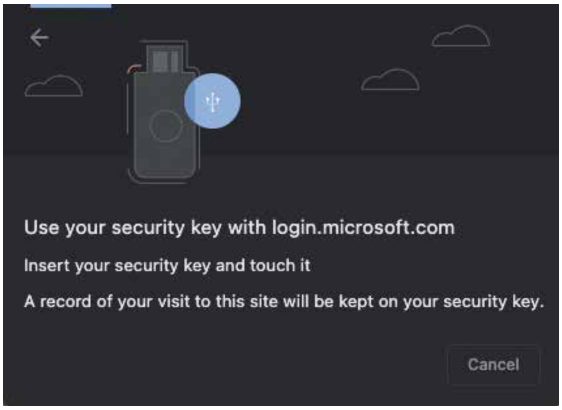
Here, Azure AD shows you a message, but the real authentication dance is built into the browser using the WebAuthn protocol. The browser now prompts you to plug your security key in. This can be seen in **Figure 9**.


Security key

Choose the type of security key that you have.

Figure 8: Security key type

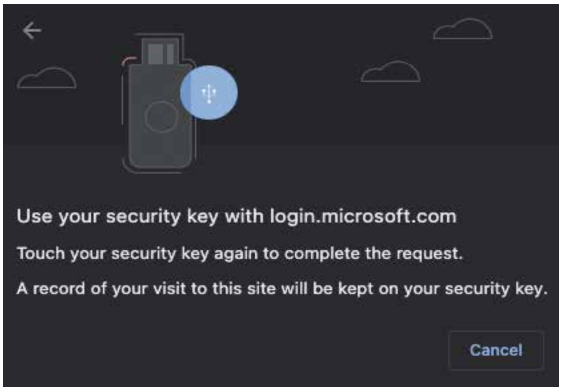
In my case, I use this key for numerous purposes, so it's locked with a PIN. As soon as I plug it in, I'm asked to enter the PIN. Once I do that, the browser then asks me if I wish to use this key with the given website, which is, in this case, login.microsoftonline.com. Although it's not very difficult to build FIDO2 authentication right on your website, most identity providers already support this, and delegating this responsibility to them is usually what we do these days anyway. This can be seen in **Figure 10.**


Use your security key with login.microsoft.com

Insert your security key and touch it

A record of your visit to this site will be kept on your security key.

Figure 9: Chrome prompts you to plug your key in.


Use your security key with login.microsoft.com

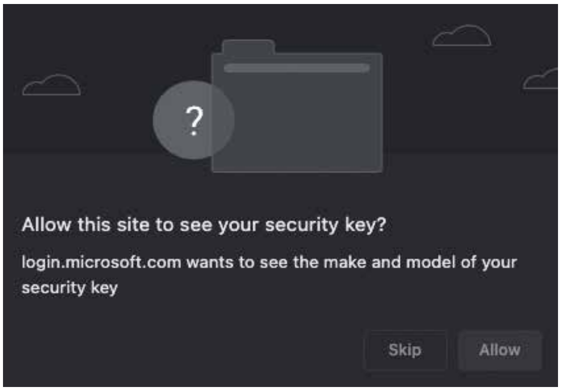
Touch your security key again to complete the request.

A record of your visit to this site will be kept on your security key.

Figure 10: Chrome prompting you if you wish to use your FIDO2 key with AAD

To allow this key to be used with login.microsoftonline.com, you now have to touch the key. This proves physical possession of the key. Remember: The key, if cloned, can easily be detected using an ever-increasing counter.

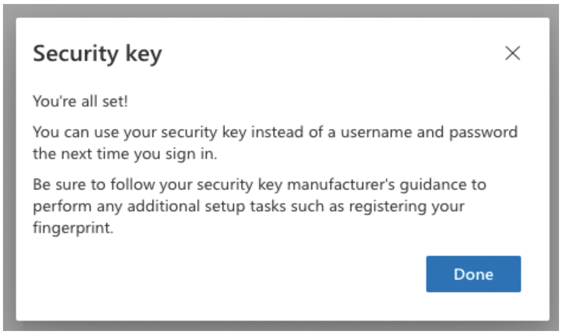
As soon as I touch the key, I'm shown a third prompt, asking me if I allow the site (in this case AAD), to see the details of my key. Say **Allow**. This can be seen in **Figure 11**.


Allow this site to see your security key?

login.microsoft.com wants to see the make and model of your security key

Figure 11: Sending information about the key

What's interesting is that all this was built right into the browser. AAD has been patiently waiting for your key to be registered before moving further. At this point, your key is registered, and AAD asks you to name it. Giving it a meaningful name, I called mine **SahilKey**, and I soon see a message confirming that the key is ready for use. This can be seen in **Figure 12**.


Security key

You're all set!

You can use your security key instead of a username and password the next time you sign in.

Be sure to follow your security key manufacturer's guidance to perform any additional setup tasks such as registering your fingerprint.

Figure 12: FIDO2 key is ready for use with AAD.

Now let's see the sign-in experience.

Go ahead and sign out from myprofile.microsoft.com. You can do so by clicking the person-like icon on the top right-hand corner and choosing **Signout**. Now relaunch the browser, and visit any site protected by the same AAD. I'll just use myprofile.microsoft.com again. Enter your username, (testuser10 in my case), and pick "Sign in with Windows Hello or a security key."

Here's a pet peeve. I'm on a Mac, and this system should be smart enough to not confuse the user with "Windows hello" on a Mac. But I digress.

I do have a security key, so I'll click on that link. Chrome now shows me a bunch of options to sign in using. This can be seen in **Figure 13.**

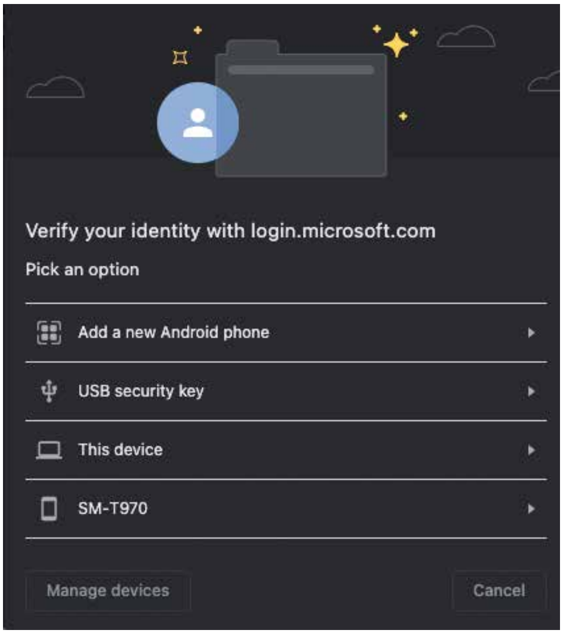
The exact list you see may be different. You may also be prompted for Bluetooth permissions at this point. I intend to use a USB YubiKey, so I pick **USB security key**. Now Chrome takes you through a simple sign-in process that involves touching the key, entering a PIN, and boom, you're signed in.

See how easy that was? Not only that, when I signed in using FIDO2, I didn't have to enter a password or remember a password, and the server's workload is also greatly reduced. Plus, I never sent anything sensitive over the wire. It's a win-win for all.

What if you lose the key? Well, you can always fall back to a back-up authentication method, such as an authenticator app in Azure AD. However, it's also not atypical to register more than one security key.

Summary

Passwords suck and I hate dealing with them. I like MFA, but it's so inconvenient to deal with MFA sometimes. FIDO2 is supported by a number of organizations. And it really simplifies the log-in process while keeping my credentials secure. Some of the places I use FIDO2 already are Facebook, Twitter, GitHub, my Azure and Google accounts, plus a few others. If you wish to see who supports FIDO2, visit www.



Verify your identity with login.microsoft.com

Pick an option

Figure 13: Many ways to sign in.

dongleauth.info. You'd be pleasantly surprised to see how many sites already support FIDO2.

No doubt this identity and security space will continue to evolve, but everyone has, at this point, unanimously agreed to kill passwords. If username password is your line of defense, I have bad news for you.

FIDO2 keys aren't perfect. There's a physical key that you must carry. But with platform authenticators, and technologies such as FaceID that are better at identifying individuals than fingerprints, the keys really are very compelling argument.

The best part is that almost every major identity provider already supports it, and it's not hard to set up. If I use a username password as the only protection on a website, I just assume it to be insecure. I won't use anything useful if it at least doesn't support MFA. But MFA is inconvenient. So if a site gives me the option to use FIDO2 keys, that's my solution.

How about you? Do you have any critical sites that you use just username password on?

Sahil Malik

ONLINE QUICK ID 2209031

YARP: I Did It Again

With developers becoming increasingly comfortable with microservices, reverse proxies have gained visibility. Inside Microsoft, someone noticed that a number of teams were building reverse proxies for their own projects. Luckily, someone realized that a single, reusable reverse proxy would be something that we could all benefit from. This led them to release "Yet Another Reverse Proxy" or YARP. Let's talk about what reverse proxies are and how YARP works.

Shawn Wildermuth
shawn@wildermuth.com
wildermuth.com
twitter.com/shawnwildermut

Shawn Wildermuth has been tinkering with computers and software since he got a Vic-20 back in the early '80s. As a Microsoft MVP since 2003, he's also involved with Microsoft as an ASP.NET Insider and ClientDev Insider. He's the author of over twenty Pluralsight courses, written eight books, an international conference speaker, and one of the Wilder Minds. You can reach him at his blog at http://wildermuth.com. He's also making his first, feature-length documentary about software developers today called "Hello World: The Film." You can see more about it at http://helloworldfilm.com.

This bring us to two important questions: "What is a Reverse Proxy?" and "How do I create a reverse proxy?"

What's a Reverse Proxy?

If you're like me, the word "proxy" is an overloaded term. In different contexts, the word proxy means something different to different people. In this case, I'm talking about a server that's an intermediary between the caller and the receiver of a networking call (usually HTTP or similar). Before you can understand a reverse proxy, let's talk about forward proxies (or proxy servers, as you might be familiar with).

A proxy server is a server that takes requests and re-executes the call to the Internet (or intranet) on behalf of the original caller. This can be used for caching requests to improve speed of execution or for filtering content (as well as other reasons). In **Figure 1**, you can see a typical proxy server diagram.

A reverse proxy is very much like a proxy server, but, not too surprisingly, in reverse. Instead of intercepting calls going outside the Internet/intranet, a reverse proxy intercepts calls from the outside and forwards them to local servers. Often the proxy server is the only accessible server in this scenario. If you look at **Figure 2**, you can see that all calls come into the reverse proxy. Often the caller has no idea that there's a reverse proxy.

Now that you have a general idea of what a reverse proxy is, let's talk about the why of reverse proxies.

Do I Need a Reverse Proxy?

Many projects have no need for a reverse proxy. You should learn about them anyway, because it's another arrow in your development quiver to use when you need it. The use-case for using a reverse proxy is fairly well defined. The reverse proxy can be used in microservice scenarios where you don't want individual clients to know about the naming or topology of your data center.

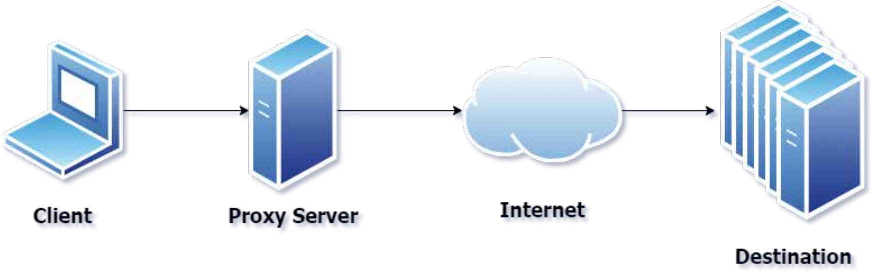


Figure 1: Proxy server

Reverse proxies are not only helpful in those microservices projects. Here are some other reasons to use a reverse proxy:

- Service gatekeeping
- Load balancing
- SSL termination
- Security
- URL writing

Although you might want to use a reverse proxy for all of these reasons, you don't need all of these services. Use a reverse proxy in the way your application works. You can use reverse proxies as a product (e.g., CloudFlare) or built into your own projects.

Let's look at a new support in .NET projects called YARP.

Using YARP

The most obvious use-case for many of you reading this article is to use a reverse proxy to provide an API gateway for microservices. A reverse proxy can expose a server that represents a single surface area for requests. The details of how the service is implemented and where the actual service resides are made opaque to the actual clients. This is what I call **service aggregation**. In this case, a reverse proxy is used to accept calls from clients and then pass them off to the underlying service (or cluster of services). This allows you to change the composition of the microservice without breaking clients.

You can use **service aggregation** to marry disparate systems without having to rewrite or change the underlying technology. For example, you might have a Java system from an acquisition, a .NET project that's built in-house, and a Python machine learning project that you have to integrate. By using a reverse proxy, you can create a union of all these services to provide a single API service area for these different technologies.

Now that you've seen a bit about what a reverse proxy is, let's see how to implement a reverse proxy it in a .NET Core project using the YARP library. To get started, you need any ASP.NET Core project. Let's create an empty project (calling it **DidItAgain.Proxy**):

```
> dotnet new web -n DidItAgain.Proxy
```

To use YARP, you just need to add the NuGet package:

```
> dotnet add package Yarp.ReverseProxy
```

Once installed, you can wire the middleware. First, you need to add the reverse proxy services and configure it:

```
var bldr = WebApplication.CreateBuilder(args);

bldr.Services.AddReverseProxy();
```

As you can see, you first add the proxy service dependencies with **AddReverseProxy**. You need to configure it, but I'll get to that soon. Before you do that, let's add the middleware:

```
var app = bldr.Build();

app.MapReverseProxy();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Configuring the Reverse Proxy

In YARP, the reverse proxy needs to know what the pattern is that you're looking for in requests and where to pass the requests to. It uses the term **Routes** for the request patterns and uses **Clusters** to represent the computers(s) to forward those requests. This means that you need a way of providing the proxy with a set of Routes and Clusters. The most direct is to use a section in your configuration files:

```
var proxy = bldr.Services.AddReverseProxy();
proxy.LoadFromConfig(
  bldr.Configuration.GetSection("Yarp"));
```

By calling the **LoadFromConfig**, the proxy expects a section that conforms to the schema of the proxy configuration. It doesn't matter what you call the section, as long as it's a set of **Routes** and **Clusters**. For example, here's the general structure of the configuration section:

```
{
  ...
  "Yarp": {
    "Routes": {
      ...
    },
    "Clusters": {
      ...
    }
  }
}
```

Let's start with the Cluster:

```
"Clusters": {
  "CustomerCluster": {
    "Destinations": {
      "customerServer": {
        "Address": "https://someurl.com/"
      }
    }
  }
}
```

A **Cluster** (named **CustomerCluster**) is just a destination for an endpoint server(s). Note that there could be multiple destinations and each could use different semantics to determine where to locate an endpoint server and transform it. Requests typically keep their paths and append them to the address. This is typically matched with a **Route**:

```
"Routes": {
  "CustomerRoute": {
    "ClusterId": "CustomerCluster",
    "Match": {
      "Path": "/api/customers/{**catch-all}"
    }
  }
}
```

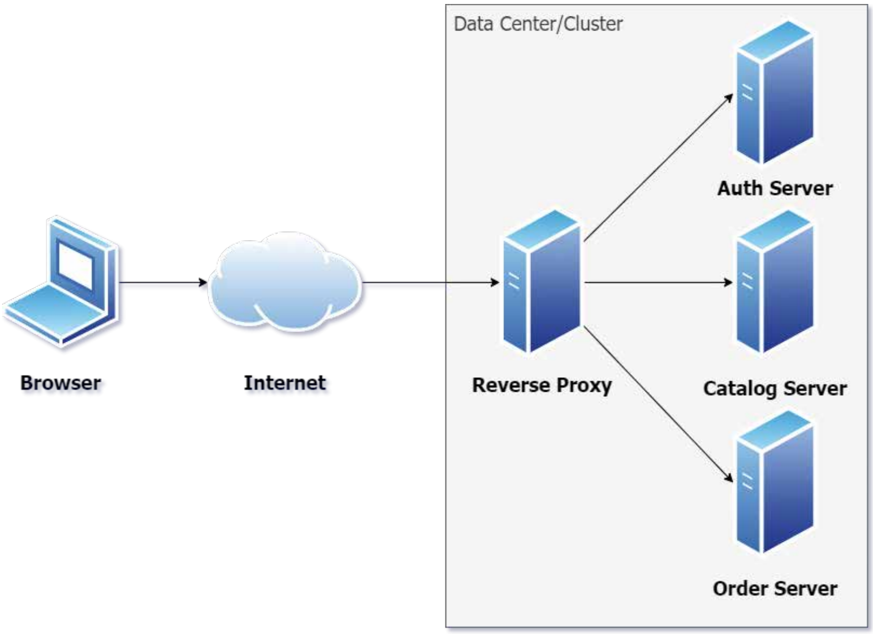


Figure 2: Reverse proxy

A route (named **CustomerRoute** in this example) is a set of rules for matching the request and pointing to a **Cluster** via the **ClusterId**. In this example, the route matches calls to the proxy server that start with **/api/customers/** and directs them to the customer Cluster. Routes can match based on various criteria:

- Path (like you've just seen)
- Headers
- Query string parameters
- HTTP method
- Host name

This gives you a lot of control over how the reverse proxy matches URIs to other computers. Although typically used as a façade to your own servers, it can be used to proxy to wherever you want.

Programmatic Configuration

Although using the configuration file is a common way to configure the proxy server, often you want to have a data-driven approach or integrate the proxy with a service discovery service (e.g., the Microsoft Tye project). To supply the configuration file, you'll need to create a class that implements the **IProxyConfigProvider** interface:

```
public class YarpProxyConfigProvider
  : IProxyConfigProvider
{
  public IProxyConfig GetConfig()
  {
    return new YarpProxyConfig();
  }

}
```

The provider requires you to implement a class that represents the **IProxyConfig** interface. Although this interface is simple, the **IProxyConfig** is where the building up of the configuration happens. For example:

```
public class YarpProxyConfig : IProxyConfig
{
  readonly List<RouteConfig> _routes;
  readonly List<ClusterConfig> _clusters;
  readonly CancellationChangeToken _changeToken;
  readonly CancellationTokenSource _cts =
    new CancellationTokenSource();

  public YarpProxyConfig()
  {
    _routes = GenerateRoutes();
    _clusters = GenerateClusters();
    _cts = new CancellationTokenSource()
    _changeToken = new
      CancellationChangeToken(_cts.Token);
  }

  public IReadOnlyList<RouteConfig> Routes
    => _routes;
  public IReadOnlyList<ClusterConfig> Clusters
    => _clusters;
  public IChangeToken ChangeToken =>
    _changeToken;
}
```

You can see that the interface has three members. The **Routes** and **Clusters** return a list of the Route and Clusters (with the same structure you see in the config file above). The **ChangeToken** is used to notify the system of changes to the configuration, if needed. Creation of the clusters looks like you'd expect:

```
private List<ClusterConfig> GenerateClusters()
{
  var collection = new List<ClusterConfig>();
  collection.Add(new ClusterConfig()
  {
    ClusterId = "FirstCluster",
    Destinations =
      new Dictionary<string, DestinationConfig>{
        {
          "server",
          new DestinationConfig()
          {
            Address = "https://someserver.com"
          }
        }
      }
  });
  return collection;
}
```

Although I'm hard-coding the configuration (which is really not any better than configuration files), you could use code to determine how the clusters should be configured. It's similar to create routes:

```
private List<RouteConfig> GenerateRoutes()
{
  var collection = new List<RouteConfig>();
  collection.Add(new RouteConfig()
  {
    ClusterId = "FirstCluster",
    Match = new RouteMatch()
    {
      Path = "/api/foo/{**catch-all}"
    }
  });

  return collection;
}
```

Again, this should look a lot like the configuration file example. There's a difference in how you wire-up the services for the reverse proxy:

```
using DidItAgain.Proxy;
using Yarp.ReverseProxy.Configuration;

var bldr = WebApplication.CreateBuilder(args);

bldr.Services.AddTransient<IProxyConfigProvider,
  YarpProxyConfigProvider>();

bldr.Services.AddReverseProxy();

var app = bldr.Build();
```

Notice that you're adding your provider into the services collection and adding the reverse proxy. When it's constructed, it queries for the proxy config provider on its own and finds yours.

Now that you've seen how to configure it, let's talk about how to configure the proxy for different features. From now on, I go back to the configuration file because it's easier to show you how the Clusters and Routes are defined.

Load Balancing

An important use of reverse proxies is to provide generalized load balancing. Again, this allows the reverse proxy to forward requests to more than one server that supplies a specific service. Now you can scale out transparently to the clients of your service(s). Although load balancing is available as a service in many cloud-deployed solutions, in some cases, you'd want more control over it (or you'd use the load balancing support indirectly).

When I say load balancing, I don't mean just sharing load between servers. There are different strategies to load balancing. For example, **Figure 3** shows a typical round-robin load balancing where calls are passed off to different servers in a linear fashion.

There are more strategies for load balancing, but this is probably the most common scenario.

To implement load balancing, you need to specify the load balancing type in the cluster:

```
"CustomerCluster": {
  "Destinations": {
    "customerServer1": { ... },
    "customerServer2": { ... }
  },
  "LoadBalancingPolicy": "RoundRobin"
}
```

The supported policies are:

- **PowerOfTwoChoices (default):** Picks two random destinations and picks the one with the least number of requests.
- **FirstAlphabetical:** Picks the next destination based on name (useful for failover instead of sharing load).
- **Random:** Picks a random server without regard for load.
- **RoundRobin:** Picks a server by going in order without regard for load.
- **LeastRequests:** Picks a server based on the smallest number of requests, but does require that it scan through each destination. This is the slowest but has the highest likelihood of dealing with overloaded servers.

Although load balancing can help you achieve scalability, it doesn't do this by knowing about your servers. If you're completely stateless in those servers, just using the load balancing policy is all you need. But sometimes you have state (e.g., server state or session state) on the servers and need to lock a client to a server once it's been picked. To do this, you can enable **SessionAffinity**:

```
"CustomerCluster": {
  "Destinations": {
    "customerServer1": { ... },
    "customerServer2": { ... }
  },
  "LoadBalancingPolicy": "RoundRobin",
  "SessionAffinity": {
    "Enabled": true
  }
}
```

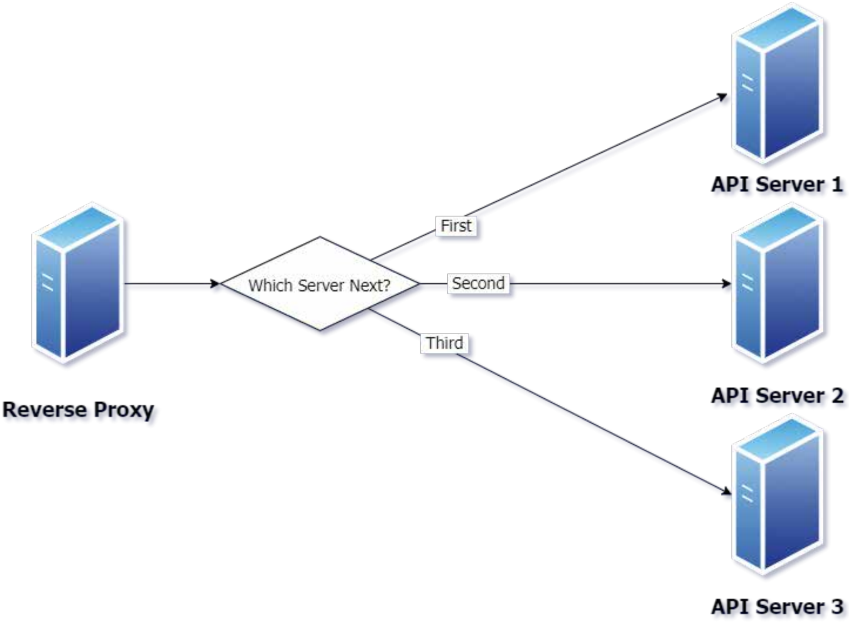


Figure 3: Round Robin Load Balancing

This tracks affinity with a cookie, although you can change the behavior to use a header instead, as well as adding other parameters. By using these two options of the cluster, you can control the behavior of load balancing in the reverse proxy.

To enable load balancing or session affinity, you'll need to opt in during the mapping of the proxy server:

```
app.MapReverseProxy(opt => {
  opt.UseLoadBalancing();
  opt.UseSessionAffinity();
});
```

With this, you can add only the features you want to use.

SSL Termination

Most of the websites that you visit use SSL now to ensure end-to-end encryption of any data. This is a good thing. A reverse proxy has this option to do something called SSL Termination. This is just a fancy name for not using SSL inside a data center. As you can see in **Figure 4**, the SSL call terminates with the proxy server.

SSL Termination allows you to decide whether you need encryption to call the proxied servers. Often, within a data center (or cluster), requests are forwarded without SSL so that you can avoid having to manage certificates for each server cluster. Whether you use SSL is just a matter of what the cluster destination URL is:

```
"Clusters": {
  "CustomerCluster": {
    "Destinations": {
      "customerServer": {
```

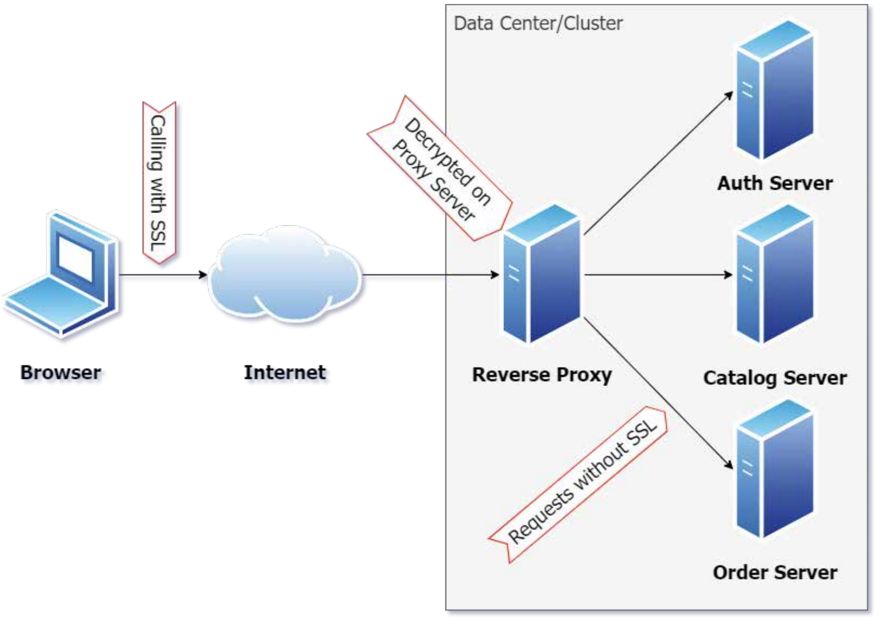


Figure 4: SSL Termination

Source Code

The source code can be downloaded at https://github.com/wilder-minds/yarp-code-magazine

```
        // http for no SSL or https for SSL
        "Address": "http://someurl.com/"
      }
    }
  }
}
```

Security

In most cases, you don't have to do anything special to enable security through the proxy server. If **UseAuthentication** and/or **UseAuthorization** are enabled, the proxy server forwards most credentials to the endpoint servers. Let's look at different types of authentication types:

- **Cookies, Bearer Tokens, API Keys:** As they are part of the request, they'll be forwarded.
- **OAuth2, OpenIdConnect, WsFederation:** As long as they are configured as cookies, they flow through to the endpoint servers.
- **Windows, Negotiate, NTLM, Kerberos:** These authentication schemes are network connection based. Because the connection is through the reverse proxy, they aren't supported in YARP's reverse proxy.

In most cases, authentication flows through to the endpoint servers. I'd be consistent with testing your authentication schemes, though.

URL Rewriting

In some cases, you may want to change the URL before it's sent to the endpoint server. The reasons for this vary, but one common case is to allow for a change to the API without having to change the endpoint API server's syntax. To do this, you'll want to introduce transforms into the configuration. Transforms are added to the Routes so that it is transformed before passing it to a Cluster. For example, if you need to change the URL path, you can do it with a transform:

```
"Routes": {
  "CustomerRoute": {
    "ClusterId": "CustomerCluster",
    "Match": {
      "Path": "/api/customers/{**catch-all}"
    },
    "Transforms": {
      "PathPattern":
        "/api/v2/customers/{**remainder}"
    }
  }
},
```

In this case, it replaces the path with a new URL and anything in the catch-all is added as the suffix. In this example, the transform could be used to redirect to a versioned API. The types of transforms include:

- **Path prefix:** Supports removing or adding a prefix to the request path.
- **Path set:** Replaces a path with a static path.
- **Path pattern:** Like the example, allows you to use pattern matching to recreate the endpoint URL.
- **Query strings:** Add, remove, or convert query strings to other parts of the request (path, query string, or header).
- **HTTP method**: Allows you to change the HTTP method before it's sent to the endpoint server.
- **Headers:** Allows you to have complex changes to headers that are added/removed before a request is sent to the endpoint server.

With the transformation support, you can really control how the requests are formatted when you're forwarding the request to the endpoint server.

Where Are We?

I hope, at this point, that you've seen the benefit of using a proxy server and, by extension YARP. This utility server can be plugged into your architectures to solve a series of different problems. I hope you find that YARP is easy to add to a server and easy to configure.

Shawn Wildermuth

ONLINE QUICK ID 2209041

Simplifying ADO.NET Code in .NET 6: Part 2

In the last article (Simplifying ADO.NET Code in .NET 6: Part 1), you wrote code to simplify ADO.NET and map columns to properties in a class just like ORMs such as the Entity Framework do. You learned to use reflection to make creating a collection of entity objects from a data reader and take advantage of attributes such as [Column] and [NotMapped]. In this article, you're going to refactor the code further to make it even more generic. In addition, you'll learn to get data from a view, get a scalar value, handle multiple result sets, and call stored procedures.

Paul D. Sheriff
http://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.

Refactor the Code for Reusability

In the last article (CODE Magazine, July/August 2022), you added methods to the ProductRepository class to read product data from the SalesLT.Product table in the AdventureWorksLT database. If you look at this code, all of it is completely generic and can be used for any table. As such, this code should be moved to a base class from which you can inherit. You can then have a ProductRepository, CustomerRepository, EmployeeRepository, and other classes that can all inherit from the base class yet add functionality that's specific for each table.

Create a Repository Base Class

Right mouse-click on the **Common** folder and create a new class named **RepositoryBase** and add the code shown **Listing 1**. Notice that the properties are the same as what you previously added to the ProductRepository class. The constructor for this class must be passed the generic DatabaseContext class. After setting the **DbContext** property, the Init() method is called to initialize all the properties to a valid start state.

Add Search() Method Just for Products

Add a Search() method to the RepositoryBase class just below the Init() method. This method is different from the Search() method previously written in the ProductRepository class because it removes the **using** around the SqlServerDatabaseContext.

```
public virtual List<TEntity>
C  Search<TEntity>() {
  List<TEntity> ret;

  // Build SQL from Entity class
  SQL = BuildSelectSql<TEntity>();
  // Create Command Object with SQL
  DbContext.CreateCommand(SQL);
  // Get the list of entity objects
  ret = BuildEntityList<TEntity>
    (DbContext.CreateDataReader());

  return ret;
}
```

You now need to move the BuildEntityList(), BuildCollumnCollection() and the BuildSelectSql() methods from the ProductRepository class into this new RepositoryBase class.

Simplify the Product Repository Class

Now that you have a RepositoryBase class with all of the methods moved from the ProductRepository class, you can greatly simplify the ProductRepository class by having it inherit from the RepositoryBase class. Modify the **ProductRepository.cs** file to look like **Listing 2**.

In the ProductRepository class you must accept a database context object in the constructor because without one, there's no way you could interact with the Product table. A specific Search() method is created to return a list of Product objects in the ProductRepository class, but it simply uses the generic Search<TEntity>() method from the RepositoryBase class.

Add Database Context Class for the AdventureWorksLT Database

Instead of using the generic DatabaseContext or SqlServerDatabaseContext classes directly, it's a better practice to create a database context class for each database you wish to interact with. Right mouse-click on the project and add a new folder named **Models**. Right mouse-click on the Models folder and add a new class named **AdvWorksDbContext** that inherits from the SqlServerDatabaseContext class, as shown in **Listing 3**.

The AdvWorksDbContext class inherits from the SqlServerDatabaseContext because the AdventureWorksLT database you are interacting with is in a SQL Server. An instance of the ProductRepository class is created in the Init() method and exposed as a public property named **Products**. The AdvWorksDbContext is passed to the constructor of the ProductRepository class because it needs the services of a database context to perform its functions against the Product table.

Try It Out

Now that you have made these changes, let's ensure that you can still retrieve all records from the Product table. Open the **Program.cs** file and add a new using statement at the top of the file.

```
using AdoNetWrapperSamples.Models;
```

Remember that you removed the **using** from the Search() method in the RepositoryBase class? You're now going to create the **using** wrapper around the AdvWorksDbContext class to have all objects disposed of properly once you've retrieved all records.

Remove all the lines of code from where you create the ProductRepository class and the call to the Search() method. Add in the code shown in the snippet below. You can now see the **using** statement that wraps up the instance of the AdvWorksDbContext class. This code should look familiar if you have used the Entity Framework (EF), as this is typically how you interact with DbContext classes you create with EF.

```
using AdvWorksDbContext db = new(ConnectString);

List<Product> list = db.Products.Search();

Console.WriteLine("*** Get Product Data ***");
// Display Data
foreach (var item in list) {
  Console.WriteLine(item.ToString());
}
Console.WriteLine();
Console.WriteLine(
  $"Total Items: {list.Count}");
Console.WriteLine();
Console.WriteLine(
  $"SQL Submitted: {db.Products.SQL}");
Console.WriteLine();
```

Run the console application and you should see the complete list of product objects displayed. In addition, you should see the SQL statement submitted by the classes you created in this article.

Searching for Data

In addition to retrieving all records, you probably want to add a WHERE clause to filter the records based on some condition. For example, you might wish to locate all Product records where the **Name** column starts with a specific character and the **ListPrice** column contains a value greater than a specific value. You want to have the wrapper classes generate a SQL statement that looks like the following.

```
SELECT * FROM SalesLT.Product
WHERE Name LIKE @Name + '%'
  AND ListPrice >= @ListPrice
```

You need to add some new functionality to create this SQL statement. You need to pass in values to fill into the **@Name** and **@ListPrice** parameters. You also need to specify what the operators (=, LIKE, or >=) are for each expression. For example, you need to put a LIKE operator for the **@Name** parameter and a greater-than or equal-to (>=) operator for the **@ListPrice** parameter.

Add a Product Search Class

To pass in the values to the Search() method, create a class to hold the parameters you wish to use for the WHERE clause. Right mouse-click on the project and add a new folder named **SearchClasses**. Right mouse-click on the SearchClasses folder and add a new class named **ProductSearch** that looks like the code below.

```
#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.SearchClasses;

public class ProductSearch {
  [Search("LIKE")]
  public string Name { get; set; }
```

Listing 1: Add a base class for all the code that does not change between all repository classes

```
#nullable disable

using System.ComponentModel.DataAnnotations
  .Schema;
using System.Data;
using System.Reflection;
using System.Text;

namespace AdoNetWrapper.Common;

public class RepositoryBase {
  public RepositoryBase(
    DatabaseContext context) {
    DbContext = context;
    Init();
  }

  protected readonly DatabaseContext DbContext;

  public string SchemaName { get; set; }
  public string TableName { get; set; }
  public string SQL { get; set; }
  public List<ColumnMapper> Columns { get; set; }

  protected virtual void Init() {
    SchemaName = "dbo";
    TableName = string.Empty;
    SQL = string.Empty;
    Columns = new();
  }
}
```

Listing 2: Modify the ProductRepository class to pass a Product object to the Search() method

```
#nullable disable

using AdoNetWrapper.Common;
using AdoNetWrapperSamples.Models;
using AdoNetWrapperSamples.EntityClasses;

namespace AdoNetWrapperSamples
  .RepositoryClasses;

public class ProductRepository
  : RepositoryBase {
  public ProductRepository(
    AdvWorksDbContext context)
      : base(context) { }

  public virtual List<Product> Search() {
    return base.Search<Product>();
  }
}
```

Listing 3: Create a DbContext class for each database you wish to interact with

```
#nullable disable

using AdoNetWrapper.Common;
using AdoNetWrapperSamples
  .RepositoryClasses;

namespace AdoNetWrapperSamples.Models;

public partial class AdvWorksDbContext
  : SqlServerDatabaseContext {
  public AdvWorksDbContext(string connectString)
   : base(connectString) { }

  protected override void Init() {
    base.Init();

    Products = new(this);
  }

  public ProductRepository Products { get; set; }
}
```

```
  [Search(">=")]
  public decimal? ListPrice { get; set; }
}
```

Create the **Name** and **ListPrice** properties to use for searching. All properties in this class should be nullable unless you wish to require the user to enter at least one search value prior to searching for records. All properties should be decorated with the [Search] attribute unless you just wish to use an equal (=) operator in the WHERE clause.

Add a Search Attribute Class

Microsoft doesn't have a [Search] attribute, so it's up to you to create one. Right mouse-click on the **Common** folder and add a new class named **SearchAttribute**, as shown in the following code snippet.

```
#nullable disable

namespace AdoNetWrapper.Common;

[AttributeUsage(AttributeTargets.Property)]
public class SearchAttribute : Attribute {
  public string SearchOperator { get; set; }
  public string ColumnName { get; set; }

  public SearchAttribute(string searchOperator) {
    SearchOperator = searchOperator ?? "=";
  }
}
```

There are two properties needed for this attribute class, **SearchOperator** and **ColumnName**. The **SearchOperator** property is assigned to an equal sign (=) if one isn't supplied. If the **ColumnName** property is a null, the code you're going to use to create the WHERE clause will use the property name of the search class.

Modify the ColumnWrapper Class

When building the collection of columns needed for the WHERE clause, the process is going to be like the code used to build the columns for the SELECT statement. However, you're going to need two additional items to keep track of: the value to supply as a parameter and for the search operator to use. Open the **ColumnMapper.cs** file in the Common folder and add a **ParameterValue** property and a **SearchOperator** property.

```
public class ColumnMapper {
  public string ColumnName { get; set; }
  public PropertyInfo PropertyInfo { get; set; }
  public object ParameterValue { get; set; }
  public string SearchOperator { get; set; }
}
```

Add Method to Build Search Column Collection

Open the **RepositoryBase.cs** file and add a new method named BuildSearchColumnCollection(), as shown in **Listing 4**. This method is just like the BuildColumnCollection() method you wrote in the last article. Create an array of PropertyInfo objects for each property in the TSearch class. Loop through the array of properties and retrieve the value for the current property of the search class. If the value is filled in, create a new ColumnMapper object. Check for a [Search] attribute and if found, see if the **ColumnName** and/or the **SearchOperator** property exists. Override those properties in the ColumnWrapper object if they do exist. Add the new ColumnWrapper object into the **ret** variable to be returned once all properties in the search class are processed.

Add Method to Create WHERE Clause for Searching

The next new method is used to build the actual WHERE clause to be added to the SELECT statement. Add a new method named **BuildSearchWhereClause()**, as shown in **Listing 5**. Pass to this method the list of ColumnWrapper objects created using the BuildSearchColumnCollection() method. Iterate over the list of objects and build the WHERE clause. Be careful when copying the code from the article as I had to break lines in the sb.Append() due to formatting of the article. The interpolated string belongs all on one line with a space between each item except between the **ParameterPrefix** and the **ColumnName** properties.

Add Method to Create Parameters for Command Object

The last new method to build is called BuildWhereClauseParameters(), as shown in **Listing 6**. In this method, you iterate over the same collection of ColumnMapper objects you created in the BuildSearchColumnCollection() method.

Listing 4: Create method to build collection of properties for the search columns

```
protected virtual List<ColumnMapper>
  BuildSearchColumnCollection<TEntity,
    TSearch>(TSearch search) {
  List<ColumnMapper> ret = new();
  ColumnMapper colMap;
  object value;

  // Get all the properties in <TSearch>
  PropertyInfo[] props =
    typeof(TSearch).GetProperties();

  // Loop through all properties
  foreach (PropertyInfo prop in props) {
    value = prop.GetValue(search, null);

    // Is the search property filled in?
    if (value != null) {
      // Create a column mapping object
      colMap = new() {
        ColumnName = prop.Name,
        PropertyInfo = prop,
        SearchOperator = "=",
        ParameterValue = value
      };

      // Does Property have a [Search] attribute
      SearchAttribute sa = prop
       .GetCustomAttribute<SearchAttribute>();
      if (sa != null) {
        // Set column name from [Search]
        colMap.ColumnName =
          string.IsNullOrWhiteSpace(sa.ColumnName)
            ? colMap.ColumnName : sa.ColumnName;
        colMap.SearchOperator =
          sa.SearchOperator ?? "=";
      }

      // Create collection of columns
      ret.Add(colMap);
    }
  }

  return ret;
}
```

Each time through, build a new SqlParameter object passing in the column name and either the value to submit by itself, or if the **SearchOperator** property is equal to "LIKE", you use the value and add on a percent sign (%).

Overload Search() Method to Accept a Command Object

Add a new overload for the Search() method to accept a Command object (**Listing 7**). This Search() method checks to ensure that the Columns collection has been built from the TEntity class. It then sets the **DbContext.CommandObject** property to the **cmd** object variable passed in. The BuildEntityList() method is then called to create the list of entity objects.

Modify the original Search() method to call the new overload you just created, as shown in the following code snippet.

```
public virtual List<TEntity> Search<TEntity>() {
  // Build SQL from Entity class
  SQL = BuildSelectSql<TEntity>();

  // Create Command Object with SQL
  DbContext.CreateCommand(SQL);

  return Search<TEntity>(DbContext.CommandObject);
}
```

Overload Search() Method to Accept Search Class

Open the **RepositoryBase.cs** file and add another overloaded Search() method that takes two type parameters TEntity and TSearch, as shown in **Listing 8**. After building the SELECT statement, call the BuildSearchColumnCollection() method that uses the TSearch class to build a collection of columns to be used in the WHERE clause. If there are any search columns, call the BuildSearchWhereClause() to build the actual WHERE clause to add to the SELECT statement. The SqlCommand object is built using the new SELECT clause, and then parameters are added with the values from the TSearch object. The SqlCommand object is then passed to the Search() method that accepts the command object.

Modify Product Repository Class

Now that you have the generic version of the Search() method to accept a search entity object, you need to add a Search() method to the ProductRespository class to accept a ProductSearch class. Open the **ProductRepository.cs** file and add a new using statement at the top of the file.

```
using AdoNetWrapperSamples.SearchClasses;
```

Add a new Search() method to the ProductRepository class to call the Search<TEntity, TSearch>() method in the RepositoryBase class.

```
public virtual List<Product>
  Search(ProductSearch search) {
  return base
    .Search<Product, ProductSearch>(search);
}
```

Try It Out

Open the **Program.cs** file and add a new **using** statement at the top of the file so you can use the ProductSearch class.

```
using AdoNetWrapperSamples.SearchClasses;
```

Listing 5: Add a method to build a WHERE clause for searching

```
protected virtual string BuildSearchWhereClause
  (List<ColumnMapper> columns) {

  StringBuilder sb = new(1024);
  string and = string.Empty;

  // Create WHERE clause
  sb.Append(" WHERE");
  foreach (var item in columns) {
    sb.Append($"{and} {item.ColumnName}
     {item.SearchOperator}
     {DbContext.ParameterPrefix}
     {item.ColumnName}");
    and = " AND";
  }

  return sb.ToString();
}
```

Listing 6: Add a method to build the parameters for the WHERE clause

```
protected virtual void BuildWhereClauseParameters
  (IDbCommand cmd,
   List<ColumnMapper> whereColumns) {

  // Add parameters for each value passed in
  foreach (ColumnMapper item in whereColumns) {
    var param = DbContext.CreateParameter(
      item.ColumnName,
      item.SearchOperator == "LIKE" ?
        item.ParameterValue + "%" :
          item.ParameterValue);
    cmd.Parameters.Add(param);

    // Store parameter info
    Columns.Find(c => c.ColumnName ==
                      item.ColumnName)
      .ParameterValue = item.ParameterValue;
  }
}
```

Listing 7: Add a Search() method that accepts a Command object

```
public virtual List<TEntity>
  Search<TEntity>(IDbCommand cmd) {
  List<TEntity> ret;

  // Build Columns if needed
  if (Columns.Count == 0) {
    Columns = BuildColumnCollection<TEntity>();
  }

  // Set Command Object
  DbContext.CommandObject = cmd;

  // Get the list of entity objects
  ret = BuildEntityList<TEntity>
    (DbContext.CreateDataReader());

  return ret;
}
```

Add an instance of the ProductSearch class and initialize the **Name** property to the value "C", and the **ListPrice** property to be 50. Call the overloaded Search() method you just added to the ProductRepository class and pass in the instance of the ProductSearch class as shown in the following code.

```
using (AdvWorksDbContext db = new(ConnectString));

ProductSearch search = new() {
  Name = "C",
  ListPrice = 50
};

List<Product> list =
  db.Products.Search(search);

// REST OF THE CODE HERE
```

Run the console application and you should see three products displayed, as shown in **Figure 1**.

Create Generic Method to Submit SQL

Sometimes you may need a way to submit any SQL statement to the database and have it return any list of objects you want. Maybe you want to submit some SQL that has a few tables joined together. Into which repository class would you want to put that? Instead of worrying about where it belongs, you can create a **Database** property on the AdvWorksDbContext class that's of the type RepositoryBase and just submit the SQL using a SqlCommand object. Open the **AdvWorksDbContext.cs** file and add a new property of the type RepositoryBase.

```
public RepositoryBase Database { get; set; }
```

Modify the constructor of the AdvWorksDbContext class to pass in the current instance of AdvWorksDbContext to the RepositoryBase class instance called **Database**.

```
public virtual void Init() {
  Database = new(this);
  Products = new(this);
}
```

Building Your Own Command Object

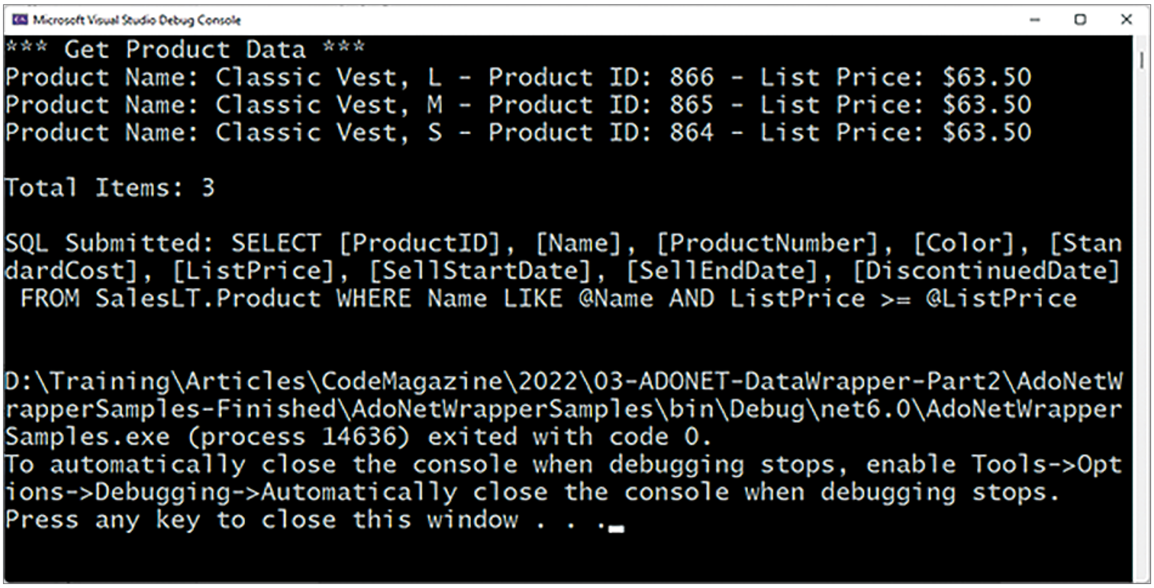
Open the **Program.cs** file and create a SQL string with the same WHERE clause you created earlier (**Listing 9**). Create a SqlCommand object by calling the CreateCommand() method and pass in the **sql** variable. Add the parameters to the command object and pass in some hard-coded values. Call the Search<Product>(cmd) method directly to retrieve the list of rows in the Product table that match the search criteria.

Try It Out

Run the console application and you should see three products displayed, as shown in **Figure 2**.

Retrieve Data from a View

Now let's retrieve the data from a view in the AdventureWorksLT database named vProductAndDescription. If this

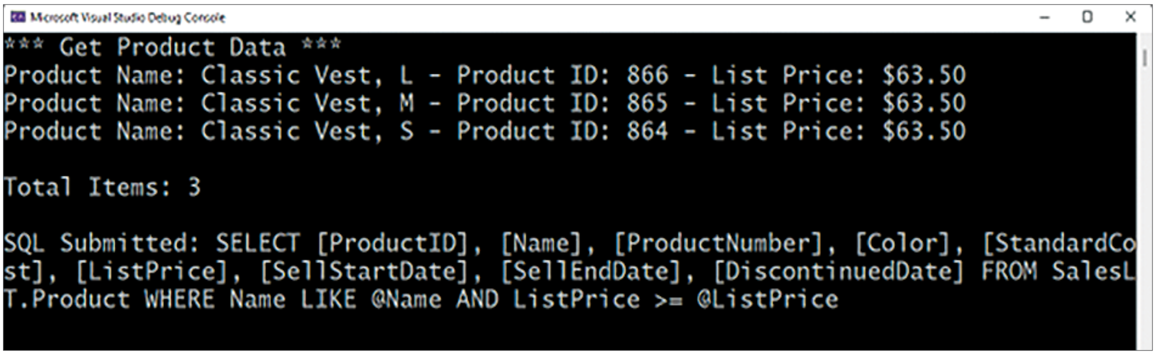


```
*** Get Product Data ***
Product Name: Classic Vest, L - Product ID: 866 - List Price: $63.50
Product Name: Classic Vest, M - Product ID: 865 - List Price: $63.50
Product Name: Classic Vest, S - Product ID: 864 - List Price: $63.50

Total Items: 3

SQL Submitted: SELECT [ProductID], [Name], [ProductNumber], [Color], [Stan
dardCost], [ListPrice], [SellStartDate], [SellEndDate], [DiscontinuedDate]
 FROM SalesLT.Product WHERE Name LIKE @Name AND ListPrice >= @ListPrice


D:\Training\Articles\CodeMagazine\2022\03-ADONET-DataWrapper-Part2\AdoNetW
rapperSamples-Finished\AdoNetWrapperSamples\bin\Debug\net6.0\AdoNetWrapper
Samples.exe (process 14636) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Opt
ions->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 1: Build a WHERE clause to limit the total records returned

Listing 8: Create an overloaded Search() method to accept a Product Search class

```
public virtual List<TEntity>
  Search<TEntity, TSearch>(TSearch search) {
  // Build SQL from Entity class
  SQL = BuildSelectSql<TEntity>();

  // Build collection of ColumnMapper objects
  // from properties in the TSearch object
  var searchColumns =
    BuildSearchColumnCollection<TEntity,
      TSearch>(search);

  if (searchColumns != null &&
      searchColumns.Any()) {
    // Build the WHERE clause for Searching
    SQL += BuildSearchWhereClause(searchColumns);
  }

  // Create Command Object with SQL
  DbContext.CreateCommand(SQL);

  // Add any Parameters?
  if (searchColumns != null &&
      searchColumns.Any()) {
    BuildWhereClauseParameters(
      DbContext.CommandObject, searchColumns);
  }

  return Search<TEntity>(DbContext.CommandObject);
}
```



```
*** Get Product Data ***
Product Name: Classic Vest, L - Product ID: 866 - List Price: $63.50
Product Name: Classic Vest, M - Product ID: 865 - List Price: $63.50
Product Name: Classic Vest, S - Product ID: 864 - List Price: $63.50

Total Items: 3

SQL Submitted: SELECT [ProductID], [Name], [ProductNumber], [Color], [StandardCo
st], [ListPrice], [SellStartDate], [SellEndDate], [DiscontinuedDate] FROM SalesL
T.Product WHERE Name LIKE @Name AND ListPrice >= @ListPrice
```

Figure 2: Add a WHERE clause to your SQL by using a search class and the [Search] attribute.

Listing 9: Create a SQL statement and a Command object to submit a search

```
using AdvWorksDbContext db = new(ConnectString);

string sql = "SELECT * FROM SalesLT.Product ";
sql += "WHERE Name LIKE @Name + '%'";
sql += " AND ListPrice >= @ListPrice";

// Create Command object
var cmd = db.CreateCommand(sql);
// Add Parameters
cmd.Parameters.Add(
  db.CreateParameter("Name", "C"));
cmd.Parameters.Add(
  db.CreateParameter("ListPrice", 50));
// Call the SELECT statement
List<Product> list =
  db.Database.Search<Product>(cmd);

Console.WriteLine("*** Get Product Data ***");
// Display Data
foreach (var item in list) {
  Console.WriteLine(item.ToString());
}
Console.WriteLine();
Console.WriteLine($"Total Items:
  {list.Count}");
Console.WriteLine();
```

view isn't already in the AdventureWorksLT database, create it using the following SQL:

```
CREATE VIEW vProductAndDescription AS
SELECT p.ProductID, p.Name,
       pm.Name AS ProductModel,
       pmd.Culture, pd.Description
FROM SalesLT.Product AS p
  INNER JOIN SalesLT.ProductModel AS pm
    ON p.ProductModelID = pm.ProductModelID
  INNER JOIN
    SalesLT.ProductModelProductDescription AS pmd
    ON pm.ProductModelID = pmd.ProductModelID
  INNER JOIN
    SalesLT.ProductDescription AS pd
    ON pmd.ProductDescriptionID =
       pd.ProductDescriptionID;
```

Add a new class named ProductAndDescription to map to the vProductAndDescription view. Right mouse-click on the EntityClasses folder and add a new class named **ProductAndDescription**, as shown in **Listing 10**.

Listing 10: Add an Entity class to map the results returned from the view

```
#nullable disable

using System.ComponentModel
  .DataAnnotations.Schema;

namespace AdoNetWrapperSamples.EntityClasses;

[Table("vProductAndDescription",
  Schema = "SalesLT")]
public partial class ProductAndDescription {
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string ProductModel { get; set; }
  public string Culture { get; set; }
  public string Description { get; set; }

  public override string ToString() {
    return $"Name={Name} -
             ProductModel={ProductModel} -
             Description={Description}";
  }
}
```

Try It Out

Open the **Program.cs** file and modify the code to call the view using the Search() method on the Database property.

```
using AdvWorksDbContext db = new(ConnectString);

// Get all rows from view
List<ProductAndDescription> list =
  db.Database.Search<ProductAndDescription>();

Console.WriteLine("*** Get Product Data ***");
// Display Data
```

```
foreach (var item in list) {
  Console.WriteLine(item.ToString());
}
Console.WriteLine();
Console.WriteLine(
  $"Total Items: {list.Count}");
Console.WriteLine();
Console.WriteLine(
  $"SQL Submitted: {db.Database.SQL}");
```

Run the console application and you should see over 1700 rows appear from the view. Many of these have a bunch of questions marks. This is because the data in the table has some foreign language characters.

Listing 11: The Find() method retrieves a single entity from the table

```
public virtual TEntity Find<TEntity>
  (params Object[] keyValues)
    where TEntity : class {
  // To assign null, use 'where TEntity : class'
  TEntity ret = null;

  if (keyValues != null) {
    List<ColumnMapper> searchColumns;

    // Build SQL from Entity class
    SQL = BuildSelectSql<TEntity>();

    // Build a collection of ColumnMapper
    // objects based on [Key] attribute
    searchColumns = Columns
      .Where(col => col.IsKeyField).ToList();

    // Number of [Key] attributes on entity class
    // must match number of key values passed in
    if (searchColumns.Count != keyValues.Length) {
      throw new ApplicationException(
        "Not enough parameters passed to Find()
         method, or not enough [Key] attributes
         on the entity class.");
    }

    // Set the values into the searchColumns
    for (int i = 0; i < searchColumns.Count;
         i++) {
      searchColumns[i].ParameterValue =
        keyValues[i];
      searchColumns[i].SearchOperator = "=";
    }

    // Build the WHERE clause for Searching
    SQL += BuildSearchWhereClause(searchColumns);

    // Create command object with SQL
    DbContext.CreateCommand(SQL);

    // Add any Parameters?
    if (searchColumns != null &&
        searchColumns.Any()) {
      BuildWhereClauseParameters(
        DbContext.CommandObject, searchColumns);
    }

    // Get the entity
    ret = Find<TEntity>(DbContext.CommandObject);
  }

  return ret;
}
```

Search Using a View

Just like you created a search class for the Product table, you can also create a search class for searching when using a view. Right mouse-click on the SearchClasses folder and add a new class named **ProductAndDescriptionSearch**, as shown in the code snippet below.

```
#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.SearchClasses;

public class ProductAndDescriptionSearch {
  [Search("=")]
  public string Culture { get; set; }
}
```

Listing 12: The overload of the Find() method executes the command

```
public virtual TEntity
  Find<TEntity>(IDbCommand cmd)
  where TEntity : class {
  // To assign null, use 'where TEntity : class'
  TEntity ret = null;

  // Build Columns if needed
  if (Columns.Count == 0) {
    Columns = BuildColumnCollection<TEntity>();
  }

  // Get the entity
  var list = Search<TEntity>(cmd);

  // Check for a single record
  if (list != null && list.Any()) {
    // Assign the object to the return value
    ret = list[0];
  }

  return ret;
}
```

Try It Out

Modify the code in **Program.c**s to create an instance of this new search class. Set the **Culture** property to the value "en" so you only grab those records where the Culture field matches this value. Call the overload of the Search() method to which you pass a search class.

```
ProductAndDescriptionSearch search = new() {
  Culture = "en",
};

// Perform a search for specific culture
List<ProductAndDescription> list =
  db.Database.Search<ProductAndDescription,
    ProductAndDescriptionSearch>(search);
```

Run the console application and you should see almost 300 rows of data returned from the view.

Find a Single Product

Now that you've learned how to create a WHERE clause, you can use this same kind of code to locate a record by its primary key. The ProductID column in the SalesLT.Product table is the primary key, so you want to create a SELECT statement that looks like the following:

```
SELECT * FROM SalesLT.Product
WHERE ProductID = @ProductID
```

Use the [Key] Attribute

To do this, you must identity the property in the Product class that holds the primary key. You're going to do this using the [Key] attribute class that .NET provides. Open the **Product.cs** file and add a using statement.

```
using System.ComponentModel.DataAnnotations;
```

Add the [Key] attribute above the **Id** property.

```
[Key]
[Column("ProductID")]
public int Id { get; set; }
```

Open the **ColumnMapper.cs** file and add a new property called ***IsKeyField*** so that as you are looping through and building the list of properties, you can set this Boolean property to true for the property decorated with the [Key] attribute.

```
public bool IsKeyField { get; set; }
```

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

```
using System.ComponentModel.DataAnnotations;
```

Locate the BuildColumnCollection() method and just below the code where you check for a ColumnAttribute and set the colMap.ColumnName, add the following code to check for the [Key] attribute:

```
// Is the column a primary [Key]?
KeyAttribute key = prop
  .GetCustomAttribute<KeyAttribute>();
colMap.IsKeyField = key != null;
```

Add a Find() Method

Add a new method named Find() to the RepositoryBase class, as shown in **Listing 11**. This method has the same signature as the LINQ Find() method, where you pass in one or more values to a parameter array. Most tables only have a single field as their primary key, but in case a table has a composite key, you need to have a parameter array for those additional values.

The BuildSelectSql() method creates the SELECT statement, and the **Columns** property. Next, the **searchColumns** variable is created as a list of ColumnMapper objects with just those columns where the **IsKeyField** property is set to true. Ensure that the number of values passed into the parameter array are equal to the number of properties with the [Key] attribute. If these two numbers don't match, throw an ApplicationException object.

Loop through the collection of **searchColumns** and fill in the **ParameterValue** property for each ColumnWrapper object in the list. Set the **SearchOperator** property for each to be an equal sign because you're looking for an exact match.

Build the WHERE clause for the SELECT statement by using the BuildSearchWhereClause() method you created earlier. Build the SqlCommand object and then build the parameters for the WHERE clause by calling the BuildWhereClauseParameters() method.

Call the overload of the Find() method shown in **Listing 12**. This method is responsible for passing the command object to the Search() method and retrieving the results back. Check the results to ensure values were found, and if there's at least one product in the list, assign the first item to the **ret** variable to be returned from this method. If no values are found, a null value is returned just like the LINQ Find() method.

Now that you have the generic Find() methods written in the RepositoryBase class, open the **ProductRepository.cs** file and add the Find() method that accepts an integer value that relates to the ProductID field in the Product table.

Listing 13: The Find() method returns a null if the record is not found, or it returns a valid entity object

```
using AdvWorksDbContext db = new(ConnectString);

Product entity = db.Products.Find(706);

Console.WriteLine("*** Get Product Data ***");
if (entity == null) {
  Console.WriteLine(
    "Can't Find Product ID=706");
}
else {
  // Display Data
  Console.WriteLine(entity.ToString());
  Console.WriteLine();
  Console.WriteLine(
    $"SQL Submitted: {db.Products.SQL}");
}
Console.WriteLine();
```

```
public virtual Product Find(int id) {
  return base.Find<Product>(id);
}
```

Try It Out

Open the **Program.cs** file and change the code to call the Find() method, as shown in **Listing 13**. This method should check to ensure that a single entity class is returned. If the value returned is null, write a message into the console window, otherwise, write the product entity into the console window. Run the console application and you should see a single product object displayed. You may need to change the product ID to match an ID from your SalesLT.Product table.

Get a Scalar Value

If you need to retrieve the value from one of the many aggregate functions in SQL Server, such as Count(), Sum(), Avg(), etc., expose a method named ExecuteScalar() from the RepositoryBase class. To retrieve the count of all records in the Product table, submit a SQL statement such as the following:

```
SELECT COUNT(*) FROM SalesLT.Product;
```

Place this SQL statement into a Command object and call the ExecuteScalar() method on the Command object. Open the **RepositoryBase.cs** file and add a new method. Because you don't know what type of object you're going to get back, return an object data type.

```
public virtual object
  ExecuteScalar(IDbCommand cmd) {
  object ret;

  // Open the Connection
  DbContext.CommandObject.Connection.Open();

  // Call the ExecuteScalar() method
  ret = DbContext.CommandObject.ExecuteScalar();

  return ret;
}
```

Add an overload of the ExecuteScalar() method to allow you pass in a simple SQL statement. This method then creates the Command object and passes it to the previous ExecuteScalar() overload for processing.

```
public virtual object
  ExecuteScalar(string sql) {
  // Store the SQL submitted
  SQL = sql;

  // Create Command object with SQL
  DbContext.CreateCommand(SQL);

  // Return the value
  return ExecuteScalar(DbContext.CommandObject);
}
```

Try It Out

Open the **Program.cs** file and add code to test this out.

```
using AdvWorksDbContext db = new(ConnectString);

string sql = "SELECT COUNT(*)
              FROM SalesLT.Product";
int rows = (int)db.Database.ExecuteScalar(sql);
```

```
Console.WriteLine(
  "*** ExecuteScalar(sql) Sample ***");
// Display Result
Console.WriteLine(rows);
Console.WriteLine();
Console.WriteLine(
  $"SQL Submitted: {db.Database.SQL}");
Console.WriteLine();
```

Run this application and you should see the total number of products within the Product table appear in the console window.

Multiple Result Sets

Sometimes, retrieving multiple result sets can help you cut down the number of roundtrips to your SQL Server. A data reader object supports reading one result set and then advancing to the next. Let's look at how this works with the wrapper classes you've created so far.

Create New Search() Method Overload

Open up the **RepositoryBase.cs** file and create a new overload of the Search() method, as shown in **Listing 14**. This method accepts both a command object and a data reader, and it's responsible for calling the BuildEntityList() method.

Modify the old Search() method to have it now call this new overload, as shown in the code snippet below. Remove the declaration of the **ret** variable, and modify the return statement to call the new overloaded Search() method.

```
public virtual List<TEntity> Search<TEntity>
  (IDbCommand cmd) {
  // Build Columns if needed
  if (Columns.Count == 0) {
    Columns = BuildColumnCollection<TEntity>();
  }

  // Set Command Object
  DbContext.CommandObject = cmd;

  return Search<TEntity>(cmd,
    DbContext.CreateDataReader());
}
```

Add a Customer Entity Class

To illustrate multiple result sets, you need a new entity class. In the AdventureWorksLT database, there's a Customer table. Let's create a new **Customer.cs** file and add the code shown in **Listing 15** to model that table.

Add a View Model Class

Instead of writing the code to handle multiple result sets in the Program.cs file, create a new view model class to encapsulate the functionality of reading both product and customer data. Right mouse-click on the project and create a folder named **ViewModelClasses**. Right mouse-click on the ViewModelClasses folder and add a new class named **ProductCustomerViewModel.cs** and add the code shown in **Listing 16**.

The code in the LoadProductsAndCustomers() method creates a string with two SQL statements in it. An instance of

Listing 14: Add a new Search() method that takes an IDataReader object

```
public virtual List<TEntity>
  Search<TEntity>(IDbCommand cmd,
    IDataReader rdr) {
  List<TEntity> ret;

  // Build Columns if needed
  if (Columns.Count == 0) {
    Columns = BuildColumnCollection<TEntity>();
  }

  // Set Command Object
  DbContext.CommandObject = cmd;

  // Get the list of entity objects
  ret = BuildEntityList<TEntity>(rdr);

  return ret;
}
```

Listing 15: Add a new entity class to illustrate how to get multiple result sets

```
#nullable disable

using System.ComponentModel.DataAnnotations;
using System.ComponentModel
  .DataAnnotations.Schema;

namespace AdoNetWrapperSamples.EntityClasses;

[Table("Customer", Schema = "SalesLT")]
public partial class Customer
{
  [Key]
  public int CustomerID { get; set; }
  public string Title { get; set; }
  public string FirstName { get; set; }
  public string MiddleName { get; set; }
  public string LastName { get; set; }
  public string CompanyName { get; set; }

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

Listing 16: Create a class to wrap up both result sets

```
#nullable disable

using AdoNetWrapperSamples.EntityClasses;
using AdoNetWrapperSamples.Models;

namespace AdoNetWrapperSamples.ViewModelClasses;

public class ProductCustomerViewModel {
  public ProductCustomerViewModel
    (string connectString) {
    ConnectString = connectString;
  }

  public string ConnectString { get; set; }
  public List<Product> Products { get; set; }
  public List<Customer> Customers { get; set; }

  public void LoadProductsAndCustomers() {
    string sql = "SELECT *
      FROM SalesLT.Product;";
    sql += "SELECT * FROM SalesLT.Customer";

    using AdvWorksDbContext db =
      new(ConnectString);

    // Create Command object
    var cmd = db.CreateCommand(sql);

    // Get the Product Data
    Products = db.Database.Search<Product>(cmd);

    // Advance to next result set
    db.DataReaderObject.NextResult();

    // Clear columns to get ready
    // for next result set
    db.Database.Columns = new();

    // Get the Customer Data
    Customers = db.Database
      .Search<Customer>(cmd, db.DataReaderObject);
  }
}
```

the AdvWorksDbContext class is created with a **using** block so all connection objects are disposed of properly. Next a SqlCommand object is created by calling the CreateCommand() method on the database context object.

The Search<Product>() method is called to load the set of product data. Call the NextResult() method on the data reader object to move to the next result set. Clear the current list of ColumnWrapper objects because that list of columns is for the Product data set. Finally, call the Search<Customer>() method passing in the command object and the current data reader object, which is now ready to loop through the customer records.

Try It Out

To try this code out to make sure it works, open the **Program.cs** file. Put the code shown below just after the code that retrieves the connection string.

```
ProductCustomerViewModel vm = new(ConnectString);

vm.LoadProductsAndCustomers();

// Display Products
foreach (var item in vm.Products) {
  Console.WriteLine(item);
}

// Display Customers
foreach (var item in vm.Customers) {
  Console.WriteLine(item);
}
```

Run the application and you should see the list of products and customers appear in the console window.

Search for Data Using a Stored Procedure

Another common method of retrieving data from a database is to call a stored procedure. If you have a three (or more) table join, it's a best practice to move that code to a stored procedure or a view in your database. Keeping complicated queries out of your C# code is better for readability and maintenance. It also allows you to tune the join in the server. Let's look at calling a stored procedure using the ADO.NET wrapper classes. Create a stored procedure in the AdventureWorksLT database named **Product_Search**, as shon in **Listing 17**.

Listing 17: Create a stored procedure to perform searching

```
CREATE PROCEDURE [SalesLT].[Product_Search]
  @Name nvarchar(50) null,
  @ProductNumber nvarchar(25) null,
  @BeginningCost money null,
  @EndingCost money null
AS
BEGIN
  SELECT *
  FROM SalesLT.Product
    WHERE (@Name IS NULL OR
      Name LIKE @Name + '%')
    AND   (@ProductNumber IS NULL OR
      ProductNumber LIKE @ProductNumber + '%')
    AND   (@BeginningCost IS NULL OR
      StandardCost >= @BeginningCost)
  AND   (@EndingCost IS NULL OR
        StandardCost <= @EndingCost)
END
```

Create Parameter Class for Calling a Stored Procedure

Because the Product_Search stored procedure has four parameters, you should create a class with four properties. Right mouse-click on the project and add a new folder named **ParameterClasses**. Right mouse-click on the ParametersClasses folder and add a new class named **ProductSearchParam**. The property names should match the parameter names within the stored procedure.

```
#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.ParameterClasses;

public class ProductSearchParam {
  public string Name { get; set; }
  public string ProductNumber { get; set; }
  public decimal? BeginningCost { get; set; }
  public decimal? EndingCost { get; set; }
}
```

Listing 18: Add new method to accept a SQL statement for calling a stored procedure

```
public virtual List<TEntity>
  SearchUsingStoredProcedure<TEntity, TParam>
    (TParam param, string sql) {
  List<ColumnMapper> searchColumns = new();
  List<TEntity> ret;

  // Store the SQL submitted
  SQL = sql;

  // Build columns collection for entity class
  Columns = BuildColumnCollection<TEntity>();

  // Create Command Object with SQL
  DbContext.CreateCommand(SQL);

  // Set CommandType to Stored Procedure
  DbContext.CommandObject.CommandType =
    CommandType.StoredProcedure;

  if (param!= null) {
    // Build a collection of ColumnMapper objects
    // based on properties in the TParam object
    searchColumns = BuildSearchColumnCollection
      <TEntity, TParam>( param);

    // Add any Parameters?
    if (searchColumns != null &&
        searchColumns.Count > 0) {
      BuildWhereClauseParameters(
        DbContext.CommandObject, searchColumns);
    }
  }

  ret = BuildEntityList<TEntity>
    (DbContext.CreateDataReader());

  return ret;
}
```

Listing 19: Modify the BuildWhereClauseParameters() method to set a DBNull.Value

```
protected virtual void
  BuildWhereClauseParameters(IDbCommand cmd,
    List<ColumnMapper> whereColumns) {
  // Add parameters for each key value passed in
  foreach (ColumnMapper item in whereColumns) {
    var param = DbContext.CreateParameter(
      item.ColumnName,
      item.SearchOperator == "LIKE" ?
      item.ParameterValue + "%" :
      item.ParameterValue);

    // Add parameter value or DBNull value
    param.Value ??= DBNull.Value;

    cmd.Parameters.Add(param);

    if (cmd.CommandType !=
        CommandType.StoredProcedure) {
      // Store parameter info
      Columns.Find(c => c.ColumnName ==
        item.ColumnName)
        .ParameterValue = item.ParameterValue;
    }
  }
}
```

Add Method to Call Stored Procedure

Open the **RepositoryBase.cs** file and create a new method named SearchUsingStoredProcedure(), as shown in **Listing 18.** In this method, pass in an instance of the parameter class and a SQL string that contains the name of the stored procedure. Assign the SQL string passed to the **SQL** property and build the columns collection for the entity class collection to be returned.

Create the command object and assign the **CommandType** property of the command object to the enumeration **CommandType.StoredProcedure**. Check the **param** parameter to ensure that it isn't null. If not, build the collection of search columns to use to build the set of parameters that will be passed to the stored procedure. You can use the same BuildWhereClauseParameters() method you used before, as this adds parameters to the command object based on the set of ColumnWrapper objects passed to it. Finally, call the stored procedure and use the result set to build the collection of entity objects.

When you were building the WHERE clause for a dynamic SQL statement, you only needed to create ColumnWrapper object for those properties in the search class that had a value in them. When calling a stored procedure, you need to create a ColumnWrapper object for all parameters whether or not there is a value in them. Locate the BuildSearchColumnCollection() method and within the foreach() loop, modify the **if** statement that checks to see if the **value is not null** to look like the following.

```
if (value != null ||
   (DbContext.CommandObject != null &&
    DbContext.CommandObject.CommandType ==
      CommandType.StoredProcedure)) {
```

One more location you need to change code to support calling stored procedures is within the BuildWhereClauseParameters() method. As you loop through each ColumnWrapper object to build the parameter, you're going either set the parameters' **Value** property to the value from the search class, or a **DBNull.Value**. Also change it so the **ParameterValue** property is set back into the collection of entity columns only if you are not calling a stored procedure. This is because the parameter names passed to the stored procedure may not be the same names as the property names in the entity column collection. Modify the BuildWhereClauseParameters() method to look like the code shown in **Listing 19**.

Try It Out

Open the **Program.cs** file and modify the code after retrieving the connection string to look like **Listing 20.** Run the console application and you should see only products with names starting with the letter C appearing in the console window.

Call Stored Procedure with No Parameters

If you have a stored procedure that doesn't have any parameters, you can call that as well. Just pass a null value as the first parameter to the new Search() overload you just added. As an example, create the following stored procedure in the AdventureWorksLT database:

```
CREATE PROCEDURE [SalesLT].[Product_GetAll]
AS
```

```
BEGIN
  SELECT *
  FROM SalesLT.Product;
END
```

Try It Out

Open the **Program.cs** file and modify the line of code that sets the name of the stored procedure to call.

```
string sql = "SalesLT.Product_GetAll";
```

Next, modify the line of code that calls the SearchUsingStoredProcedure() method. The TEntity and TParam types passed should both be the Product entity class. Pass a null value to the first parameter to avoid creating any parameters for this stored procedure call.

```
List<Product> list = db.Database
  .SearchUsingStoredProcedure
    <Product, Product>(null, sql);
```

Run the console application and you should see all of the product data displayed after making this call to the stored procedure.

Stored Procedure with Output Parameter

Stored procedures can not only have input parameters, but output parameters as well. To retrieve the value from an OUTPUT parameter, you need to ensure that you read the parameter immediately after calling the stored procedure. If you're reading data using a data reader, you need to close the reader, but NOT close the connection. To test this, create the following stored procedure in the AdventureWorksLT database:

```
CREATE PROCEDURE
  [SalesLT].[Product_GetAllWithOutput]
  @Result nvarchar(10) OUTPUT
AS
BEGIN
  SELECT *
  FROM SalesLT.Product;

  /* Set the output parameter */
  SELECT @Result = 'Success';
END
```

Create [OutputParam] Attribute

You need to inform the RepositoryBase class if you're going to have an OUTPUT parameter that needs to be returned. An easy way to do this is to create another attribute. Right mouse-click on the **Common** folder, create a new class named **OutputParamAttribute**, and enter the code shown below in this new file.

```
#nullable disable
using System.Data;
namespace AdoNetWrapper.Common;

[AttributeUsage(AttributeTargets.Property)]
public class OutputParamAttribute:Attribute {
  public ParameterDirection Direction
   { get; set; }
  public DbType DbType { get; set; }
  public int Size { get; set; }

  public OutputParamAttribute(
    ParameterDirection direction) {
    Direction = direction;
  }
}
```

The OutputParamAttribute class inherits from the Attribute class and exposes three public properties. The **Direction** property is the one exposed from the constructor, as that's the one you're going to use the most.

Create Search Class with OutputParam Attribute

Any time you have a stored procedure with parameters, you need to build a parameter class to map to those parameters. Right mouse-click on the **ParameterClasses** folder, create a new class named **ProductGetAllParam**, and enter the code shown below into this new file. Notice that the **Result** property is decorated with the new [OutputParam] attribute you just created.

```
#nullable disable

using AdoNetWrapper.Common;
using System.Data;

namespace AdoNetWrapperSamples.ParameterClasses;

public class ProductGetAllParam {
  [OutputParam(ParameterDirection.Output,
    Size = 10)]
  public string Result { get; set; }
}
```

Modify ColumnMapper Class

Because you now have additional properties within the [OutputParam] attribute, you need to add these same properties to the ColumnMapper class. As you iterate over the properties for a search class, you can store the data from the [OutputParam] attribute into the ColumnMapper object for use when calling the stored procedure. Open the **ColumnMapper.cs** file and add a Using statement.

```
using System.Data;
```

Listing 20: Call a stored procedure using the SearchUsingStoredProcedure() method

```
using AdvWorksDbContext db = new(ConnectString);

string sql = "SalesLT.Product_Search";
ProductSearchParam param = new() {
  Name = "C"
};

List<Product> list = db.Database
  .SearchUsingStoredProcedure<Product,
    ProductSearchParam>(param, sql);

// Display Products
foreach (var item in list) {
  Console.WriteLine(item);
}

Console.WriteLine();
Console.WriteLine(
  $"Total Items: {list.Count}");
Console.WriteLine();
Console.WriteLine(
  $"SQL Submitted: {db.Database.SQL}");
```

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.pdsa.com/downloads. Select "Articles" from the Category drop-down. Then select " Simplifying ADO.NET Code in .NET 6: Part 2" from the Item drop-down.

Add the following new properties to the ColumnWrapper class.

```
public ParameterDirection Direction
  { get; set; }
public DbType DbType { get; set; }
public int Size { get; set; }
```

Add a constructor to the ColumnMapper class to set the default parameter direction to **Input**. Also take this opportunity to initialize the ***SearchOperator*** the equal sign (=).

```
public ColumnMapper() {
  SearchOperator = "=";
  Direction = ParameterDirection.Input;
}
```

Modify the BuildSearchColumnCollection() Method

Open the **RepositoryBase.cs** file and modify the BuildSearchColumnCollection() method to check for an [OutputParam] attribute. If one is found, transfer the properties found in the OutputParam into the ColumnWrapper object. Within the foreach loop, after the code that checks for a [Search] attribute, add the following code to check for an [OutputParam] attribute.

```
// Does Property have an [OutputParam] attribute
OutputParamAttribute oa = Prop
  .GetCustomAttribute<OutputParamAttribute>();
if (oa != null) {
  colMap.Direction = oa.Direction;
  colMap.DbType = oa.DbType;
  colMap.Size = oa.Size;
}
```

Modify the BuildSearchWhereClause() Method

Now locate the BuildSearchWhereClause() method and modify the code in the foreach() to only retrieve those columns where the **Direction** property is either **Input** or **InputOutput**. Those properties that have a **Direction** set to **Output** don't need to be included in the WHERE clause.

```
foreach (var item in columns
   .Where(c => c.Direction ==
     ParameterDirection.Input
     || c.Direction ==
       ParameterDirection.InputOutput)) {
```

Modify the BuildWhereClauseParameters() Method

Find the BuildWhereClauseParameters() method and modify the foreach() to only retrieve those columns where the **Direction** property is either **Input** or **InputOutput**.

```
foreach (ColumnMapper item in whereColumns
  .Where(c => c.Direction ==
    ParameterDirection.Input
    || c.Direction ==
      ParameterDirection.InputOutput)) {
```

Add a BuildOutputParameters() Method

For working with stored procedure OUTPUT parameters, build a new method to handle those columns in the search class that are decorated with the [OutputParam] attribute. Create a new method named BuildOutputParameters that accepts a Command object and a list of columns from the search class. In the foreach() iterator, you're only going to extract those columns where the **Direction** property is either **Output** or **InputOutput**.

```
protected virtual void BuildOutputParameters
  (IDbCommand cmd, List<ColumnMapper> columns) {
  // Add output parameters
  foreach (ColumnMapper item in columns
     .Where(c => c.Direction ==
              ParameterDirection.Output ||
            c.Direction ==
              ParameterDirection.InputOutput)) {
    var param = DbContext.CreateParameter(
      item.ColumnName, null);
    param.Direction = item.Direction;
    param.DbType = item.DbType;
    cmd.Parameters.Add(param);
  }
}
```

Listing 21: Create a new method to get the output parameter values

```
protected virtual void GetOutputParameters
  <TParam>(TParam param,
    List<ColumnMapper> columns) {
  // Get output parameters
  foreach (ColumnMapper item in columns
    .Where(c => c.Direction ==
       ParameterDirection.Output ||
       c.Direction ==
       ParameterDirection.InputOutput)) {
    // Get the output parameter
    var outParam = DbContext
      .GetParameter(item.ColumnName);
    // Set the value on the parameter object
    typeof(TParam).GetProperty(item.ColumnName)
     .SetValue(param, outParam.Value, null);
  }
}
```

Listing 22: Create a SqlServerRepositoryBase class to override those methods that have SQL Server specific functionality

```
#nullable disable

using System.Data;
using System.Data.SqlClient;

namespace AdoNetWrapper.Common;

public class SqlServerRepositoryBase
   : RepositoryBase {
  public SqlServerRepositoryBase(
    SqlServerDatabaseContext context)
      : base(context) { }

  protected override void
    BuildOutputParameters(IDbCommand cmd,
      List<ColumnMapper> columns) {
    // Add output parameters
    foreach (ColumnMapper item in columns
      .Where(c => c.Direction ==
              ParameterDirection.Output)) {
      var param = (SqlParameter)DbContext
        .CreateParameter(item.ColumnName, null);
      param.Direction = item.Direction;
      param.DbType = item.DbType;
      // Need to set the Size for SQL Server
      param.Size = item.Size;
      cmd.Parameters.Add(param);
    }
  }
}
```

Listing 23: Modify the SearchUsingStoredProcedure() method to build output parameters

```
if (param != null) {
  // Build collection of ColumnMapper objects
  // based on properties in the TParam object
  searchColumns = BuildSearchColumnCollection
    <TEntity, TParam>(param);

  // Add any Parameters?
  if (searchColumns != null &&
      searchColumns.Count > 0) {
    BuildWhereClauseParameters(DbContext
     .CommandObject, searchColumns);
  }
  // Add any Output Parameters?
  if (searchColumns.Where(c => c.Direction ==
    ParameterDirection.Output ||
    c.Direction ==
    ParameterDirection.InputOutput).Any()) {
    BuildOutputParameters(DbContext.CommandObject,
      searchColumns);
  }
}
```

Add GetOutputParameters() Method

After the stored procedure has been processed is when you may retrieve any OUTPUT parameters. Create a new method named GetOutputParameters() (shown in **Listing 21**) to iterate over the search columns and retrieve the value from the stored procedure and place it into the appropriate property of the search class.

Create SqlServerRespositoryBase Class

When using SQL Server to retrieve OUTPUT parameters, you must set the **Size** property when adding the parameter to the Command object. This might not be true for all .NET data providers, but you need it for SQL Server. Unfortunately, the **Size** parameter does not exist on the IDbCommand interface, so you must create a SqlServerRepositoryBase class that inherits from the RepositoryBase class and override the BuildOutputParameters() method. Within this override, you set the **Size** property on the parameter object. Right mouse-click on the **Common** folder and add a new class named **SqlServerRepositoryBase**. Place the code shown in **Listing 22** into this new file.

Modify SearchUsingStoredProcedure() Method

Open the **RepositoryBase.cs** file and locate the SearchUsingStoredProcedure() method. Within the If statement (**Listing 23**) that checks that the ***param*** variable is not null, add a new If statement immediately after the existing If statement.

Move a little further down in this method and, just after the call to the BuildEntityList() method and before the **return** statement, add the following code to retrieve any output parameters:

```
// Retrieve Any Output Parameters
if (searchColumns.Where(c => c.Direction ==
   ParameterDirection.Output ||
   c.Direction ==
   ParameterDirection.InputOutput).Any()) {
  // Must close DataReader for output
  // parameters to be available
  DbContext.DataReaderObject.Close();

  GetOutputParameters(param, searchColumns);
}
```

Try It Out

Open the **AdvWorksDbContext.cs** file and modify the ***Database*** property to use the new SqlServerRepositoryBase class.

```
public SqlServerRepositoryBase Database
  { get; set; }
```

Open the **Program.cs** file and modify the code to look like the following.

```
string sql = "SalesLT.Product_GetAllWithOutput";
ProductGetAllParam param = new() {
  Result = ""
};

List<Product> list = db.Database
  .SearchUsingStoredProcedure<Product,
    ProductGetAllParam>(param, sql);
```

Add the following code after the loop displaying all the items returned.

```
Console.WriteLine();
Console.WriteLine($"Output Param:
  '{param.Result}'");
```

Run the console application and you should see the OUTPUT parameter named **Result** appear after all the products have been displayed.

Summary

This article built more functionality into the wrapper classes around ADO.NET to give you the ability to add WHERE clauses to SELECT statements. In addition, you saw how to retrieve data from views and stored procedures. Multiple result sets can be handled, and you can now retrieve scalar values. The best thing is that most of the code is going into generic classes, so as you add more classes to work with more tables, the code you write for each of those is minimal.

In the next article, you'll learn to insert, update, and delete data. You will also learn to submit transactions, validate data using data annotations, and to handle exceptions.

Paul D. Sheriff

ONLINE QUICK ID 2209051

Customized Object-Oriented and Client-Server Scripting in C#

In this article, I'm going to talk about using a custom object-oriented scripting in C#. By "custom," I mean that all you're going to see here is available to use and modify from GitHub. By "C#," I mean that the scripting language is implemented in C# and you can just include it in your project in order to adjust it as you wish. As a scripting language, I'm going to use CSCS (customized scripting in C#). I've talked about this language in a few previous CODE Magazine articles (see links in the sidebar). CSCS is an open-source scripting language that's very easy to integrate into your C# project.

Vassili Kaplan
VassiliK@gmail.com

Vassili is a former Microsoft Lync developer. He's been studying and working in a few countries, such as Russia, Mexico, the USA, and Switzerland.

He has a Masters in Applied Mathematics with Specialization in Computational Sciences from Purdue University, West Lafayette, Indiana and a Bachelor in Applied Mathematics from ITAM, Mexico City.

In his spare time, Vassili works on the CSCS scripting language. His other hobbies are traveling, biking, badminton, and enjoying a glass of a good red wine.

You can contact him through his website: http://www.iLanguage.ch or e-mail: vassilik@gmail.com

You're going to see how to use classes and objects in scripting, and also how they're implemented in C#. It's important that you have a full control of how the object-oriented functionality is implemented. For instance, you can have multiple inheritance in scripting, which is forbidden in C# or in JavaScript. But you could also disable it if you think that it's against your beliefs. It's important that you, and not another architect, decide what features you want to have to solve a particular problem.

> The great thing about object-oriented code is that it can make small, simple problems look like large, complex ones.
>
> *Anonymous*

As an example of using object-oriented scripting, I'm going to take a look at a client-server application, where I'll show how you can send and receive objects. I'll also show a simple marshalling-unmarshalling mechanism (converting objects to a string and back) to pass data across the wire. You can use a similar approach for any custom client-server communication, just using a couple of lines of a scripting code.

To distinguish between the C# code and CSCS scripting code, all C# code is provided below with the syntax highlighting, whereas all scripting code doesn't use it.

Let's start by looking at how you can set up scripting in your .NET Visual Studio project.

Setting Up CSCS Scripting

One of the simplest ways to start using CSCS scripting is to download the source code from GitHub (see https://github.com/vassilych/cscs) and add the source code directly to your C# .NET project. The license lets you modify and use the code without any restrictions.

An example of including the CSCS Scripting Engine in a Windows GUI project is a WPF project, available here: https://github.com/vassilych/cscs_wpf.

Another example is a Xamarin iOS—Android mobile project that can be downloaded from here: https://github.com/vassilych/mobile.

CSCS is a functional language, syntactically very similar to JavaScript. To add a new functionality to CSCS, you'll need to perform just these three steps:

1. Define a CSCS function name as a constant. When parsing this constant, the CSCS parser triggers the appropriate implementation code.
2. Implement a new class, deriving from the ParserFunction class. The most important method is Evaluate(). It will be triggered when the constant defined in the previous step is parsed.
3. Register the newly created class with the parser.

Let's see how this is done using the implementation of the power function .

First, you define an appropriate constant in the Constants.cs file:

```
public const string MATH_POW = "Math.Pow";
```

Next, you define the implementation.

```
class PowFunction : ParserFunction {
  protected override Variable Evaluate(
       ParsingScript script) {
    List<Variable> args = script.
             GetFunctionArgs();
    Utils.CheckArgs(args.Count, 2, m_name, true);
    Variable arg1 = args[0];
    Variable arg2 = args[1];

    arg1.Value = Math.Pow(arg1.Value,
                          arg2.Value);
    return arg1;
  }
  public override string Description() {
    return "Returns a specified number \
          raised to the specified power.";
  }
}
```

Finally, the last step is registering this new functionality with the parser at the program initialization stage:

```
ParserFunction.RegisterFunction(
    Constants.MATH_POW, new PowFunction());
```

You're done now. As soon as the parser sees something like Math.Pow(2, 5), the Evaluate() method above is triggered and the correct value of 32 calculated.

The Description method is triggered when the user calls a Help scripting method.

Note the convenient method script.GetFunctionArgs(). It returns all comma-separated arguments between the parentheses (e.g., it returns 2 and 5 for Math.Pow(2, 5)). You can also put some variables and arrays as function arguments—their value will be recursively extracted during the GetFunctionArgs() call.

> Object-oriented programming had boldly promised "to model the world." Well, the world is a scary place where bad things happen for no apparent reason, and in this narrow sense I concede that OO does model the world.
>
> *Dave Fancher*

In the next sections, I'm going to show how you can use the new function definition shown in this section to define classes and objects.

"Hello, World!" in Object-Oriented Scripting

Let's first see how classes and objects are defined and used in scripting and then how they are implemented in C#.

I hope you'll find this very intuitive and similar to other languages, with some few exceptions (like multiple inheritance).

> With enough practice, any interface is intuitive.
>
> *Anonymous*

Let's see two simple examples of a class definition in CSCS:

```
class Stuff1 {
  x = 2;
  Stuff1(a) {
    x = a;
  }
  function helloWorld() {
    print("Hello, World!");
  }
}

class Stuff2 {
  y = 3;
  Stuff2(b) {
    y = b;
  }
  function addStuff2(n) {
    return n + y;
  }
}
```

You can now create new objects and use these classes as usual:

```
obj1 = new Stuff1(10);
obj2 = new Stuff2(5);
print(obj1.X + obj2.Y); // prints 15.
print(obj1); // prints stuff1.obj1[x=10]
```

Now let's use multiple inheritance, something you can't do in many modern languages. Let's define a class that inherits both the method implementations and variables from the base classes:

```
class CoolStuff : Stuff1, Stuff2 {
  z = 3;
  CoolStuff(a=1, b=2, c=3) {
    x = a;
    y = b;
    z = c;
  }
  function addCoolStuff() {
    return x + addStuff2(z);
  }
  function ToString() {
    return "{" + x + "," + y + "," + z + "}";
  }
}
```

Here's how you can use this newly defined class:

```
obj3 = new CoolStuff(11, 22, 33);
obj3.HelloWorld(); // prints "Hello, World!"
print(obj3.AddStuff2(20)); // prints 42
print(obj3); // prints {11,22,33}
```

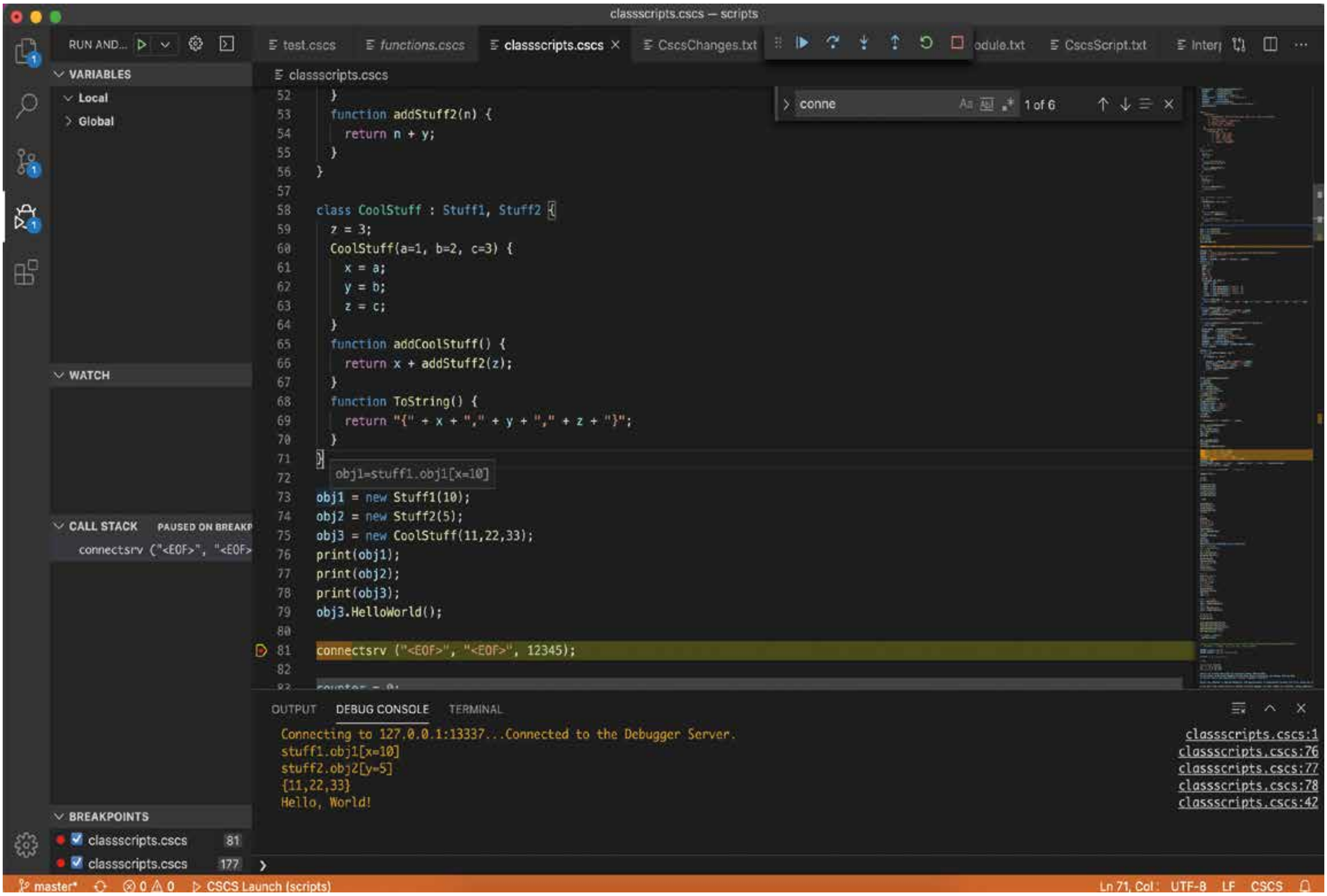
As you can see, both variables and methods can be used from the base classes. A special method is ToString(). When defined, it overrides the string representation of the object (e.g., what's printed in print(object) statement). The default ToString() implementation is the following: ClassName.InstanceName[variable,variable2,...].

You probably noted that some of the class methods start with a lowercase letter, others with an uppercase: it doesn't matter, CSCS scripting language is case insensitive.

> CSCS scripting language is case-insensitive.

You can also debug a CSCS script. The easiest method is to install the CSCS Debugger and REPL Extension for Visual Studio Code (https://marketplace.visualstudio.com/items?itemName=vassilik.cscs-debugger). This CODE Magazine article explains how to use Visual Studio Code Extensions for debugging: https://www.codemag.com/article/1809051.

Figure 1 shows a debugging session with some CSCS scripting statements.



```
  }
  function addStuff2(n) {
    return n + y;
  }
}

class CoolStuff : Stuff1, Stuff2 {
  z = 3;
  CoolStuff(a=1, b=2, c=3) {
    x = a;
    y = b;
    z = c;
  }
  function addCoolStuff() {
    return x + addStuff2(z);
  }
  function ToString() {
    return "{" + x + "," + y + "," + z + "}";
  }
}

obj1 = new Stuff1(10);
obj2 = new Stuff2(5);
obj3 = new CoolStuff(11,22,33);
print(obj1);
print(obj2);
print(obj3);
obj3.HelloWorld();

connectsrv ("<EOF>", "<EOF>", 12345);
```

```
Connecting to 127.0.0.1:13337...Connected to the Debugger Server.    classscripts.cscs:1
stuff1.obj1[x=10]                                                     classscripts.cscs:76
stuff2.obj2[y=5]                                                      classscripts.cscs:77
{11,22,33}                                                            classscripts.cscs:78
Hello, World!                                                         classscripts.cscs:42
```

Figure 1: Debugging CSCS Scripting with Visual Studio Code on macOS

Implementing Scripting Classes and Objects in C#

Let's see briefly how the classes and objects scripting functionality from the previous section is implemented in C#. As you previously saw with the Math.Pow() example, all of the CSCS functionality is implemented as functions. Yes, even classes are implemented this way, no matter how strange it sounds.

When the CSCS parser reads a class definition, that starts with **Class ClassName ...**, the C# implementation is triggered (see **Listing 1**).

The code in **Listing 1** defines a new class, which can now be instantiated in CSCS. This also needs to be registered with the parser before being used:

```
ParserFunction.RegisterFunction(Constants.CLASS,
  new ClassCreator());
  // CLASS is defined as "class"
```

As soon as the CSCS parser sees this statement, obj1 = new Stuff1..., another C# implementation is triggered, namely the Evaluate() method of the NewObjectFunction class (see Listing 2). The NewObjectFunction must also be registered with the CSCS parser as follows:

```
ParserFunction.RegisterFunction(Constants.NEW,
  new NewObjectFunction());
  // NEW is defined as "new"
```

I encourage you to take a look at the CSCS GitHub page (https://github.com/vassilych/cscs) for more implementation details.

Next, let's see an example of using scripting to access Web Services.

Accessing Web Services from Scripting

As an example of accessing a Web Service, you're going to use Alpha Vantage Web Service (https://www.alphavantage.co). Alpha Vantage provides a financial market data API.

The main advantages of using Alpha Vantage are that it's pretty straightforward to create a request and that it's also free to use (well, up to five requests per minute or 500 requests per day, as of this writing). To replicate what you're doing here, you need to request a free API key here: https://www.alphavantage.co/support/#api-key.

Here is how you create a URL to access their Web Service in CSCS:

Listing 1: C# Code to create a scripting class

```
public class ClassCreator : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    string className = Utils.GetToken(script);
    string[] baseClasses = Utils.GetBaseClasses(script);
    var newClass = new CSCSClass(className, baseClasses);

    newClass.ParentOffset = script.Pointer;
    newClass.ParentScript = script;
    string scriptExpr = Utils.GetBodyBetween(script,
        Constants.START_GROUP, Constants.END_GROUP);
    string body = Utils.ConvertToScript(Utils.GetBodyBetween(
      script, Constants.START_GROUP, Constants.END_GROUP, out _);

    ParsingScript tempScript = script.GetTempScript(body);
    tempScript.CurrentClass = newClass;
    tempScript.DisableBreakpoints = true;
    var result = tempScript.ExecuteScript();
    return result;
  }
}
```

Listing 2: C# code for the new object implementation

```
public class NewObjectFunction : ParsingFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    string className = Utils.GetToken(script);
    className = Constants.ConvertName(className);
    List<Variable> args = script.GetFunctionArgs();

    var csClass = CSCSClass.GetClass(className) as CompiledClass;
    if (csClass != null) {
      ScriptObject obj = csClass.GetImplementation(args);
      return new Variable(obj);
    }
    var instance = new CSCSClass.ClassInstance(
      script.CurrentAssign, className, args, script);

    var newObject = new Variable(instance);
    newObject.ParamName = instance.InstanceName;
    return newObject;
  }
}
```

Listing 3: JSON string returned from the Alpha Vantage Web Service

```
{
    "Meta Data": {
        "1. Information": "Daily Prices (open, high, low, close)
            and Volumes",
        "2. Symbol": "MSFT",
        "3. Last Refreshed": "2022-05-26 16:00:01",
        "4. Output Size": "Compact",
        "5. Time Zone": "US/Eastern"
    },
    "Time Series (Daily)": {
        "2022-05-26": {
            "1. open": "262.2700",
            "2. high": "267.0000",
            "3. low": "261.4300",
            "4. close": "265.9000",
            "5. volume": "24960766"
        },
        "2022-05-25": {
            "1. open": "258.1400",
            "2. high": "264.5800",
            "3. low": "257.1250",
            "4. close": "262.5200",
            "5. volume": "28547947"
        },
...
    }
};
```

```
baseURL = "https://www.alphavantage.co/" +
  "query?function=TIME_SERIES_DAILY&symbol=";
apikey  = "Y12T0TY5EUS6BXXX";
symbol  = "MSFT";
stockUrl = baseURL + symbol + "&apikey=" +
           apikey;
```

As a result, you'll get a JSON file (see an example in **Listing 3**).

This is the function to create a Web Request using CSCS scripting:

```
WebRequest(Request, Url, Load, Tracking,
  OnSuccess, OnFailure, ContentType, Headers);
```

Here is what these parameters mean:

- Request is one of the standards GET, POST, PUT, etc. For Alpha Vantage, you need GET.
- The Web service URL, as defined above.
- The load is some additional data to send.
- The Tracking variable is needed for multiple requests. When you get a response back, the Tracking variable associates it with the right request.
- OnSuccess and OnFailure are CSCS callback methods triggered when the response is received.
- The content type by default is **application/x-www-form-urlencoded**.
- You can also send some headers with the request. This is useful for the REST API requests.

All parameters, except Request and URL, are optional. If the OnSuccess and OnFailure callback methods aren't supplied, the request is executed synchronously and the result of the request is returned from the WebRequest method.

An example of accessing the Alpha Vantage Web Service is the following:

```
result = WebRequest("GET", stockUrl, "", symbol);
```

The result of this call is shown in **Listing 3** for the Microsoft stock. To be able to use the returned JSON string, there's an auxiliary CSCS GetVariableFromJSON() function. After applying this function, the main parts of the JSON string are split into a list and their subparts can be accessed by a key. Here is how you can access the resulting string (see **Listing 3** for details):

```
function processResponse(text)
{
  if (text.contains("Error")) {
    return text;
  }
  jsonFromText = GetVariableFromJSON(text);
  metaData     = jsonFromText[0];
  result       = jsonFromText[1];
  symbol       = metaData["2. Symbol"];
  last         = metaData["3. Last Refreshed"];
  allDates     = result.keys;
  dateData     = result[allDates[0]];
  myStock = new Stock(symbol, last, dateData);
  return myStock;
}
```

The processResponse() function returns a Stock object. Its class definition is shown below. The main work of processing the results is in the Stock class constructor. Here is the Stock class definition:

```
class Stock {
  symbol = "";
  date = "";
  open = 0;
  low = 0;
  high = 0;
  close = 0;
  volume = 0;
  Stock(symb, dt, data) {
    symbol = symb;
    date   = dt;
    open   = Math.Round(data["1. open"], 2);
    high   = Math.Round(data["2. high"], 2);
    low    = Math.Round(data["3. low"],  2);
    close  = Math.Round(data["4. close"],2);
    volume = data["5. volume"];
  }
}
```

Additionally, you can define a custom function for converting the Stock object into a string. An example of such a function is the following (this method should be added inside of the Stock class definition):

```
function ToString()
{
  return symbol +" "+ date + ". Open: " + open +
    ", Close: " + close + ": Low: " + low +
    ", High: " + high + ", Volume: " + Volume;
}
```

Now add the following CSCS code:

```
result = WebRequest("GET", stockUrl, "", symbol);
stock = processResponse(result);
print(stock);
```

This prints the returned Stock object according to the ToString() method defined:

```
MSFT 2022-05-26 16:00:01. Open: 262.27,
Close: 265.9: Low: 261.43, High: 267,
Volume: 24960766
```

Before getting into the main example of this article, the Client-Server communication, let's take a look at marshalling and unmarshalling objects in CSCS scripting.

Marshalling and Unmarshalling Objects

Using CSCS scripting, you can convert any object or variable to a string and back using these methods: Marshal(object) and Unmarshal(string).

The converted string looks like a simplified XML, but it's not XML. You can tweak the C# implementation code a bit if you want it to be a legal XML string.

> The only place where you should really use XML is your resume.
>
> *Anonymous*

Here's an example of marshalling a Stock object from the previous section:

```
mystock = processResponse(r);
ms = Marshal(mystock);
// Returns:
// <mystock:class:stock><symbol:STR:"MSFT">
// <date:STR:"2022-05-27"><open:NUM:268.48>
// <low:NUM:267.56><high:NUM:273.34>
// <close:NUM:273.24><volume:STR:"26910806">

ms.type; // Returns STRING
```

Here's how you construct an object back from a string:

```
ums = Unmarshal(ms);
ums.type; // Returns
// SplitAndMerge.CSCSClass+ClassInstance: Stock
```

You can also marshal and unmarshal any other data structures:

```
str = "a string";
mstr = Marshal(str);
  // Returns: <str:STR:"a string">
umstr = Unmarshal(mstr);
int = 13;
mint = Marshal(int); // Returns: <int:NUM:13>
umint = Unmarshal(mint);
```

The marshalling and unmarshalling is done recursively. Here's an example of an array (which is also a map for some elements) where one of the elements of the original array is an array itself (note that in general, that the data in an array doesn't have to be of the same type):

```
a[0]=10;
a[1]="blah";
```

```
a[2]=[9, 8, 7];
a["x:lol"]=12;
a["y"]=13;
ma = marshal(a);
maa = unmarshal(ma);
// Returns:
// ["x:lol":10, "y":11, 10, "blah", [9, 8, 7]]
maa.type; // Returns ARRAY
```

Now you're ready for the main example of this article: sending and receiving objects between a server and a client, all implemented in scripting.

A Client-Server Example

The client server example encompasses what you've seen before: a Web Server client, marshalling and unmarshalling objects, and processing JSON strings.

Sample server code does the following for each connected client: in case the request is equal to **stock**, the server interprets the load parameter as the stock name (e.g., MSFT) and then sends a stock request to the Alpha Vantage Web Service that I discussed earlier. After receiving the data, the server sends back the Stock object containing all the stock data fields.

To start a server, you just need to call a startsrv() scripting function, supplying as arguments a function to be triggered on each client connection and a port where the server is going to listen for the incoming requests. With each request, the server expects the request name and a load object (which can be an array of arguments).

Here's the scripting server-side code:

```
counter = 0;
function serverFunc(request, obj) {
  counter++;
  if (request == "stock") {
    stockUrl  = baseURL + obj + "&apikey=" +
                apikey;
    print(counter + ", request: " + stockUrl);
    data = WebRequest("GET", stockUrl, "",
                      symbol);
    result = processResponse(data);
    return result;
  }
}

startsrv("serverFunc", 12345);
```

You can update the server scripting code on the fly without restarting the server.

Note that you can change the server scripting method to be executed on each client connection on the fly without restarting the server. You can just update and redefine the serverFunc method (e.g., by using the VS Code CSCS REPL extension, mentioned earlier).

On the scripting client-side, the connecting code looks like this:

```
response = connectsrv(request, load, port,
                      host = "localhost");
```

(If the server host isn't supplied, the local host is used for connections). Let's see an example of accessing the server defined above:

```
response = connectsrv("stock", "MSFT", 12345);
print(response.Symbol + ": Close: " +
  response.Close + ", Volume: " +
  response.Volume);
// MSFT: Close: 273.24, Volume: 26910806
```

As you can see, the resulting object is returned directly from the connectsrv() call because all of the marshalling and unmarshalling is done by the scripting framework.

Wrapping Up

The main advantages of using a scripting module inside of your projects are:

- You'll save time when writing code because most of the code is usually much shorter than it would've been for making the same functionality in C#. This is what you saw with the Client-Server example.
- You can use any features not available directly in C# (e.g., multiple inheritance).
- You can modify scripting code on the fly without the necessity of recompiling and restarting the service.

I'm looking forward to your feedback, especially how you use CSCS scripting in your projects, what Web Services you access, and any performance tricks you're using.

Vassili Kaplan

References

Developing Cross-Platform Native Apps with a Functional Scripting Language:
https://www.codemag.com/Article/1711081

Using a Scripting Language to Develop Native Windows WPF GUI Apps:
https://www.codemag.com/Article/2008081

Prototyping with Microsoft Maquette: A New Virtual Reality Tool:
https://www.codemag.com/Article/2009071

Using CSCS Scripting Language for Cross-Platform Development:
https://www.smashingmagazine.com/2020/01/cscs-scripting-language-cross-platform-development

CSCS Scripting GitHub:
https://github.com/vassilych/cscs

CSCS Debugger & REPL:
https://marketplace.visualstudio.com/items?itemName=vassilik.cscs-debugger

CSCS Language eBook:
https://www.syncfusion.com/ebooks/implementing-a-custom-language

Writing Native Mobile Apps in a Functional Language Succinctly eBook:
https://www.syncfusion.com/ebooks/writing_native_mobile_apps_in_a_functional_language_succinctly

Benchmarking .NET 6 Applications Using BenchmarkDotNet: A Deep Dive

The benchmarking technique helps determine the performance measurements of one or more pieces of code in your application. You can take advantage of benchmarking to determine the areas in your source code that need to be optimized. In this article, I'll examine what benchmarking is, why benchmarking is essential, and how to benchmark .NET code using BenchmarkDotNet. If you're to work with the code examples discussed in this article, you need the following installed in your system:

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.

- Visual Studio 2022
- .NET 6.0
- ASP.NET 6.0 Runtime
- BenchmarkDotNet

If you don't already have Visual Studio 2022 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

What's a Benchmark?

A benchmark is a simple test that provides a set of quantifiable results that can help you determine whether an update to your code has increased, decreased, or had no effect on performance. It's necessary to comprehend the performance metrics of your application's methods to leverage them throughout the code optimization process. A benchmark may have a broad scope or it can be a micro-benchmark that evaluates minor changes to the source code.

Why You Should Benchmark Code

Benchmarking involves comparing the performance of code snippets, often against a predefined baseline. It's a process used to quantify the performance improvement or degradation of an application's code rewrite or refactor. In other words, benchmarking code is critical for knowing the performance metrics of your application's methods. Benchmarking also allows you to zero in on the parts of the application's code that need reworking.

There are several reasons to benchmark applications. First, benchmarking can help to identify bottlenecks in an application's performance. By identifying the bottlenecks, you can determine the changes required in your source code to improve the performance and scalability of the application.

Introducing BenchmarkDotNet

BenchmarkDotNet is an open-source library compatible with both .NET Framework and .NET Core applications that can convert your .NET methods into benchmarks, monitor those methods, and get insights into the performance data collected. BenchmarkDotNet can quickly transform your methods into benchmarks, run those benchmarks and obtain the results of the benchmarking process. In the BenchmarkDotNet terminology, an operation refers to executing a method decorated with the Benchmark attribute. A collection of such operations is known as an iteration.

What's Baselining? Why Is It Important?

You can also mark a benchmark method as a baseline method and take advantage of baselining to scale your results. When you decorate a benchmark method with the Baseline attribute and set it to "true," the summary report generated after the benchmark shows an additional column named "Ratio." This column has the value 1.00 for a benchmark method that has been baselined. All other columns will have a value relative to the Ratio column's value.

Benchmarking Application Performance in .NET 6

It's time for some measurements. Let's now examine how to benchmark the performance of .NET 6 applications. You'll create two applications: a console application for writing and executing benchmarks and an ASP.NET 6 app for building an API that will be benchmarked later.

Create a New Console Application Project in Visual Studio 2022

Let's create a console application project that you'll use for benchmarking performance. You can create a project in Visual Studio 2022 in several ways. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

To create a new Console Application Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **Console App,** and click **Next** to move on.
3. Specify the project name as BenchmarkingConsoleDemo and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click **Next** to move on.
5. In the next screen, specify the target framework you would like to use for your console application.
6. Click **Create** to complete the process.

You'll use this application in the subsequent sections of this article.

Install NuGet Package(s)

So far so good. The next step is to install the necessary NuGet Package(s). To install the required packages into your project, right-click on the solution and the select **Manage NuGet Packages for Solution…**. Now search for the package named BenchmarkDotNet in the search box and install it. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package BenchmarkDotNet
```

Create a Benchmarking Class

To create and execute benchmarks:

1. Create a Console application project in Visual Studio 2022.
2. Add the BenchmarkDotNet NuGet package to the project.
3. Create a class having one or more methods decorated with the Benchmark attribute.
4. Run your benchmark project in Release mode using the Run method of the BenchmarkRunner class.

A typical benchmark class contains one or more methods marked or decorated with the Benchmark attribute and, optionally, a method that's decorated with the GlobalSetup attribute, as shown in the code snippet given below:

```
public class MyBenchmarkDemo
{
   [GlobalSetup]
   public void GlobalSetup()
   {
       //Write your initialization code here
   }

   [Benchmark]
   public void MyFirstBenchmarkMethod()
   {
       //Write your code here
   }

   [Benchmark]
   public void MySecondBenchmarkMethod()
   {
       //Write your code here
   }
}
```

In BenchmarkDotNet, diagnosers are attached to the benchmarks to provide more useful information. The MemoryDiagnoser is a diagnoser that, when attached to your benchmarks, provides additional information, such as the allocated bytes and the frequency of garbage collection.

> Note that BenchmarkDotNet works only with Console applications. It won't support ASP.NET 6 or any other application types.

Here's how your benchmark class looks once you've added the MemoryDiagnoser attribute:

```
[MemoryDiagnoser]
public class MyBenchmarkDemo
{
   //Code removed for brevity
}
```

Setup and Cleanup

You might want to execute some code just once and you don't want to benchmark the code. As an example, you might want to initialize your database connection or create an HttpClient instance to be used by other methods decorated with the [Benchmark] attribute.

BenchmarkDotNet comes with a few attributes that can help you accomplish this. These attributes are [GlobalSetup], [GlobalCleanup], [IterationSetup], and [IterationCleanup].

You can take advantage of the GlobalSetup attribute to initialize an HttpClient instance, as shown in the code snippet given below:

```
private static HttpClient _httpClient;

[GlobalSetup]
public void GlobalSetup()
{
  var factory = new
  WebApplicationFactory<Startup>()
  .WithWebHostBuilder(configuration =>
   {
     configuration.ConfigureLogging
     (logging =>
     {
         logging.ClearProviders();
     });
         });

   _httpClient = factory.CreateClient();
}
```

Similarly, you can take advantage of the GlobalCleanup attribute to write your cleanup logic, as shown in the code snippet below:

```
[GlobalCleanup]
public void GlobalCleanup()
{
   //Write your cleanup logic here
}
```

Benchmarking LINQ Performance

Let's now examine how to benchmark LINQ methods. Create a new class named BenchmarkLINQPerformance in a file having the same name with the code shown in **Listing 1**. This is a simple class that benchmarks the performance of the Single and First methods of LINQ. Now that the benchmark class is ready, examine how to run the benchmark using BenchmarkRunner in the next section.

Execute the Benchmarks

As of this writing, you can use BenchmarkDotNet in a console application only. You can run benchmark on a specific type or configure it to run on a specific assembly. The following code snippet illustrates how you can trigger a benchmark on all types in the specified assembly:

```
var summary = BenchmarkRunner.Run
(typeof(Program).Assembly);
```

Listing 1: Benchmarking performance of LINQ

```
public class BenchmarkLINQPerformance
    {
        private readonly List<string>
        data = new List<string>();

        [GlobalSetup]
        public void GlobalSetup()
        {
            for(int i = 65; i < 90; i++)
            {
                char c = (char)i;
                data.Add(c.ToString());
            }
        }

        [Benchmark]
        public string Single() =>
        data.SingleOrDefault(x => x.Equals("M"));

        [Benchmark]
        public string First() =>
        data.FirstOrDefault(x => x.Equals("M"));
    }
```

You can use the following code snippet to run benchmarking on a specific type:

```
var summary = BenchmarkRunner.Run
<BenchmarkLINQPerformance>();
```

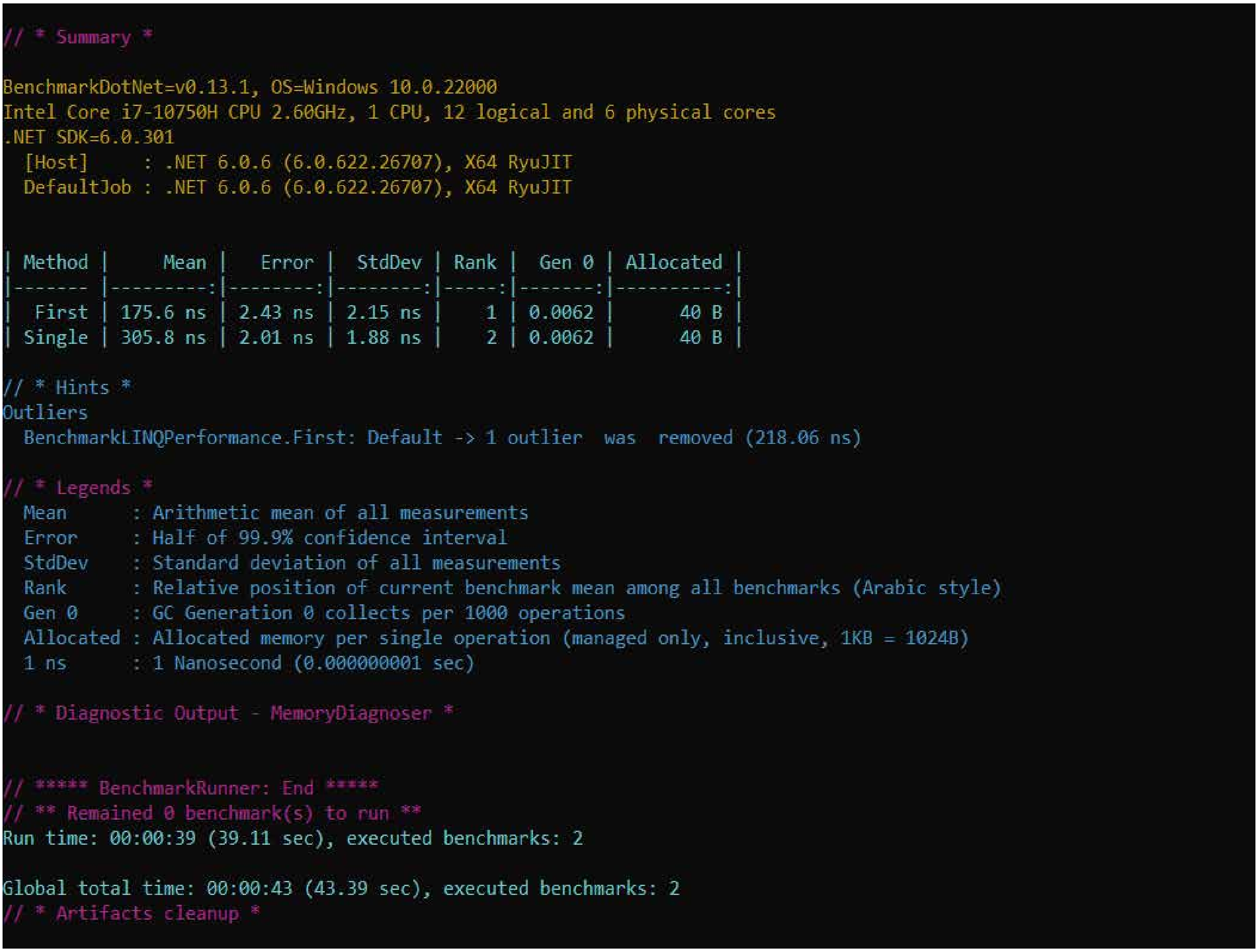
Or you can use:

```
var summary = BenchmarkRunner.Run
(typeof(BenchmarkLINQPerformance));
```

For the benchmark you created in the preceding section, you can use any of these statements in the Program class to execute the benchmark. **Figure 1** shows the results of the benchmark:

Interpreting the Benchmarking Results

As you can see in **Figure 6**, for each of the benchmarked methods, a row of the result data is generated. Because there are two benchmark methods called using three param values, there are six rows of benchmark result data. The benchmark results show the mean execution time, garbage collections (GCs), and the allocated memory.



```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=6.0.301
  [Host]     : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT
  DefaultJob : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT


| Method |     Mean |   Error |  StdDev | Rank |  Gen 0 | Allocated |
|------- |---------:|--------:|--------:|-----:|-------:|----------:|
|  First | 175.6 ns | 2.43 ns | 2.15 ns |    1 | 0.0062 |      40 B |
| Single | 305.8 ns | 2.01 ns | 1.88 ns |    2 | 0.0062 |      40 B |

// * Hints *
Outliers
  BenchmarkLINQPerformance.First: Default -> 1 outlier  was  removed (218.06 ns)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:39 (39.11 sec), executed benchmarks: 2

Global total time: 00:00:43 (43.39 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Figure 1: Benchmarking results of Single() vs First() methods

The Mean column shows the average execution time of both the methods. As is evident from the benchmark results, the First method is much faster than the Single method in LINQ. The Allocated column shows the managed memory allocated on execution of each of these methods. The Rank column shows the relative execution speeds of these methods ordered from fastest to slowest. Because there are two methods here, it shows 1 (fastest) and 2 (slowest) for the First and Single methods respectively.

Here's what each of the legends represent:

- **Method:** This column specifies the name of the method that has been benchmarked.
- **Mean:** This column specifies the average time or the arithmetic mean of the measurements made on execution of the method being benchmarked.
- **StdDev:** This column specifies the standard deviation, i.e., the extent to which the execution time deviated from the mean time.
- **Gen 0:** This column specifies the Gen 0 collections made for each set of 1000 operations.
- **Gen 1:** This column specifies the Gen 1 collections made for each set of 1000 operations.
- **Gen 2:** This column specifies the Gen 2 collections made for each set of 1000 operations. (Note that here, Gen 2 isn't shown because there were no Gen 2 collections in this example.)
- **Allocated:** This column specifies the managed memory allocated for a single operation.

Benchmarking StringBuilder Performance

Let's now examine how you can benchmark the performance of the StringBuilder class in .NET. Create a new class named BenchmarkStringBuilderPerformance with the code in **Listing 2**.

Now, write the two methods for benchmarking performance of StringBuilder with and without using StringBuilderCache, as shown in **Listing 3**. The complete source code of the BenchmarkStringBuilderPerformance class is given in **Listing 4**.

Executing the Benchmarks

Write the following piece of code in the Program.cs file of the BenchmarkingConsoleDemo console application project to run the benchmarks:

```
using BenchmarkingConsoleDemo;
using System.Runtime.InteropServices;
class Program
{
 static void Main(string[] args)
 {
  BenchmarkRunner.Run
  <BenchmarkStringBuilderPerformance>();
 }
}
```

Listing 2: Benchmarking performance of StringBuilder and StringBuildercache

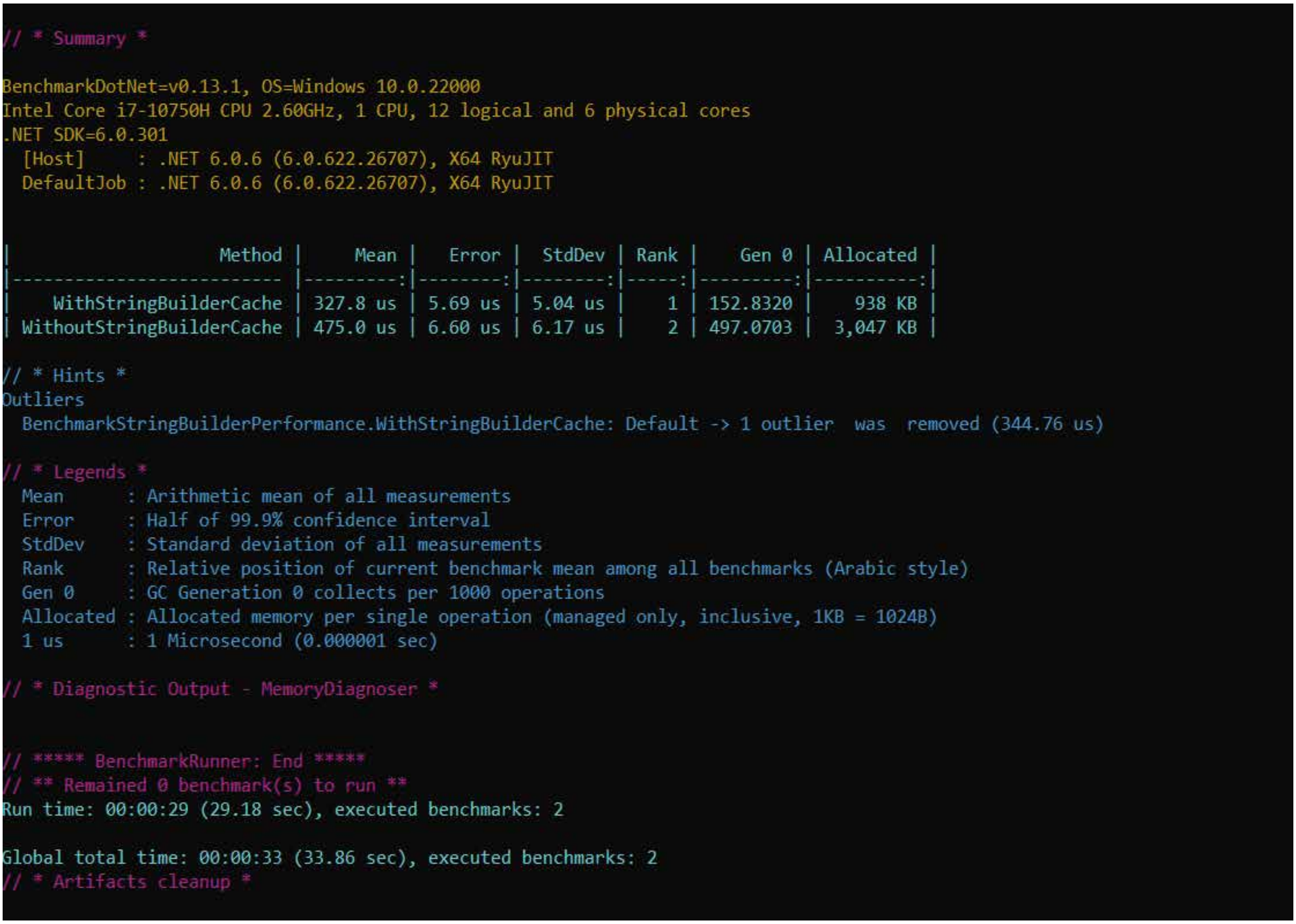
```
[MemoryDiagnoser]
[Orderer(BenchmarkDotNet.Order.
SummaryOrderPolicy.FastestToSlowest)]
[RankColumn]
public class
BenchmarkStringBuilderPerformance
{
    const string message =
   "Some text for testing purposes only.";
    const int CTR = 10000;
}
```

Listing 3: Continued from Listing 2

```
[Benchmark]
public void WithoutStringBuilderCache()
{
   for (int i = 0; i < CTR; i++)
   {
       var stringBuilder =
       new StringBuilder();
       stringBuilder.Append(message);
       _ = stringBuilder.ToString();
   }
}

[Benchmark]
public void WithStringBuilderCache()
{
    for (int i = 0; i < CTR; i++)
    {
       var stringBuilder =
       StringBuilderCache.Acquire();
       stringBuilder.Append(message);
       _= StringBuilderCache.
       GetStringAndRelease(stringBuilder);
    }
}
```

Listing 4: Benchmarking performance of StringBuilderCache

```
[MemoryDiagnoser]
[Orderer(BenchmarkDotNet.Order.
SummaryOrderPolicy.FastestToSlowest)]
[RankColumn]
public class
BenchmarkStringBuilderPerformance
{
    const string message =
    "Some text for testing purposes only.";
    const int CTR = 10000;

    [Benchmark]
    public void WithoutStringBuilderCache()
    {
        for (int i = 0; i < CTR; i++)
        {
            var stringBuilder =
            new StringBuilder();
            stringBuilder.Append(message);
            _ = stringBuilder.ToString();
        }
    }

    [Benchmark]
    public void WithStringBuilderCache()
    {
        for (int i = 0; i < CTR; i++)
        {
           var stringBuilder =
           StringBuilderCache.Acquire();
           stringBuilder.Append(message);
           _= StringBuilderCache.
           GetStringAndRelease(stringBuilder);
        }
     }
}
```



```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=6.0.301
  [Host]     : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT
  DefaultJob : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT


|                    Method |     Mean |    Error |   StdDev | Rank |    Gen 0 | Allocated |
|-------------------------- |---------:|---------:|---------:|-----:|---------:|----------:|
|    WithStringBuilderCache | 327.8 us | 5.69 us | 5.04 us |    1 | 152.8320 |    938 KB |
| WithoutStringBuilderCache | 475.0 us | 6.60 us | 6.17 us |    2 | 497.0703 |  3,047 KB |

// * Hints *
Outliers
  BenchmarkStringBuilderPerformance.WithStringBuilderCache: Default -> 1 outlier  was  removed (344.76 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:29 (29.18 sec), executed benchmarks: 2

Global total time: 00:00:33 (33.86 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Figure 2: Benchmarking StringBuilder performance

To execute the benchmarks, set the compile mode of the project to Release and run the following command in the same folder where your project file resides:

```
dotnet run -p
BenchmarkingConsoleDemo.csproj -c Release
```

Figure 2 shows the result of the execution of the benchmarks.

The following code snippet illustrates how you can mark the WithStringBuilderCache benchmark method as a baseline method.

```
[Benchmark (Baseline = true)]
public void WithStringBuilderCache()
{
   for (int i = 0; i < CTR; i++)
   {
     var stringBuilder =
     StringBuilderCache.Acquire();
     stringBuilder.Append(message);
     _= StringBuilderCache.
     GetStringAndRelease(stringBuilder);
    }
}
```

StringBuilderCache is an internal class that represents a per-thread cache with three static methods: Acquire, Release, and GetStringAndRelease. Here's the complete source code of this class: shorturl.at/dintW.

The Acquire method can acquire a StringBuilder instance. The Release method can store the StringBuilder instance in the cache if the instance size is within the maximum allowed size. The GetStringAndRelease method is used to return a string instance and return the StringBuilder instance to the cache.

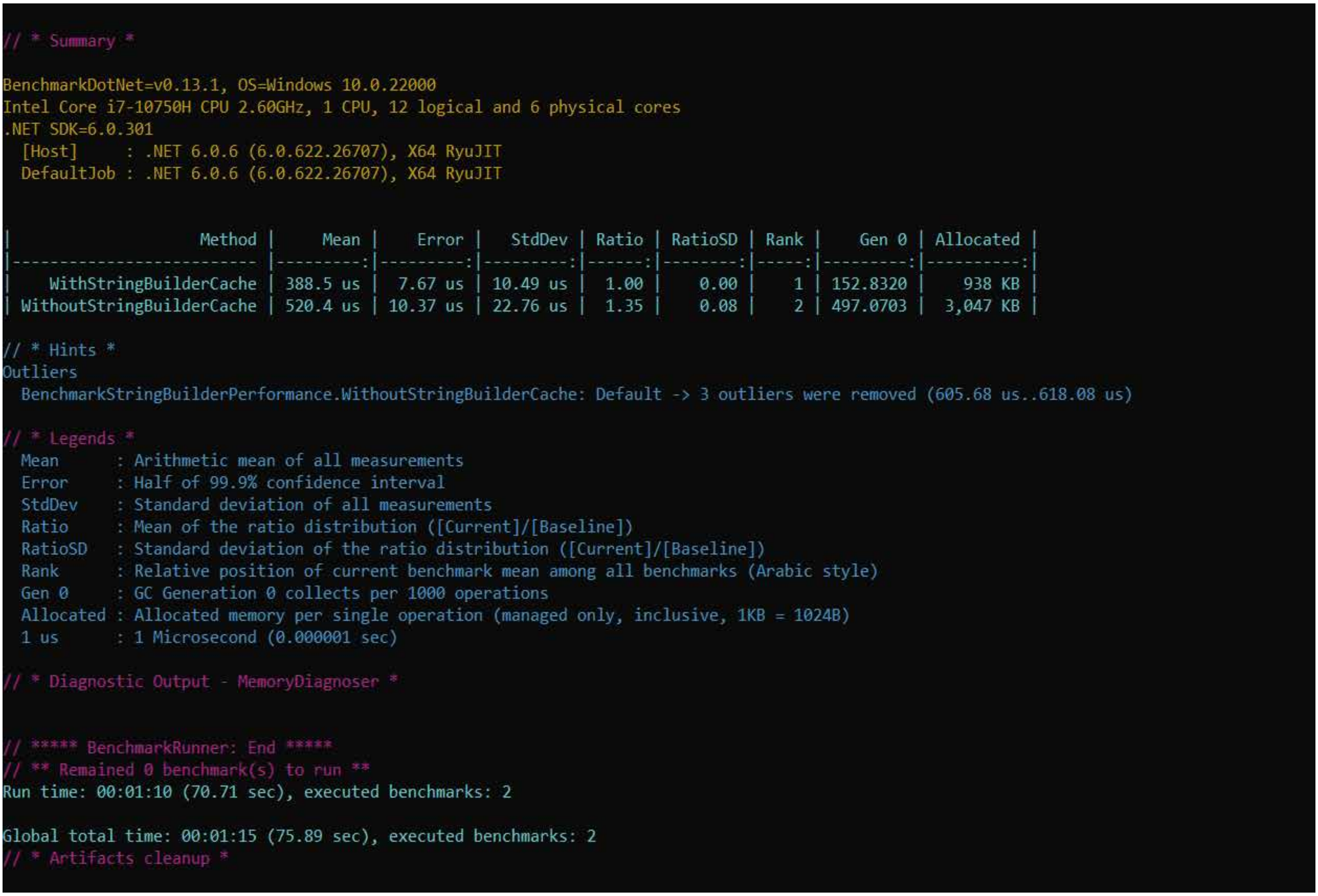
When you run the benchmarks this time, the output will be similar to **Figure 3**.

Benchmarking ASP.NET 6 Applications

In this section, you'll examine how to benchmark ASP.NET 6 applications to retrieve performance data.

Create a New ASP.NET 6 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.



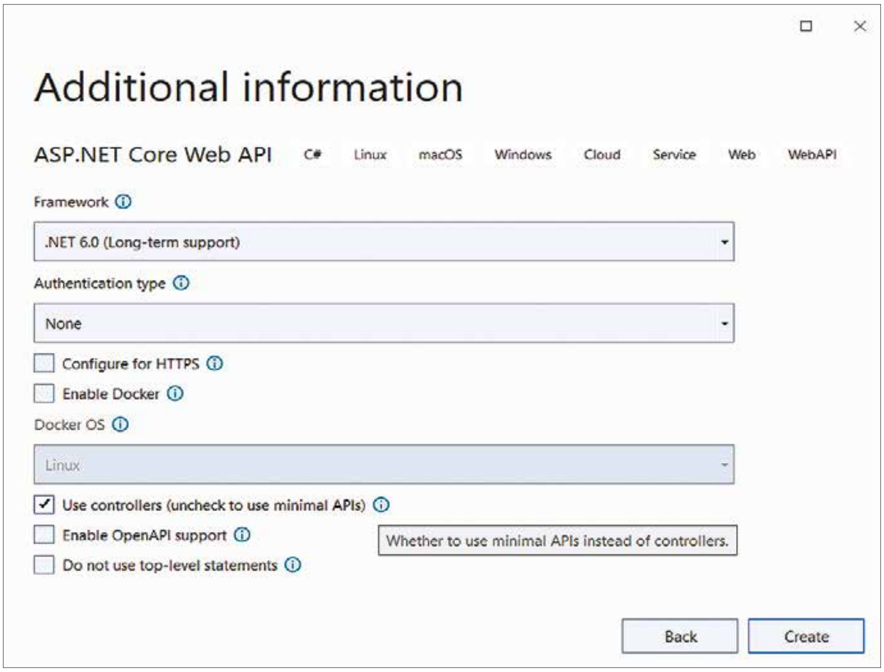
```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=6.0.301
  [Host]     : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT
  DefaultJob : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT


|                    Method |     Mean |     Error |    StdDev | Ratio | RatioSD | Rank |    Gen 0 | Allocated |
|-------------------------- |---------:|----------:|----------:|------:|--------:|-----:|---------:|----------:|
|    WithStringBuilderCache | 388.5 us |   7.67 us |  10.49 us |  1.00 |    0.00 |    1 | 152.8320 |    938 KB |
| WithoutStringBuilderCache | 520.4 us |  10.37 us |  22.76 us |  1.35 |    0.08 |    2 | 497.0703 |  3,047 KB |

// * Hints *
Outliers
  BenchmarkStringBuilderPerformance.WithoutStringBuilderCache: Default -> 3 outliers were removed (605.68 us..618.08 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
  RatioSD   : Standard deviation of the ratio distribution ([Current]/[Baseline])
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:01:10 (70.71 sec), executed benchmarks: 2

Global total time: 00:01:15 (75.89 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Figure 3: The performance of benchmark methods with one of them set as a baseline method

To create a new ASP.NET 6 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the "Create a new project" window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name as BenchmarkingWebDemo and the path where it should be created in the "Configure your new project" window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the "Place solution and project in the same directory" checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," and the "Enable OpenAPI support" checkboxes are unchecked because you won't use any of these in this example.
6. Because you'll be using minimal APIs in this example, remember to uncheck the Use controllers (uncheck to use minimal APIs) checkbox, as shown in **Figure 4**.
7. Click Create to complete the process.

Minimal API is a new feature added in .NET 6 that enables you to create APIs with minimal dependencies. You'll use this application in this article. Let's now get



Additional information

ASP.NET Core Web API

Framework: .NET 6.0 (Long-term support)

Authentication type: None

☐ Configure for HTTPS

☐ Enable Docker

Docker OS: Linux

☑ Use controllers (uncheck to use minimal APIs)

☐ Enable OpenAPI support

☐ Do not use top-level statements

Whether to use minimal APIs instead of controllers.

Figure 4: Enable minimal APIs for your Web API

started benchmarking ASP.NET applications with a simple method.

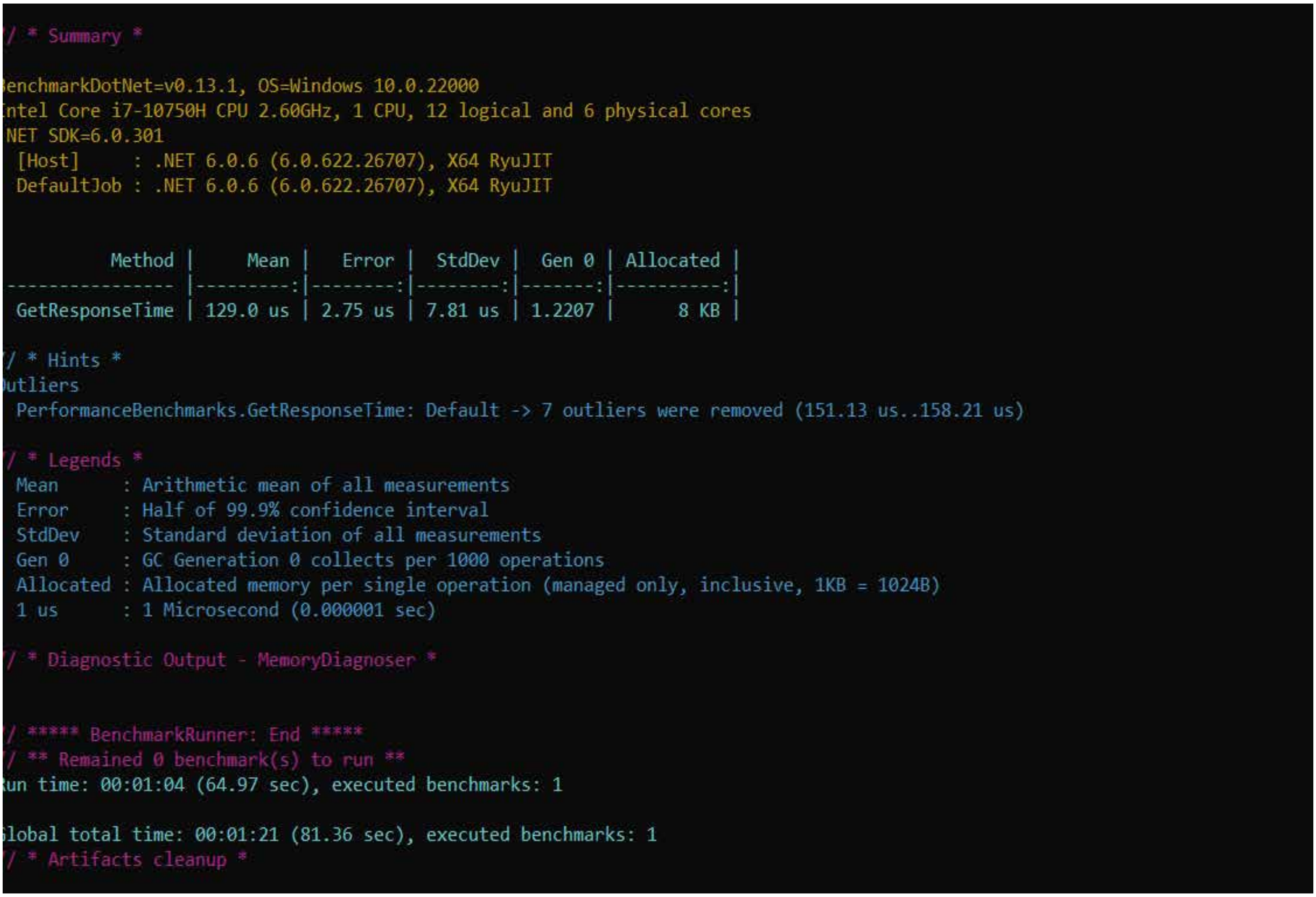
Get the Response Time in ASP.NET 6

You can easily get the response time of an endpoint using BenchmarkDotNet. To execute the ASP.NET 6 endpoints, you can use the HttpClient class. To create an instance of HttpClient, you can use the WebApplicationFactory, as shown in the code snippet given below:

```
var factory = new WebApplicationFactory
<Startup>()
    .WithWebHostBuilder(configuration =>
    {
        configuration.ConfigureLogging
        (logging =>
        {
            logging.ClearProviders();
        });
```

Listing 5: Benchmarking response time of an API

```
public class BenchmarkAPIPerformance
{
 private static HttpClient _httpClient;

 [GlobalSetup]
 public void GlobalSetup()
 {
     var factory = new WebApplicationFactory
                <Startup>()
      .WithWebHostBuilder(configuration =>
       {
          configuration.
          ConfigureLogging(logging =>
          {
             logging.ClearProviders();
          });
       });

      _httpClient =
      factory.CreateClient();
   }

   [Benchmark]
   public async Task GetResponseTime()
   {
       var response =
       await _httpClient.GetAsync("/");
   }
}
```



```
/ * Summary *

enchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
ntel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
NET SDK=6.0.301
 [Host]     : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT
 DefaultJob : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT


          Method |     Mean |    Error |   StdDev |  Gen 0 | Allocated |
---------------- |---------:|--------:|--------:|-------:|----------:|
 GetResponseTime | 129.0 us | 2.75 us | 7.81 us | 1.2207 |      8 KB |

/ * Hints *
utliers
 PerformanceBenchmarks.GetResponseTime: Default -> 7 outliers were removed (151.13 us..158.21 us)

/ * Legends *
 Mean      : Arithmetic mean of all measurements
 Error     : Half of 99.9% confidence interval
 StdDev    : Standard deviation of all measurements
 Gen 0     : GC Generation 0 collects per 1000 operations
 Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
 1 us      : 1 Microsecond (0.000001 sec)

/ * Diagnostic Output - MemoryDiagnoser *


/ ***** BenchmarkRunner: End *****
/ ** Remained 0 benchmark(s) to run **
un time: 00:01:04 (64.97 sec), executed benchmarks: 1

lobal total time: 00:01:21 (81.36 sec), executed benchmarks: 1
/ * Artifacts cleanup *
```

Figure 5: Benchmarking results of the response time of an API endpoint

```
    });
_httpClient = factory.CreateClient();
```

To benchmark the response time of an endpoint, you can use the following code:

```
[Benchmark]
   public async Task GetResponseTime()
   {
       var response =
       await _httpClient.GetAsync("/");
   }
```

The complete source code is given in **Listing 5** for your reference. The benchmark results are shown in **Figure 5**.

Real-World Use Case of BenchmarkDotNet

In this section, you'll examine how to take advantage of BenchmarkDotNet to measure the performance of an application, determine the slow running paths, and take necessary steps to improve the performance. You'll use an entity class named Product that contains a Guid field named **Id**. Note that a call to Guid.NewGuid consumes resources and is slow.

If you replace the Guid property with an int property, it consumes significantly fewer resources and improves performance. You'll create an optimized version of the Product class and then benchmark the performance of both these classes.

Create the Entity Classes

In the Solution Explorer Window, right-click on the project and create a new file named Product with the following code in there:

```
public class Product
{
   public Guid Id { get; set; }
   public string Name { get; set; }
   public string Category { get; set; }
   public decimal Price { get; set; }
}
```

Let's create another entity class named ProductOptimized, which is a replica of the Product class but optimized for improving performance. The following code snippet illustrates the ProductOptimized class:

```
public struct ProductOptimized
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Category { get; set; }
        public decimal Price { get; set; }
    }
```

In the ProductOptimized class, you've changed the data type of the ID and the Category properties of the Product class with integers.

Create the Product Repository

Create a new class named ProductRepository in a file having the same name with a .cs extension. Now write the following code in there:

```
public class
ProductRepository :
IProductRepository
    {

    }
```

The ProductRepository class illustrated in the code snippet below, implements the methods of the IProductRepository interface. Here is how this interface should look:

```
public interface IProductRepository
{
    public Task<List<Product>> GetAllProducts();
    public Task<List<ProductOptimized>>
    GetAllProductsOptimized();
}
```

The ProductRepository class implements the two methods of the IProductRepository interface:

```
public Task<List<Product>>
GetAllProducts()
```

Listing 6: The GetProducts and GetProductsOptimized methods

```
private List<Product> GetProductsInternal()
{
   List<Product> products =
   new List<Product>();

   for(int i=0; i<1000;i++)
   {
        Product product = new Product
        {
           Id = Guid.NewGuid(),
           Name = "Lenovo Legion",
           Category = "Laptop",
           Price = 3500
        };
   }
   return products;
}

private List<ProductOptimized>
GetProductsOptimizedInternal()
{
  List<ProductOptimized> products = new
           List<ProductOptimized>(1000);

  for (int i = 0; i < 1000; i++)
  {
      ProductOptimized product =
      new ProductOptimized
      {
          Id = i,
          Name = "Lenovo Legion",
          Category = 1,
          Price = 3500
       };
  }
   return products;
}
```

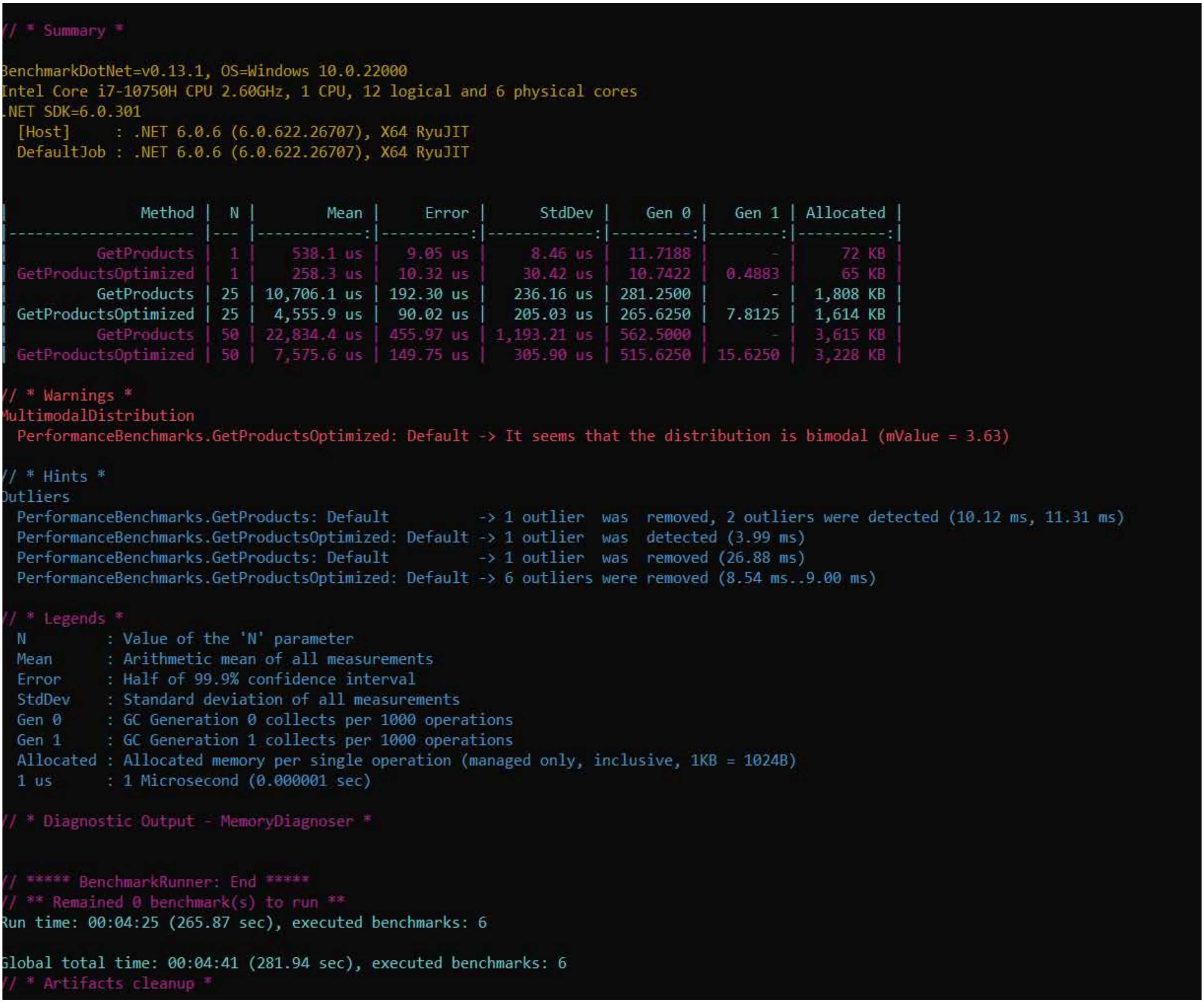
```
{
    return Task.FromResult
    (GetProductsInternal());
}

public Task<List<ProductOptimized>>
GetAllProductsOptimized()
{
    return Task.FromResult
    (GetProductsOptimizedInternal());
}
```

Although the GetAllProducts method returns a list of the Product class, the GetAllProductsOptimized method returns a list of the ProductOptimized class you created earlier. These two methods call the private methods named GetProductsInternal and GetProductsOptimizedInternal respectively. These private methods return a List of Product and ProductOptimized class respectively.

The GetProductsInternal method creates a List of the Product class. It uses the Guid.NewGuid method to generate new Guids for the ID field. Hence, it creates 1000 new Guids, one for each instance of the Product class. Contrarily, the GetProductsOptimizedInternal method creates a List of the ProductOptimized class. In this class, the ID property is an integer type. So, in this method, 1000 new integer IDs are created. Create new Guids is resource intensive and much slower than creating an integer.

Note that this implementation has been made as simple as possible because my focus is on how you can benchmark the performance of these methods.

The source code given in **Listing 6** illustrates the GetProductsInternal and GetProductsOptimizedInternal methods. Note that in the GetProductsOptimizedInternal method, a list of ProductOptimized entity class is created and the size of the list has been specified as well.



```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=6.0.301
  [Host]     : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT
  DefaultJob : .NET 6.0.6 (6.0.622.26707), X64 RyuJIT


|               Method |  N |        Mean |      Error |      StdDev |     Gen 0 |   Gen 1 | Allocated |
|--------------------- |--- |------------:|-----------:|------------:|----------:|--------:|----------:|
|          GetProducts |  1 |    538.1 us |    9.05 us |     8.46 us |   11.7188 |       - |     72 KB |
| GetProductsOptimized |  1 |    258.3 us |   10.32 us |    30.42 us |   10.7422 |  0.4883 |     65 KB |
|          GetProducts | 25 | 10,706.1 us |  192.30 us |   236.16 us |  281.2500 |       - |  1,808 KB |
| GetProductsOptimized | 25 |  4,555.9 us |   90.02 us |   205.03 us |  265.6250 |  7.8125 |  1,614 KB |
|          GetProducts | 50 | 22,834.4 us |  455.97 us | 1,193.21 us |  562.5000 |       - |  3,615 KB |
| GetProductsOptimized | 50 |  7,575.6 us |  149.75 us |   305.90 us |  515.6250 | 15.6250 |  3,228 KB |

// * Warnings *
MultimodalDistribution
  PerformanceBenchmarks.GetProductsOptimized: Default -> It seems that the distribution is bimodal (mValue = 3.63)

// * Hints *
Outliers
  PerformanceBenchmarks.GetProducts: Default          -> 1 outlier  was  removed, 2 outliers were detected (10.12 ms, 11.31 ms)
  PerformanceBenchmarks.GetProductsOptimized: Default -> 1 outlier  was  detected (3.99 ms)
  PerformanceBenchmarks.GetProducts: Default          -> 1 outlier  was  removed (26.88 ms)
  PerformanceBenchmarks.GetProductsOptimized: Default -> 6 outliers were removed (8.54 ms..9.00 ms)

// * Legends *
  N         : Value of the 'N' parameter
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen 0     : GC Generation 0 collects per 1000 operations
  Gen 1     : GC Generation 1 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:04:25 (265.87 sec), executed benchmarks: 6

Global total time: 00:04:41 (281.94 sec), executed benchmarks: 6
// * Artifacts cleanup *
```

Figure 6: The benchmarking results of the GetProducts and GetProductsOptimized methods

Listing 7: Benchmarking performance of GetProducts and GetProductsOptimized API methods

```
[MemoryDiagnoser]
public class BenchmarkAPIPerformance
{
  private static HttpClient _httpClient;

  [Params(1, 25, 50)]
  public int N;

  [GlobalSetup]
  public void GlobalSetup()
  {
     var factory =
     new WebApplicationFactory<Startup>()
         .WithWebHostBuilder(configuration =>
     {
         configuration.
         ConfigureLogging(logging =>
         {
               logging.ClearProviders();
              });
         });
      _httpClient = factory.CreateClient();
   }

  [Benchmark]
  public async Task GetProducts()
  {
     for(int i = 0;i < N; i++)
     {
        var response =
        await _httpClient.
        GetAsync("/GetProducts");
     }
  }

  [Benchmark]
  public async Task GetProductsOptimized()
  {
      for (int i = 0; i < N; i++)
      {
         var response =
         await _httpClient.
         GetAsync("/GetProductsOptimized");
      }
   }
}
```

Create the Endpoints

You'll create two endpoints, GetProducts and GetProductsOptimized, and then benchmark them. Because you're using minimal API in this example, write the following code snippet in the Program class of your ASP.NET 6 Web API project to create the two endpoints:

```
app.MapGet("/GetProducts", async
(IProductRepository productRepository) =>
{
    return Results.Ok(await
productRepository.GetAllProducts());
});

app.MapGet("/GetProductsOptimized", async
(IProductRepository productRepository) =>
{
    return Results.Ok(await
productRepository.GetAllProductsOptimized());
});
```

Create the Benchmarks

Let's now create the benchmarking class that contains the methods to be benchmarked using BenchmarkDotNet. To do this, create a class named BenchmarkManager in a file with the same name and a .cs extension and write the code shown in **Listing 7** in there.

The two methods that need to be benchmarked are the GetProducts and GetProductsOptimized methods. Note the Benchmark attribute on each of these methods. These two methods use the HttpClient instance to execute the two endpoints GetProducts and GetProductsOptimized respectively.

Figure 6 shows the output of the execution of benchmarks. As you can see, the GetProductsOptimized consumes less memory and is much faster than its counterpart, i.e., the GetProducts method.

Conclusion

BenchmarkDotNet is a compelling and easy-to-use framework to benchmark .NET code. You can execute a benchmark on a single method, module, or entire application to check the performance of the code without affecting its functionality. Remember that to improve the performance and scalability of your application, you must adhere to the best practices, if not, merely benchmarking your application's code won't help.

Joydip Kanjilal

ONLINE QUICK ID 2209071

Event Sourcing and CQRS with Marten

Jeremy D. Miller
jeremydmiller@yahoo.com
www.jeremydmiller.com
@jeremydmiller

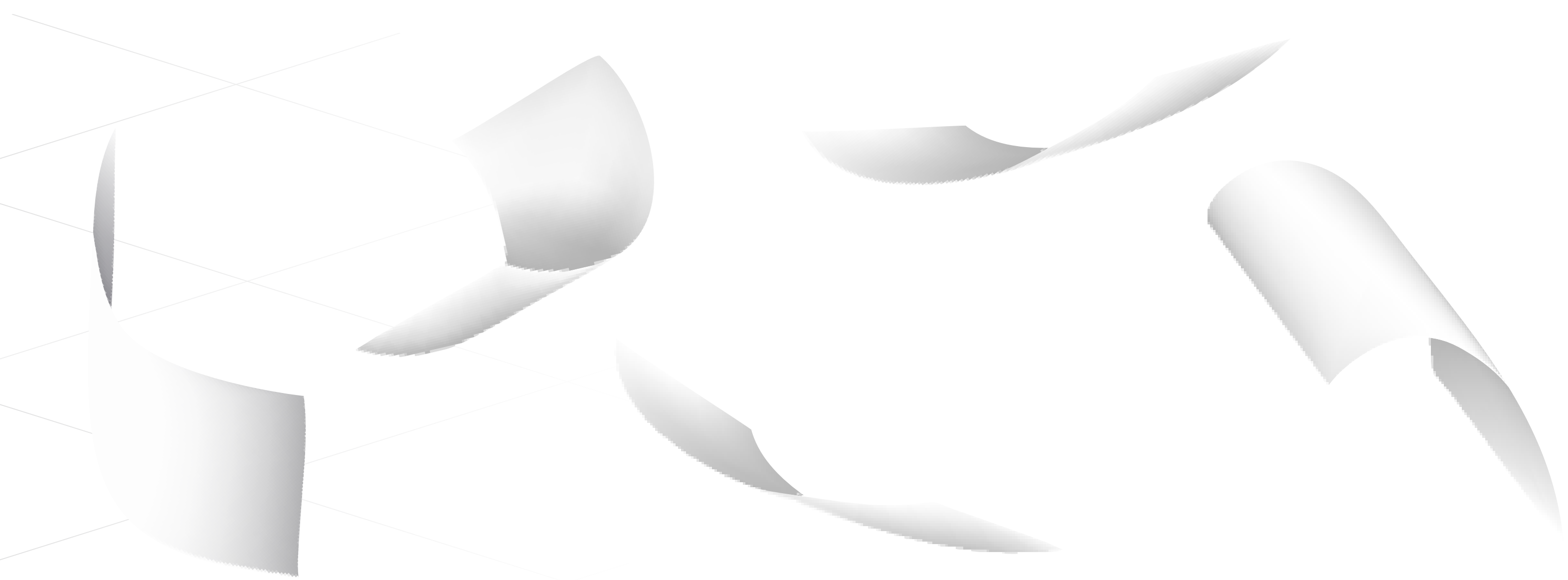
Jeremy Miller is the Senior Director of Software Architecture at MedeAnalytics. Jeremy began his software career writing "Shadow IT" applications to automate his tedious engineering documentation, then wandered into software development because it looked like more fun. Jeremy is heavily involved in open source .NET development as the lead developer of Marten, Lamar, Alba, and other projects in the JasperFx family. Jeremy occasionally manages to write about various software topics at http://jeremydmiller.com.

In this article, I'm going to examine the usage of Event Sourcing and the Command Query Responsibility Segregation (CQRS) architectural style through a sample telehealth medicine application. For tooling, I'm going to use the Marten library (https://martendb.io), which provides robust support for Event Sourcing on top of the Postgresql database engine. Before I move on to Event Sourcing though, let's think about the typical role of a database within your systems that don't use Event Sourcing. For most of my early career as a software professional, I assumed that system state would be persisted in a relational database that would act as the source of truth for the system. Very frequently, I've visualized systems with something like the simple layered view shown in **Figure 1**.

With this typical architecture, almost all input and output of the system will involve reading or writing to this one database. Different use cases will have different needs, so at various times I might need to write explicit code to:

- Map the middle tier model to the database tables
- Map incoming input from outside the system to the database tables
- Translate the internal database to different structures for outgoing data in Web services

The point I'm trying to make here is that the single database model can easily bring with it a fair amount of mechanical



effort in translating the one database structure to the specific needs of various system input or output—and I'll need to weigh this effort when I compare the one database model to the Event Sourcing and CQRS approach I'll examine later.

I've also long known that in the industry, no one database structure can be optimized for both reading and writing, so I might very well try to support a separate, denormalized database specifically for reporting. That reporting database will need to be synchronized somehow from the main transactional database. This is just to say that the idea of having essentially the same information in multiple databases within the software architectures is not exactly new.

Alternative approaches using Event Sourcing or CQRS can look scary or intimidating upon your first exposure. Myself, I walked out after a very early software conference presentation in 2008 on what later became known as CQRS shaking my head and thinking that it was completely crazy and would never catch on, yet here I am, recommending this pattern for some systems.

Telehealth System Example

Before diving into the nomenclature or concepts around Event Sourcing or CQRS architectures, I want to consider a sample problem domain. Hastened by the COVID pandemic, "telehealth" approaches to health care, where you can speak to a health care provider online without having to make an in-office visit, rapidly became widespread. Imagine that I'm tasked with building a new website application that allows potential patients to request an appointment, connect with a provider (physician, nurse, nurse practitioner, etc.), and host the online appointments.

The system will need to provide functionality to coordinate the on-call providers. In this case, I'm going to attempt to schedule the appointments as soon as a suitable provider is available, so I'll need to be able to estimate expected wait times. I do care about patient happiness and want the providers working with the system and to have a good experience with the system, so it's going to be important to be able to collect a lot of metrics to help adjust staffing. Moreover, I need to plan for problems during a normal business day and give the administrative users the ability to understand what transpired during the day that might have caused patient wait times to escalate.

Moreover, because this is related to health care, I should plan on having some pretty stringent requirements for auditing all activity within the system through a formal audit log.

> This sample problem domain is based on a project I was a part of where I successfully used Event Sourcing, but on a very different technical stack.

Event Sourcing

Event Sourcing is an architectural approach to data persistence that captures all incoming data as an append-only event log. In effect, you're making the logical change log into the single source of truth in your system. Rather than modeling the current state of the system and trying to map every incoming input and outgoing query to this centralized state, event sourcing explicitly models the changes to the system.

So how does it work? First, let's do some modeling of the online telehealth system. Just considering events around the online appointments I might model events for:

- Appointment Requested
- Appointment Scheduled

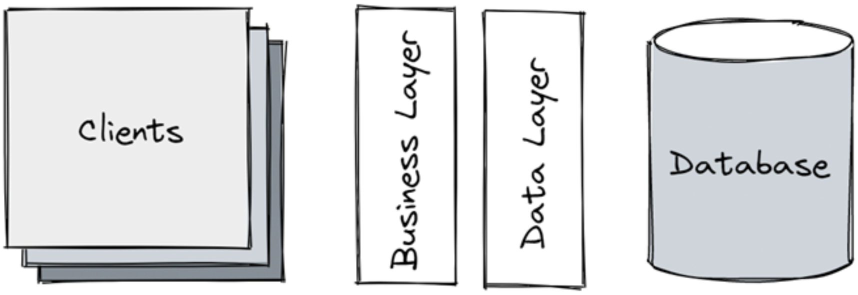


Figure 1: Traditional layered architecture

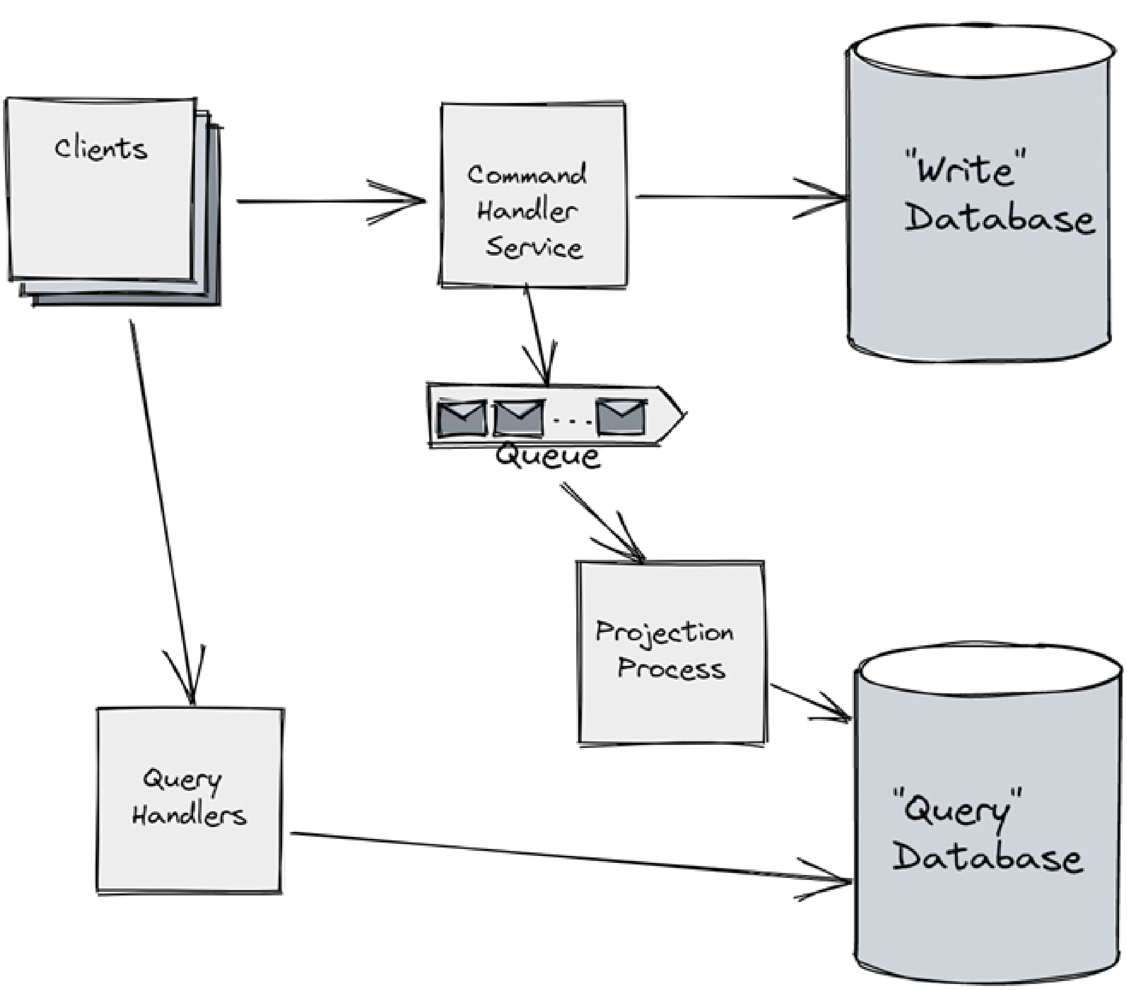


Figure 2: Scary, complicated CQRS architecture

- Appointment Started
- Appointment Finished
- Appointment Cancelled

These events are small types carrying data that models the change of state whenever they're recorded. Do note that the event type names are expressed in the past tense and are directly related to the business domain. The event name by itself can be important in understanding the system behavior. As an example, here's a C# version of **AppointmentScheduled** that models whether the appointment is assigned to a certain provider (medical professional):

```
public record AppointmentScheduled(
    Guid ProviderId,
    DateTimeOffset EstimatedTime
);
```

Taking Marten as a relatively typical approach, these event objects are persisted in the underlying database as serialized JSON in a single table that's ordered sequentially by the order in which the events are appended. Like other event stores, Marten also tracks metadata about the events, like the time the event was captured, and potentially more data related to distributed tracing, like correlation identifiers.

The events are organized into streams of related events that model a single workflow within the system. In the telehealth system, there are event streams for:

- Appointments
- Provider Shift to model the activity of a single provider during a single day

Although events in an Event Sourcing approach are the source of truth, you do still need to understand the current system state to support incoming system commands or supply clients with queries against the system state. This is where the concept of a projection comes into play. A projection is a view of the underlying events suitable for providing the **write model** to validate or apply incoming commands or a **read model** that's suitable for usage by system queries. If you're familiar with the concept of materialized views in relational database engines, a projection in a system based on Event Sourcing plays a very similar role.

The advantages, or maybe just the applicability, of Event Sourcing are:

- It creates a rich audit log of business activity.
- It supports the concept of "Time Travel" or temporal querying to be able to analyze the state of the system in the past by selectively replaying events.

- Using Event Sourcing makes it possible to retrofit potentially valuable metrics about the system after the fact by again replaying the events.
- Event Sourcing fits well with asynchronous programming models and event-driven architectures.
- Having the event log can often clarify system behavior.

Command Query Responsibility Segregation

CQRS is an architectural pattern that calls for the system state to be divided into separate models. The **write** model is optimized for transactions and is updated by incoming commands. The **read** model is optimized for queries to the system. As you'll rightly surmise, ***something*** has to synchronize and transform the incoming **write** model to the outgoing **read** model. That leads to the architectural diagram in **Figure 2**, which I think of as the "scary view of CQRS."

In this common usage of CQRS, there's some kind of background process that's asynchronously and continuously applying data updates to the **write** model to update the **read** model. This can lead to extra complexity through more necessary infrastructure compared to the classic "one database model" system model. I don't believe this is necessarily more code than using the traditional one database model. Rather, I would say that the hidden mapping and translation code in the one database model is much more apparent in the CQRS approach.

Event Sourcing and CQRS can be used independently of each other, but you'll very frequently see these two techniques used together. Fortunately, as I'll show in the remainder of this article, Marten can help you create a simpler architecture for CQRS with Event Sourcing than the diagram above.

Requirements through Event Storming

Event Storming (https://www.eventstorming.com/) is a very effective requirements workshop format to help a development team and their collaborating business partners understand the requirements for a software system. As the name suggests, Event Storming is a natural fit with Event Sourcing (and CQRS architectures).

Although there are software tools to do Event Storming sessions online, the easiest way to get started with Event Storming is to grab some colored sticky notes, a couple of markers, and convene a session with both the development team and the business domain experts near a big whiteboard or a blank wall.

The first step is to start brainstorming on the domain events within the business processes. As you discover these logical events, you'll write the event name down on an orange card and stick it on the board. As an example from the telehealth problem domain, some events might be "Appointment Requested" or "Appointment Scheduled" or "Appointment Cancelled." Note that these events are named tersely and are expressed in the past tense. As much as possible, you want to try to organize the events in the sequential order in which they occur within the system. If using a whiteboard, I also like to add some ad hoc arrows to delineate possible branching or relationships, but that's not part of the formal Event Storming approach.

The next step—but don't think for a minute that this must be a linear flow and that you shouldn't iterate between steps at any time—is to identify the commands or input to the system that will cause the previously identified events in the system. These commands are recorded as blue notes just to the left of the event or events that the command may cause in the system. The nomenclature, in this case, is in the present tense, like "Request Appointment."

In the third step, try to identify the business entities you'll need in order to process the incoming command inputs and decide which events should be raised. In Event Storming (and Event Sourcing) nomenclature, these are referred to as "Aggregates." In the case of the telehealth, I've identified the need to have an Appointment aggregate that reflects the current state of an ongoing or requested patient appointment and a "Provider Shift" to track the time and activity of a provider during a particular day. These aggregates are captured in yellow cards and posted to the board.

Beyond that, you can optionally use:

- Green cards to denote informational views that users of the system need to access to carry out their work. In the case of the telehealth system, I'm calling out the need for a Board view that represents a related group of appointments and providers during a single workday. For example, pediatric appointments in the state of Texas on July 18, 2022 are a single Board.
- Significant business logic processes that potentially create one or more domain events are recorded in purple notes. In the telehealth example, there's going to be some kind of "matching logic" that tries to match appropriate providers with the incoming appointments based on a combination of availability, specialty, and the licensure of the provider.
- External system dependencies can be written down on pink cards to record their existence. In this case, I'll probably use Twilio, or something similar, to host any kind of embedded chat or teleconferencing, so I'm noting that in the Event Storming session.

Figure 3 shows a sample for what an Event Storming session on the telehealth system might look like.

I'm a big fan of Event Storming to discover requirements and to create a common understanding of the business domain. Event Storming stands apart from many traditional requirements elicitation techniques by directly pointing the way toward the artifacts in your code. Event Storming sessions are a great way to discover the ubiquitous language for the system that is a necessary element of doing domain-driven development (DDD).

Getting Started with Marten

To get started with Marten as your event store, you'll first need a Postgresql database. My preference for local development is to use Docker to run the development database, and this is a copy of a **docker-compose.yaml** file that will get you started:

```
version: '3'
services:
  postgresql:
    image: "clkao/postgres-plv8:latest"
```

Keep Your Streams Short

Although it's technically possible and maybe a little tempting to just throw all your events into one logical stream, you're more likely to be successful by dividing the event store into shorter streams.

Marten can be vulnerable to concurrent access problems if appending simultaneously to the same stream. Separating the event store into smaller streams avoids that issue.

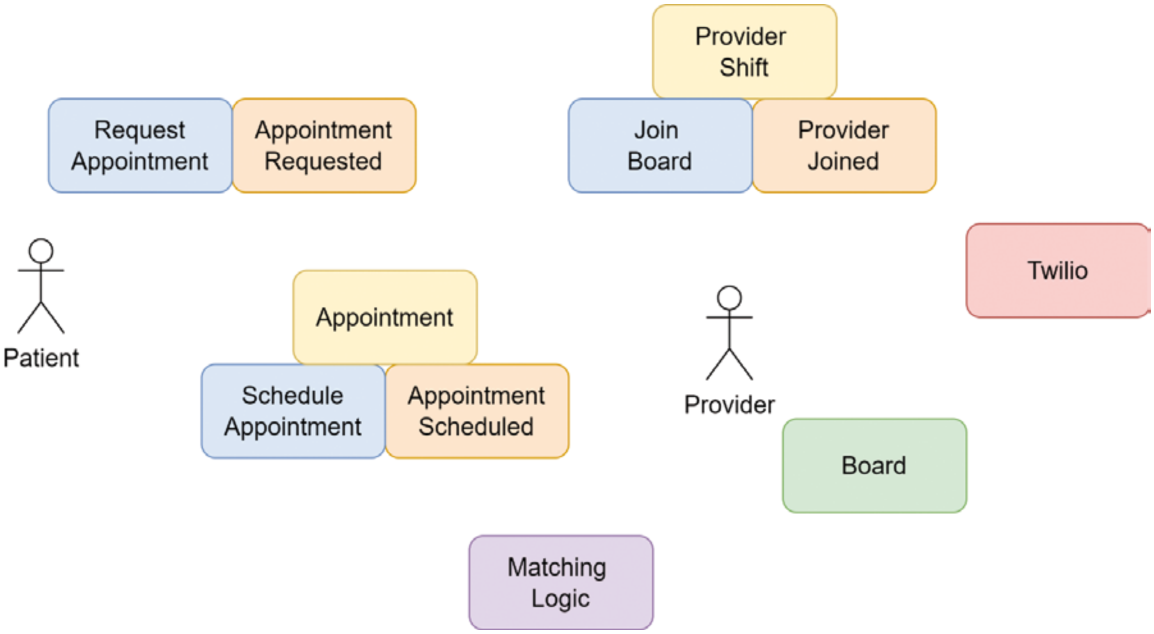


Figure 3: Event Storming sample

```
    ports:
     - "5433:5432"
```

Assuming that you have Docker Desktop installed on your local development computer, you just need to type this in your command line at the same location as the file above:

```
docker compose up -d
```

The command above starts the Docker container in the background. Next, let's start a brand new ASP.NET Core Web API project with this command:

```
dotnet new webapi
```

And let's add a reference to Marten with some extra command line utilities you'll use later with:

```
dotnet add package Marten.CommandLine
```

Switching to the application bootstrapping in the **Program** file created by the **dotnet new** template that I used, I'll add the following code:

```
builder.Services.AddMarten(opts =>
    {
    var connString = builder
        .Configuration
        .GetConnectionString("marten");

    opts.Connection(connString);

    // There will be more here later...
});
```

Last, I'll add an entry to the **appsettings.json** file for the database connection string:

```
{
  "ConnectionStrings": {
    "marten": "connection string"
  }
}
```

To enable some administrative command line tooling that I'll use later, replace the last line of code in the generated **Program** file with this call:

```
// This is using the Oakton library
await app.RunOaktonCommands(args);
```

Appending Events with Marten

Marten (https://martendb.io) started its life as a library to allow .NET developers to exploit the robust JSON support in the Postgresql database engine as a full-fledged document database with a small event sourcing capability bolted onto the side. As Marten and Marten's community have grown, the event sourcing functionality has matured and probably drives most of the growth of Marten at this point.

In the telehealth system, I'll write the very simplest possible code to append events for the start of a new **ProviderShift** stream. First though, let's add some event types for the **ProviderShift** workflow:

```
public record ProviderAssigned(
    Guid AppointmentId);
public record ProviderJoined(Guid BoardId,
    Guid ProviderId);
public record ProviderReady();
public record ProviderPaused();
public record ProviderSignedOff();
public record ChartingFinished();
public record ChartingStarted();
```

I'm assuming the usage of .NET 6 or above here, so it's legal to use C# record types. That isn't mandatory for Marten usage, but it's convenient because events should never change during the lifetime of the system. Mostly for me though, using C# records just makes the code very terse and easily readable.

If you're interested, the underlying table structure for streams and events that Marten generates is shown in **Listing 1** and **Listing 2**.

Listing 1: Stream Table

```
CREATE TABLE mt_streams (
    id                  uuid           NOT NULL,
    type                varchar        NULL,
    version             bigint         NULL,
    timestamp           timestamptz    NOT NULL DEFAULT (now()),
    snapshot            jsonb          NULL,
    snapshot_version    integer        NULL,
    created             timestamptz    NOT NULL DEFAULT (now()),
    tenant_id           varchar        NULL DEFAULT '*DEFAULT*',
    is_archived         bool           NULL DEFAULT FALSE,
CONSTRAINT pkey_mt_streams_id PRIMARY KEY (id)
);
```

Listing 2: Events Table

```
CREATE TABLE mt_events (
    seq_id            bigint                      NOT NULL,
    id                uuid                        NOT NULL,
    stream_id         uuid                        NULL,
    version           bigint                      NOT NULL,
    data              jsonb                       NOT NULL,
    type              varchar(500)                NOT NULL,
    timestamp         timestamp with time zone    NOT NULL DEFAULT '(now())',
    tenant_id         varchar                     NULL DEFAULT '*DEFAULT*',
    mt_dotnet_type    varchar                     NULL,
    is_archived       bool                        NULL DEFAULT FALSE,
CONSTRAINT pkey_mt_events_seq_id PRIMARY KEY (seq_id)
);

ALTER TABLE mt_events
ADD CONSTRAINT fkey_mt_events_stream_id FOREIGN KEY(stream_id)
REFERENCES cli.mt_streams(id)ON DELETE CASCADE;
```

You'll also notice that I'm not adding a lot of members to most of the events. As you'll see in the next code sample, Marten tags all these captured events to the provider shift ID anyway. Just the name of the event type by itself denotes a domain event, so that's informative. In addition, Marten tags each event captured with metadata like the event type, the version within the stream, and, potentially, correlation and causation identifiers.

Now, on to appending events with Marten. In the following code sample, I spin up a new Marten DocumentStore that's the root of any Marten usage, then start a new ProviderShift stream with a couple initial events:

```
// This would be an input
var boardId = Guid.NewGuid();

var store = DocumentStore
    .For("connection string");

using var session = store.LightweightSession();

session.Events.StartStream<ProviderShift>(
    new ProviderJoined(boardId),
    new ProviderReady()
);

await session.SaveChangesAsync();
```

Similar to Entity Framework Core's DbContext type, the Marten **IDocumentSession** represents a unit of work that I can use to organize transactional boundaries by gathering up work that should be done inside of a single transaction, then helping to commit that work in one native Postgresql transaction. From the Marten side of things, it's perfectly possible to capture events for multiple event streams and even a mix of document updates within one **IDocumentSession**.

Projections with Marten

Now that you know how to append events, the next step is to have the provider events projected into a **write model** representing the state of the ProviderShift that you'll need later. That's where Marten's projection model comes into play.

As a simple example, let's say that you want all of the provider events for a single ProviderShift rolled up into this data structure:

```
public class ProviderShift
{
    public Guid Id { get; set; }
    public int Version { get; set; }
    public Guid BoardId { get; private set; }
    public Guid ProviderId { get; init; }
    public ProviderStatus Status {
        get; private set; }
    public string Name { get; init; }
    public Guid? AppointmentId { get; set; }

    // More here in just a minute...
}
```

Hopefully you'll be able to trace how all of this information could be gleaned from the event records like ProviderReady that I defined earlier. In essence, what you need to do is to apply the "left fold" concept from functional programming to combine all the events for a single ProviderShift event stream into that structure above.

The one exception is the ProviderShift.Version property. One of Marten's built-in naming conventions (which can be overridden) is to treat any public member of an aggregated type with the name "Version" as the stream version, such that when Marten applies the events to update the projected document, this member is set by Marten to be the most recent version number of the stream itself. To make that concrete, if a ProviderShift stream contains four events, then the version of the stream itself is 4.

As the simplest possible example, I'm going to use Marten's self-aggregate feature to add the updates by event directly to the ProviderShift type up above. Do note that it's possible to use an immutable aggregate type for this inside of Marten, but I'm choosing to use a mutable object type just because that leads to simpler code. In real usage, be aware that opting for immutable aggregate types works the garbage collection in your system harder by spawning more object allocations. Also, be careful with immutable aggregates because that can occasionally bump into JSON serialization issues that are easily avoidable with mutable aggregate types.

In this case, the event stream within the application should be started with the ProviderJoined event, so I'll add a method to the ProviderShift type up above that creates a new ProviderShift object to match that initial ProviderJoined event, like so:

```
public static async Task<ProviderShift> Create(
    ProviderJoined joined,
    IQuerySession session)
{
    var p = await session
        .LoadAsync<Provider>(joined.ProviderId);

    return new ProviderShift
    {
        Name = $"{p.FirstName} {p.LastName}",
        Status = ProviderStatus.Ready,
        ProviderId = joined.ProviderId,
        BoardId = joined.BoardId
    };
}
```

A couple notes about the code above:

- There's no interface or mandatory base class of any kind from Marten in this usage, just naming conventions.
- The method name **Create()** with the first argument type being **ProviderJoined** exercises a naming convention in Marten to identify this method as taking part in the projection.
- The Marten team urges some caution with this, but it's possible to query Marten for additional information inside the **Create()** method by passing in the Marten **IQuerySession** object.
- As implied by this code, it's quite possible with Marten to store reference or relatively static data like basic information about a provider (name, phone number, qualifications) in a persisted document type while also using the Marten event store capabilities.

Now let's add some additional methods to handle other event types. The easiest thing to do is to add more methods named **Apply(event type)** like this one:

```
public void Apply(ProviderReady ready)
{
    AppointmentId = null;
    Status = ProviderStatus.Ready;
}

public void Apply(ProviderAssigned assigned)
{
    Status = ProviderStatus.Assigned;
    AppointmentId = assigned.AppointmentId;
}
```

Or even better, if the resulting method can be a one line, use the newer C# method expression option:

```
// This is kind of a catch all for any paperwork
// the provider has to do after an appointment
// for the just concluded appointment
public void Apply(ChartingStarted charting) =>
    Status = ProviderStatus.Charting;
```

Again, to be clear, these methods are added directly to the ProviderShift class to teach Marten how to apply events to the ProviderShift aggregate.

Let's move on to applying the aggregate with Marten's "live aggregation" mode:

```
public async Task access_live_aggregation(
    IQuerySession session,
    Guid shiftId)
{
    // Fetch all the events for the stream, and
    // apply them to a ProviderShift aggregate
    var shift = await session
        .Events
        .AggregateStreamAsync<ProviderShift>(
            shiftId);
}
```

In the code above, **IQuerySession** is a read-only version of Marten's **IDocumentSession** that's available in your application's Dependency Injection container in a typical .NET Core application. The code above is fetching all the captured events for the stream identified by **shiftId**, then passed one at a time, in order, to the ProviderShift aggregate to create the current state from the events.

This usage queries for every single event for the stream, and deserializes each event object from persisted JSON in the database, so it could conceivably get slow as the event stream grows. Offhand, I'm guessing that I'm probably okay with the ProviderShift aggregation only happening "live," but I do have other options.

The second option is to use Marten's "inline" lifecycle to apply changes to the projection at the time that events are captured. To use this, I'm going to need to do just a little bit of configuration in the Marten set up:

```
var store = DocumentStore.For(opts =>
{
    opts.Connection(“connection string”);
    opts.Projections
        .SelfAggregate<ProviderShift>(
        ProjectionLifecycle.Inline);
});
```

Now, when I capture events against a ProviderShift event stream, Marten applies the new events to the persisted ProviderShift aggregate for that stream, and updates the aggregated document and appends the events in the same transaction for strong consistency:

```
var shiftId = session.Events.StartStream<ProviderShift>(
    new ProviderJoined(boardId),
    new ProviderReady()
).Id;

// The ProviderShift aggregate will be
// updated at this time
await session.SaveChangesAsync();
```

Events are Immutable

In most of the literature you'll see about Event Sourcing, the strong recommendation is to assume that event data is immutable. That's not to say that you should plan on event data being infallible.

Rather than reaching into the database to correct erroneous event data, you can use additional, corrective events to "fix" any errors.

```
// Load the persisted ProviderShift right out
// of the database
var shift = await session
    .LoadAsync<ProviderShift>(shiftId);
```

Right here, you can hopefully see the benefit of Marten coming with both a document database feature set and the event store functionality. Without any additional configuration, Marten can store the projected ProviderShift documents directly to the underlying Postgresql database.

Lastly, there's one last choice. I can use eventual consistency and allow the ProviderShift aggregate to be built in an asynchronous manner in background threads. This is going to require a little more configuration, though, as I need to be using the full application bootstrapping, as shown below:

```
builder.Services.AddMarten(opts =>
{
    // This would typically come from config
    opts.Connection("connection string");

    opts.Projections
         .SelfAggregate<ProviderShift>(
             ProjectionLifecycle.Async);
})

    // This adds a hosted service to run
    // asynchronous projections in the background
    .AddAsyncDaemon(DaemonMode.HotCold);
```

As shown in **Figure 4**, Marten has an optional subsystem called the "async daemon" that's used to process asynchronous projections with an eventual consistency model in a background process.

The async daemon runs as a .NET **IHostedService** in a background thread. The daemon constantly scans the underlying event store tables and applies new events to the registered projections. In the case of the **ProviderShift** aggregation, the async daemon applies new incoming events like the **ProviderReady** or **ProviderAssigned** events that are handled by the **ProviderShift** aggregate to update the **ProviderShift** aggregate documents and persists them using Marten's document database functionality. The async daemon comes with guarantees to:

- Apply events in sequential order
- Apply all events at least once

The async daemon is an example of eventual consistency where the query model (the ProviderShift aggregate in this case) is updated to match the incoming events rather than the strong consistency model allowed by Marten's inline projection lifecycle.

To summarize the projection lifecycles in Marten and their applicability, refer to **Table 1**.

Time Travel

One of the advantages of using Event Sourcing is the ability to use "time travel" to replay events up to a certain time to recreate the state of the system at a certain time or at a certain revision. In the sample below, I'm going to recreate the state of a given ProviderShift at a time in the past:

```
public async Task time_travel(
    IQuerySession session,
    Guid shiftId,
    DateTimeOffset atTime)
{
    // Fetch all the events for the stream, and
    // apply them to a ProviderShift aggregate
    var shift = await session
        .Events
        .AggregateStreamAsync<ProviderShift>(
            shiftId,
            timestamp:atTime);
}
```

In this usage, Marten queries for all the events for the given ProviderShift stream up to the point in time expressed by the **atTime** argument and calculating the projected state **at that time**. Inside of this fictional telehealth system, it might very well be valuable for the business to replay events throughout the day to understand how the appointments and provider interaction played out and diagnose scheduling delays.

Projecting Events to a Flat Table

One of the advantages of Marten is that it allows you to be flexible in your persistence approach within a single database engine without having to introduce yet more infrastructure. Marten was originally built to be a document database with a nascent event store capability over the top of the existing Postgresql database engine, but the event store functionality has matured greatly since then. In addition, Postgresql is a great relational database engine, so I can even take advantage of that and write projections that write some of the events to a plain old SQL table.

Back to the fictional telehealth system, one of the features I'll absolutely need is the ability to predict the wait times

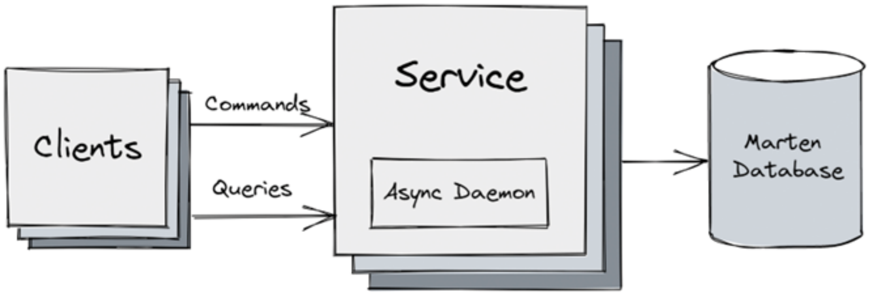


Figure 4: CQRS with Marten

Task	Description
Live	The projected documents are evaluated from the raw events on demand. This lifecycle is recommended for short event streams or in cases where you want to optimize much more for fast writes with few reads.
Inline	The projected documents are updated and persisted at the time of event capture, and in the same database transaction for a strong consistency model.
Async	Projections are updated from new events in a background process. This lifecycle should be used any time there's a concern about concurrent updates to a projected document and should almost always be used for projections that span multiple event streams.

Table 1: Marten Projection Lifecycles

that patients should expect when they request an appointment. To support that calculation, the system needs to track statistics about how long appointments last during different times of the day. To that end, I'm going to add another projection against the same events I'm already capturing, but this time, I'm going to use Marten's **EventProjection** recipe that allows me to be more explicit about how the projection handles events.

First, I'm going to start a new class for this projection and define through Marten itself what the table structure is:

```
public AppointmentDurationProjection()
{
    // Defining an extra table so Marten
    // can manage it for us
    var table
        = new Table("appointment_duration");
    table.AddColumn<Guid>("id")
        .AsPrimaryKey();
    table.AddColumn<DateTimeOffset>("start");
    table.AddColumn<DateTimeOffset>("end");

    SchemaObjects.Add(table);
}

// more later...
```

Next, using Marten's naming conventions, I'm going to add a method that handles the **AppointmentStarted** event in this projection:

```
public void Apply(
    IEvent<AppointmentStarted> @event,
    IDocumentOperations ops)
{
  var sql = "insert into appointment_duration"
      + " (id, start) values (?, ?)";
  ops.QueueSqlCommand(sql,
      @event.Id,
      @event.Timestamp);
}
```

And an additional method for the **AppointmentFinished** event:

```
public void Apply(
    IEvent<AppointmentFinished> @event,
    IDocumentOperations ops)
{
    var sql = "update appointment_duration "
              + "set end = ? where id = ?";
    ops.QueueSqlCommand(sql,
        @event.Timestamp,
        @event.Id);
}
```

The next step is to add this new projection to the system by revisiting the **AddMarten**() section of the **Program** file and adding that projection like so:

```
builder.Services.AddMarten(opts =>
{
    // other configuration...

    opts.Projections
        .Add<AppointmentDurationProjection>(
            ProjectionLifecycle.Async);

    // OR ???

    opts.Projections
        .Add<AppointmentDurationProjection>(
            ProjectionLifecycle.Inline);
});
```

There's a decision to be made about the new **AppointmentDurationProjection** that I'm adding to a system that's already in production. If I make the **AppointmentDurationProjection** asynchronous and deploy that change to production, the Marten async daemon attempts to run every historical event from the beginning of the system through this new projection until it has eventually reached what Marten calls the "high water mark" of the event store, and then continues to process new incoming events at a normal pace.

> There's the concept of stream archival in Marten that you can use to avoid the potential performance problem of having to replay every event from the beginning of the system.

If, instead, I decide to make the new **AppointmentDurationProjection** run inline with event capture transactions, that new table only reflects events that are captured from that point on. And maybe that's perfectly okay for the purposes here.

But what if, instead, I want that new projection to run inline and also want it applied to every historical event? That's the topic of the next section.

Replaying Events or Rebuilding Projections

It's an imperfect world, and there will occasionally be reasons to rebuild the stored document or data from a projection against the persisted events. Maybe I had a reason to change how the projection was created or structured? Maybe I've added a new projection? Maybe, due to intermittent errors of some sort, the async daemon had to skip over some missing events or there was some sort of "poison pill" event that Marten had to skip over due to errors in the projection code?

The point is that the events are the single source of truth, the stored projection data is a read only view of that raw data, and I can rebuild the projections from the raw events later.

Here's an example of doing this rebuild programmatically:

```
public async Task rebuild_projection(
    IDocumentStore store,
    CancellationToken cancellation)
{
    // create a new instance of the async daemon
    // as configured in the document store
    using var daemon = await store
        .BuildProjectionDaemonAsync();

    await daemon
        .RebuildProjection
        <AppointmentDurationProjection>(
        cancellation);
}
```

That code deletes any existing data in the **appointment_duration** table, reset Marten's record of the progress of the existing projection, and start to replay all non-archived events in the system from event #1 all the way to the known "high water mark" of the event store at the beginning of this operation.

This can function, simultaneously with the running application, as long as the projection being rebuilt isn't also running in the application.

To make this functionality easier to access and apply at deployment time, Marten comes with some command line extensions to your .NET application with the Marten.CommandLine library. Marten.CommandLine works with the related Oakton (https://jasperfx.github.io/oakton) library that allows .NET developers to expose additional command line tools directly to their .NET applications.

Assuming that your application has a reference to Marten.CommandLine, you can opt into the extended command line options with this line of code in your **Program** file:

```
// This is using the Oakton library
await app.RunOaktonCommands(args);
```

From the command line at the root of your project using the Marten.CommandLine library, type:

```
dotnet run -- help projections
```

to access the built-in usage help for the Oakton commands active in your system. With Marten.CommandLine, you should see some text output like this:

```
projections - Marten's asynchronous projection...
└── Marten's asynchronous projection…
    └── dotnet run -- projections
        ├── [-i, --interactive]
        ├── [-r, --rebuild]
        ├── [-p, --projection <projection>]
        ├── [-s, --store <store>]
        ├── [-l, --list]
        ├── [-d, --database <database>]
        ├── [-l, --log <log>]
        ├── [-e, --environment <environment>]
        ├── [-v, --verbose]
        ├── [-l, --log-level <loglevel>]
        └── [----config:<prop> <value>]
```

To rebuild only the new AppointmentDurationProjection from the command line, type this at the command line at the root of the telehealth system:

```
dotnet run -- projections --rebuild
-p AppointmentDurationProjection
```

This command line usage was intended for both development or testing time, but also for scripting production deployments.

The Marten team and community, of course, looks forward to the day when Marten is able to support a "zero downtime" projection rebuild model.

Command Handlers with Marten

I've spent a lot of time talking about Event Sourcing so far, but little about CQRS, so let's amend that by considering the code that you'd need to write as a command handler. As part of the telehealth system, the providers need to perform a business activity called "charting" at the end of each patient appointment where they record whatever notes or documentation is required to close out the appointment. The telehealth system absolutely needs to track the time that providers spend charting.

To mark the end of the charting activity, the system needs to accept a command message from the provider's user interface client that might look something like this:

```
public record CompleteCharting(
    Guid ShiftId,
    int Version
);
```

To write the simplest possible ASP.NET Core controller endpoint method that handles this incoming command, verifies the request against the current state of the ProviderShift, and raises a new ChartingFinished event, I'll write this code:

```
public async Task CompleteCharting(
    [FromBody] CompleteCharting charting,
    [FromServices] IDocumentSession session)
{
    var shift = await session
        .LoadAsync<ProviderShift>(
        charting.ShiftId);

    // Validate the incoming data before making
    // the status transition
    if (shift.Status != ProviderStatus.Charting)
    {
        throw new Exception("invalid request");
    }

    var finished = new ChartingFinished();
    session.Events.Append(
        charting.ShiftId,
        finished);

    await session.SaveChangesAsync();
}
```

The big thing I missed up there is any kind of concurrency protection to verify that either I'm not erroneously receiving duplicate commands for the same ProviderShift or that I want to force the commands against a single ProviderShift to be processed sequentially.

First, let's try to solve the potential concurrency issues with optimistic concurrency, meaning that I'm going to start by telling Marten what initial version of the ProviderShift stream the command thinks the stream should be at. If, at the time of saving the changes on the IDocumentSession, Marten determines that the event stream in the database has moved on from that version, Marten throws a concurrency exception and rollback the transaction.

Recent enhancements to Marten make this workflow much simpler. The following code rewrites the Web service method above to incorporate optimistic concurrency control based on the CompleteCharting.Version value that's assumed to be the initial stream version:

```
public async Task CompleteCharting(
    [FromBody] CompleteCharting charting,
    [FromServices] IDocumentSession session)
{
    var stream = await session
        .Events
        .FetchForWriting<ProviderShift>(
            charting.ShiftId,
            charting.Version);

    // Validation code...

    var finished = new ChartingFinished();
    stream.AppendOne(finished);

    await session.SaveChangesAsync();
}
```

Strong vs. Eventual Consistency

Marten is unusual for an event store tool because it offers the strongly consistent "inline" mode.

You need to be cognizant of the differences and potential problems with using the strong versus eventual consistency models. Eventual consistency may help your system scale by removing work to background processes, but can lead to subtle bugs if your developers aren't careful.

And, for another alternative, if you're comfortable with a functional programming inspired "continuation passing style" usage of Lambdas:

```
    return session
        .Events
        .WriteToAggregate<ProviderShift>(
            charting.ShiftId,
            charting.Version,
            stream =>
    {
        // validation code...

        var finished = new ChartingFinished();
        stream.AppendOne(finished);
    });
```

Optimistic concurrency checks are very efficient, assuming that actual concurrent access is rare, because it avoids any kind of potential expensive database locking. However, this requires some kind of exception-handling process that may include selective retries. That's outside the scope of this article.

Because Marten is built on top of the full-fledged Postgresql database, Marten can take advantage of Postgresql row locking to wait for exclusive access to write to a specific event stream. I'll rewrite the code in the previous sample to instead use exclusive locking:

```
    return session
        .Events
        .WriteExclusivelyToAggregate
            <ProviderShift>(
            charting.ShiftId,
            stream =>
    {
        // validation code...

        var finished = new ChartingFinished();
        stream.AppendOne(finished);
    });
```

This usage uses the database itself to order concurrent operations against a single event stream, but be aware that this usage can also throw exceptions if Marten is unable to attain a write lock on the event stream before timing out.

Summary

Marten is one of the most robust and feature-complete tools for Event Sourcing on the .NET stack. Arguably, Marten is an easy solution for Event Sourcing within CQRS solutions because of its "event store in a box" inclusion of both the event store and asynchronous projection model within one single library and database engine.

Event Sourcing is quite different from the traditional approach of persisting system state in a single database structure, but has strengths that may well fit business domains better than the traditional approach. CQRS can be done without necessarily having a complicated infrastructure.

Jeremy D. Miller

MAUI

ONLINE QUICK ID 2209081

Putting Data Science into Power BI

Microsoft’s Power BI works as the ultimate power tool for data analytics. It lets you connect to many different data source types (even within the same model) and then transform the connections into useful data tables. You can then use this data to create DAX calculations, and build visuals to communicate model trends, outcomes, and key numbers. The main languages of Power BI are M (in Power Query) and DAX. Data science is an area in the data analytics space focusing on models like those that make predictions. Artificial intelligence is an area of data science that lets you use cognitive science to recognize and act on patterns within the data points that you have. Machine learning models are a subgroup of AI that involve using feedback loops to further improve the model. You can combine and use these data science models to create visuals and make forecasts and better decisions in the future. The three main languages of data science are SQL, R, and Python.

Helen Wall
http://www.linkedin.com/in/helenrmwall/
www.helendatadesign.com

Helen Wall is a data science consultant who founded Helen Data Design. She is a power user of Microsoft Power BI, Excel, Tableau, and AWS QuickSight. Her primary driver behind working in these tools is finding the point where data science and design intersect.

She is a LinkedIn Learning instructor for data science courses focusing on Power BI, AWS QuickSight, Excel, R, and Python. She is also a lecturer at the Rice University business school focusing on Python, as well as an instructor at Cornell University’s online certificate programs for data science and analytics using R and Excel.

She has a double bachelor’s degree from the University of Washington where she studied math and economics and was a Division I varsity rower. (The real-life characters from the book The Boys in the Boat were Husky rowers that came before her). She has a master’s degree in financial management from Durham University in England.

Given the power of Power BI and data science, how can you combine these two facets of data modeling together? The data for this article focuses on economic and weather trends in the greater Houston, Texas area. One data table contains employment numbers from the U.S. Bureau of Labor Statistics (BLS Data Viewer at https://beta.bls.gov/dataViewer/view/timeseries/LAUCN482010000000004 and BLS Data Viewer at https://beta.bls.gov/dataViewer/view/timeseries/LAUCN482010000000005). The other data table contains the daily high temperatures at the city’s Hobby Airport over the last two years from the NOAA Climate Data Online (CDO) data portal (https://www.ncdc.noaa.gov/cdo-web/).

Ways to Leverage Algorithms in Power BI

One way you can explore a combined framework with BI and data science is through the capabilities of Power BI. To see the opportunities for these algorithms, let’s divide the AI and machine learning functionalities within Power BI into three categories.

- Those that Power BI automatically runs
- Pre-built models you can connect to within Power BI
- Models you can build yourself using R, Python, or even DAX

Power BI makes these algorithms available to you in the Power Query Editor once you load the data into Power BI Desktop through the modeling and visualization options. **Figure 1** shows the combined capabilities available within a single query for all three categories listed above.

Power BI Guesses

In **Figure 1**, you can see that Power Query automatically chooses the data type for each column in the existing query so far. You can change the data types yourself if it doesn’t already do this, or if the automatically selected data types don’t match with the data types you want to use. Power Query uses the first 1000 rows of data that appear in the table preview to make an educated guess for the actual data types. Similarly, it can also guess whether you want to promote the first row of the data table into the header position or, given enough information, it can also guess an entire series of query steps that you can see in the Applied Steps on the right.

Once you load the data into Power BI, you can explore several visuals where you pick the visual, but it nudges you toward the next step. For example, in the Model view, you might see the tables automatically joined together based on how Power BI thinks the dimension and fact tables connect. You should also check to make sure that it joins the tables on the fields you want them to join on.

When you configure visuals, the decomposition tree and key influencers visuals use AI to predict the next step that you or the end user should take in analyzing the data in the visual. You can also leverage the smart narrative visual for the insights that Power BI automatically provides as to why trends or metrics might occur.

Connect to a Model

Power BI also lets you connect to built-in algorithms directly in the Power Query Editor as part of the transformation process with the AI Insights options like Text Analytics, Vision, and Azure Machine Learning, like the options you see in the options for the Add Column ribbon in **Figure 1**. In the Power Query Editor, for example, you can connect to models from Azure Cognitive Services and Azure Machine Learning models built outside Power BI and Power Query. Examples of available Azure Cognitive Services models include Image Recognition and Text Analytics. Within Text Analytics, you can choose from algorithms like language detection, key phrase extraction, and score sentiment (which tells you the positive or negative tone of a text input).

The fuzzy matching algorithm uses natural language processing (NLP) to match together similar strings of text. If you’re using Power BI dataflows in the Power BI service (either Pro or Premium accounts), you can connect to the cluster values algorithm to transform the existing data table by either adding a column or grouping the existing values in the grouped column together. Both functionalities use a very similar fuzzy matching algorithm to what you see in the merging functionality, except it only returns results on a single data table instead of combining two tables together. Fuzzy matching uses NLP to match together similar strings of text. You can configure the parameters for the matching within the fuzzy matching options for all three iterations of this algorithm.

Another example of an NLP model within a Power BI visual is the Q&A visual, which lets you ask questions and get re-

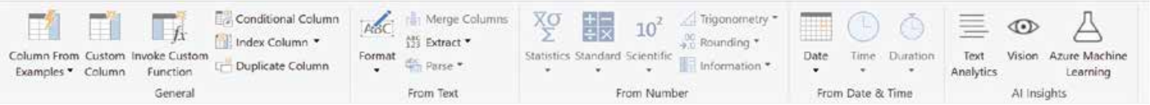


Figure 1: Levels of coding in Power Query

sponses about the data. With the visualization options, you can also connect to the built-in machine learning algorithms directly to find clusters or anomalies in existing data points. You can also use linear regression to find trend lines and forecasting to project the data trends into the immediate future. With any of these pre-built models, you'll want to either know or have an idea of the fields you want to use in the model. Even though you don't have to build them yourself, it's still important to know what fields you can pass into the model as parameters to get the outcome you're looking for (and the requirements to make the models work properly).

Constructing Your Own Visuals

Finally, you can build your own visuals to represent the outcomes of these algorithms. One way you can do this is using custom visuals from the Power BI AppSource store. Many of these visuals use R behind the scenes to construct the visual, but you don't need to write any R code yourself. Examples of models supported by custom visuals using R include ARIMA, TBATS, clustering, and outliers. Power BI installs packages. Make sure you're using the right versions of the library packages.

In addition to importing custom visuals that run R behind the scenes in Power BI, you can also write R scripts directly in the R and the Python visuals, as well as running scripts for both languages in the Power Query Editor. For the sake of simplicity in this article, I'm going to use R as the sample code, but you can absolutely use Python too. One of the challenges I encounter is that although the Power BI service supports almost one thousand R packages, it only supports a few Python libraries. To use these languages directly in Power BI, you need to do the following:

- Install R or Python on your own computer.
- Enable R and/or Python scripts to run in Power BI (and while you're at it, enable Python scripts because I'll discuss this later).

1. Once you upload reports as analysis to the Power BI service, it will run through the cloud instead of your computer.

What Are You Looking For?

Build machine learning models through building scatter plot and line chart visuals as starting points for understanding the high-level behavior of the data. Understanding algorithms can seem intimidating but you can divide what you're looking for into three different goal categories.

- Trends
- Groups
- Outliers or anomalies

Trends

When you're looking at data points, you want to see if there's a direction that they orientate in. For example, if you look at the time series trends for employment and unemployment data by month on the left side of **Figure 2**, you can see that

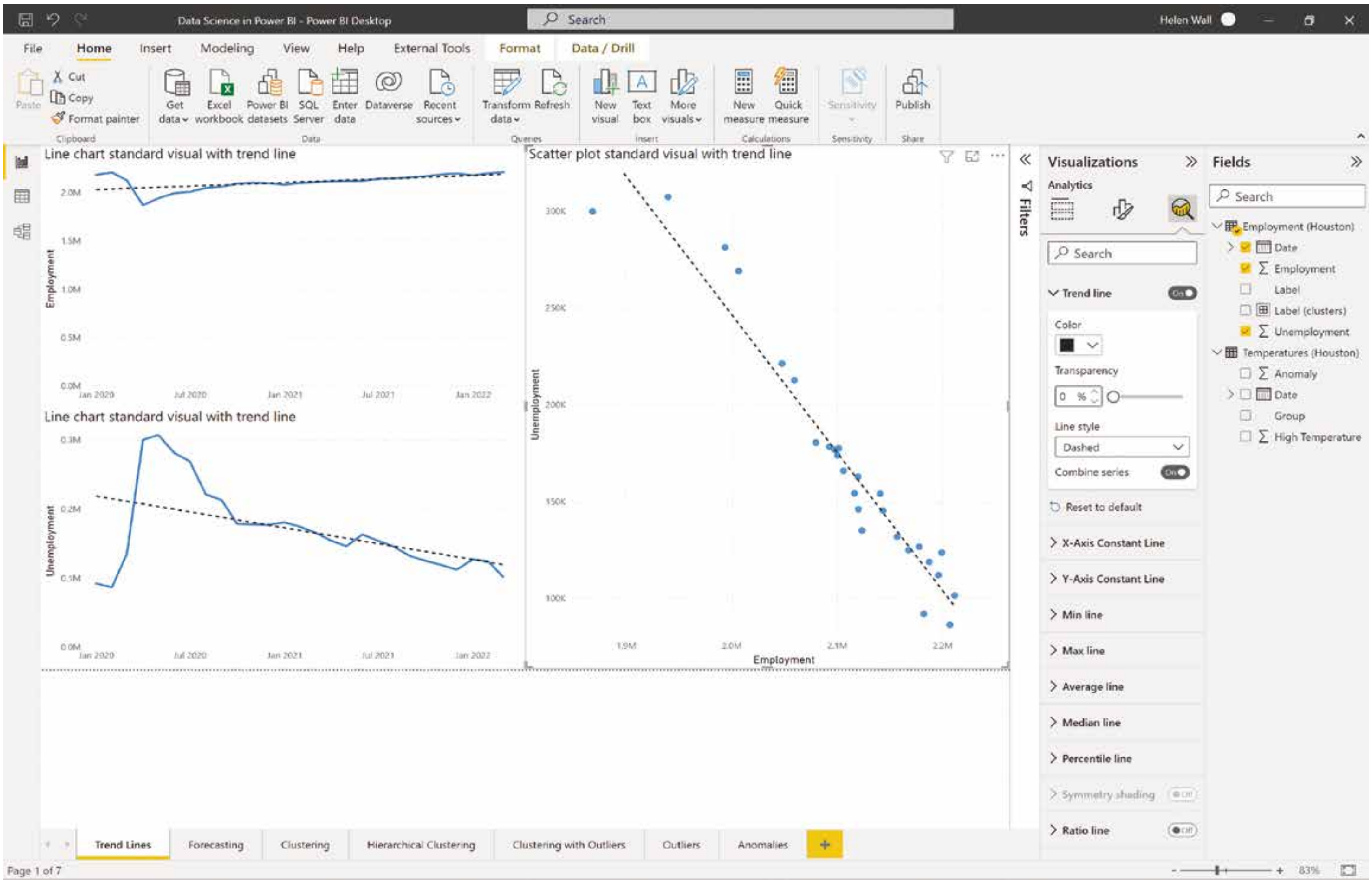


Figure 2: Trend lines

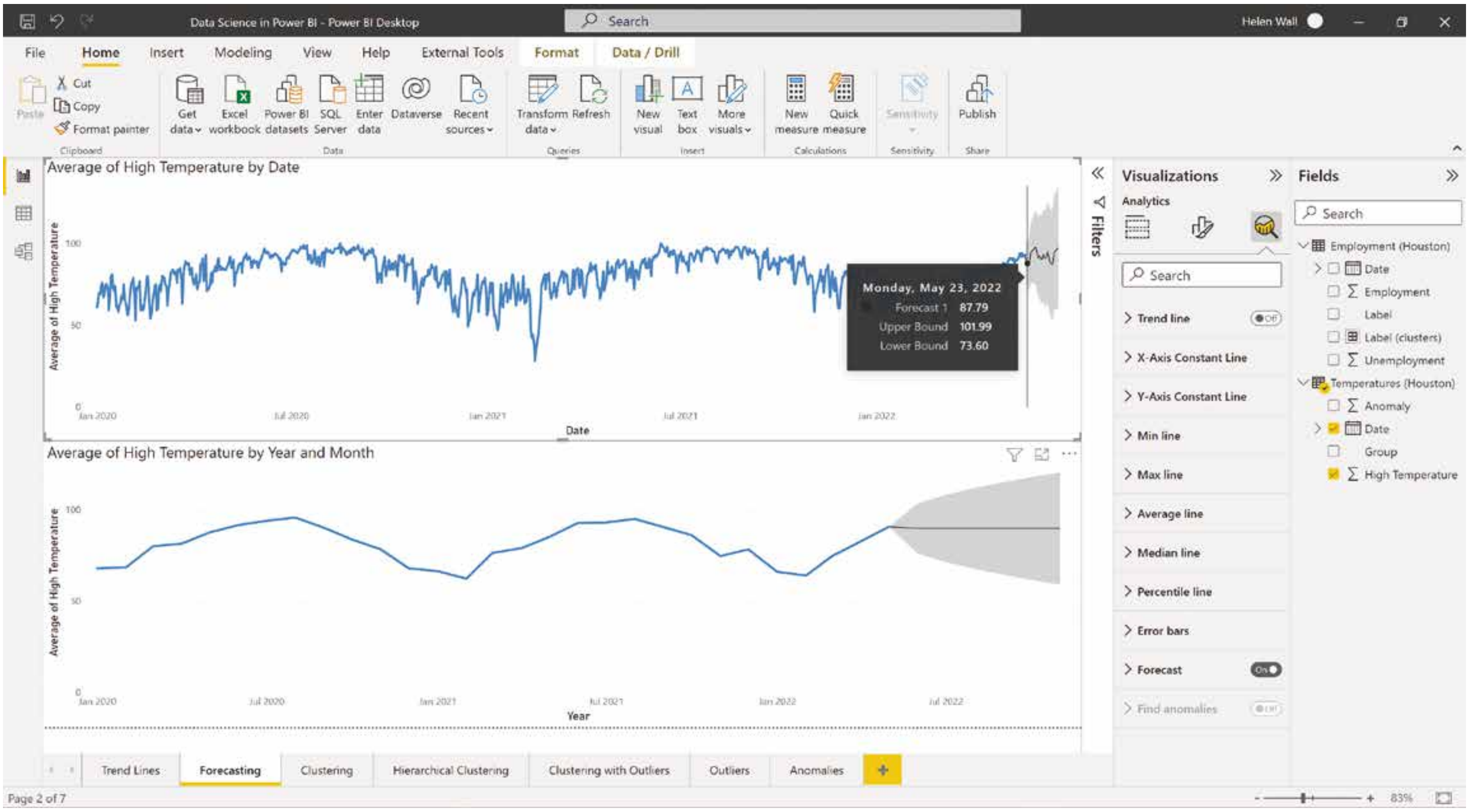


Figure 3: Forecasting

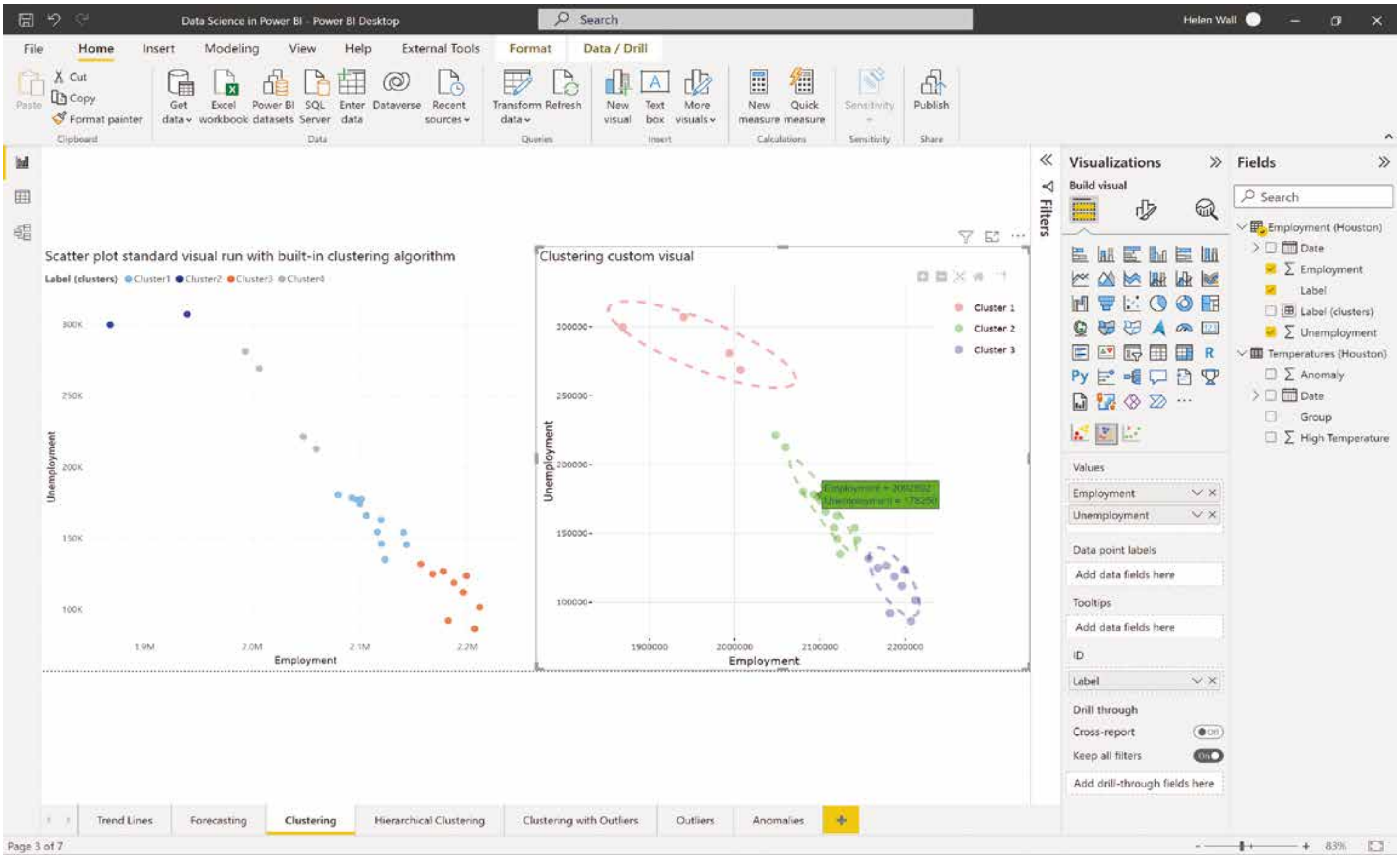


Figure 4: Clustering with built-in options and clustering custom visual

since the beginning of 2020, the overall employment numbers increased in the Houston area, while the overall unemployment numbers decrease over the same period. You can add these dashed lines directly to several types of visuals, including the line charts you see representing the time-series trends for both these metrics. To add these lines, turn them on directly through the analytics options in the Visualizations pane. The lines you see represent the outcomes of linear regression modeling using ordinary least squares (OLS). If you calculated this yourself, whether that's through downloading the data to Excel, running an R or Python script on it, or even calculating it directly using DAX, you'll get the same slope and intercept that you see on these charts (see **Figure 2**).

You can also see how linear regression looks on a scatter plot instead of a time series chart in the visual on the right of **Figure 2**. This models two variables against each other. You can see that as employment increases in Harris County, Texas, the numbers for unemployment also go down. Power BI lets you add the trend line in the same way you could for the line chart visual on the scatter plot. Again, it uses OLS for linear regression to calculate the intercept and slope of this trend line.

Let's say you want to forecast the outcomes of time-series data into the future. You can do this through the forecasting option available at the bottom of the same analytics pane as the trend line options that you see in **Figure 3**. Note though, that the forecasting option only works on visuals like line charts with a time-series field on the x-axis, like dates. Within the formatting options, you can also change the length of the forecast, and whether to ignore some historical data points, and, most importantly, you can also choose to include seasonality. Notice that the forecast option adds both a line and a gray shaded area around it representing the confidence interval.

Groups

Another way you can apply machine learning algorithms to data points is by grouping them together. Examples of clustering algorithms include names you might already know like KMeans and hierarchical clustering. If you have an existing scatter plot, let Power BI find the clusters for you using the built-in clustering algorithm. This adds a new clusters field to existing fields that contain the outcomes of the clustering model in the table that you choose to add them to. Within the clustering options, you can let Power BI automatically determine the number of clusters. You can also change them manually. In the scatter plot on the left in **Figure 4**, you can see that the built-in clustering algorithm you add to the visual creates four clusters for the data. You can also find clusters for more than two fields if you use a table visual instead of a scatter plot.

On the right of **Figure 4**, you can see what the clustering algorithm gives you if you import the clustering custom visual

Let Power BI Do as Much of the Work as Possible

Although it seems tempting to try to gain full control over the models by writing scripts in R or Python for example, you should aim to let Power BI do a lot of the heavy lifting for you. An example of this includes using built-in Power Query functions to connect to and transform the data. This means that Power BI might automatically perform a step for you (that you should then check) or you can even connect to an algorithm like one for text analytics, image recognition, or even clustering. Once you load the data into Power BI, you can also follow a very similar approach.

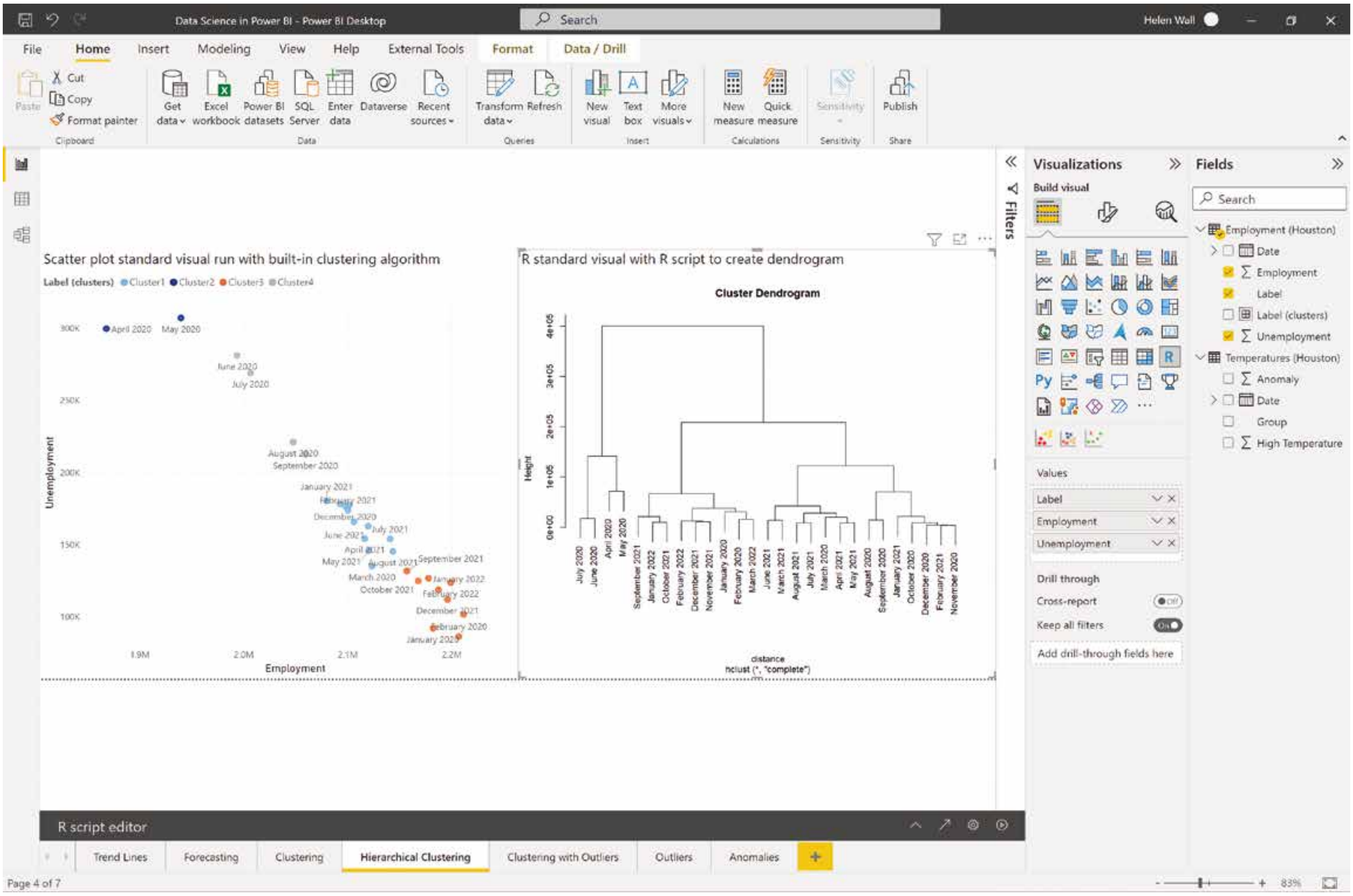


Figure 5: Hierarchical clustering with R script

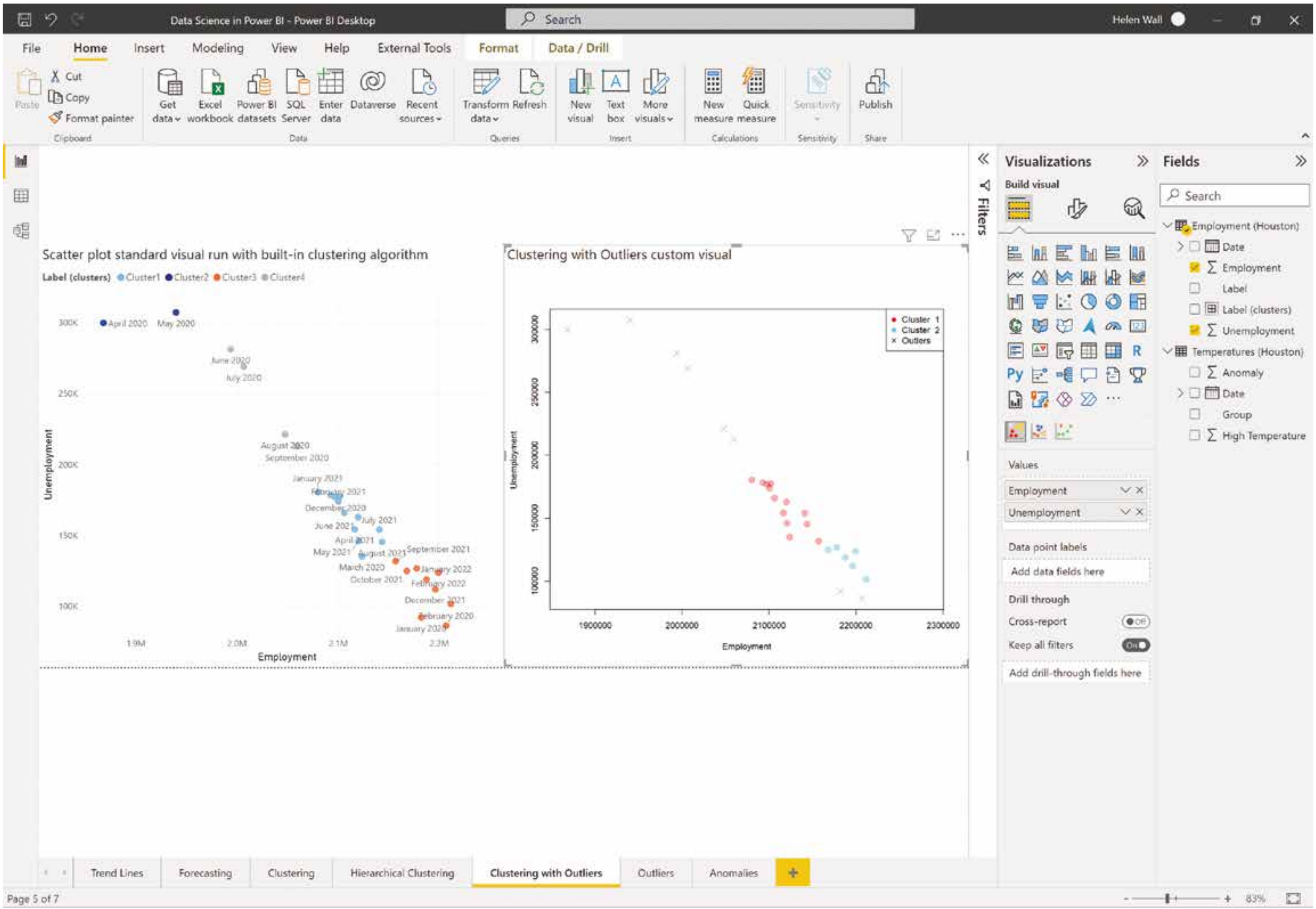


Figure 6: Clustering with outlier custom visual

from the Power BI AppSource store. This means that you can add the ellipsis around the data points in the clusters that the visual determines. This gives an example of an algorithm where the R script runs behind the scenes, but you don't have to write the R code yourself for it to appear. You can see that they also display in the green tooltip in the highlighted visual.

Another way you can find clusters to group data points together is using the hierarchical clustering algorithm. At this point, you don't have a visual or algorithm you can plug the data into to create the visual you see on the right in **Figure 5**, but instead, you can construct it using R code.

First calculate the distances between data points on the left of **Figure 5** using the dist function on the data.frame dataset variable with the text labels removed. You then group each set together in pairs using the hclust R function. In order for these visuals to properly display, use the standard R visual in Power BI. Before you create these visuals directly in Power BI Desktop, make sure you install R (or Python) and then enable it directly in Power BI.

```
# The following code to create a dataframe and
#remove duplicated rows is always executed and
#acts as a preamble for your script:

# dataset <- data.frame(Label, Employment, Unemployment)
# dataset <- unique(dataset)
# Paste or type your script code here:

rownames(dataset) <- dataset$Label
#determine row labels in final visual
distance <- dist(dataset[, c('Employment', 'Unemployment')]
, diag = TRUE)
#2D distance between data points
hc <- hclust(distance)
#model hierarchical clustering
plot(hc)
#create cluster dendrogram plot
```

You might also find it helpful to test out your code first on an IDE like RStudio, which makes it easier to troubleshoot issues. Although Power BI is an amazing tool, it's also a bit limited in terms of ways to test out code before implementing it.

Outliers or Anomalies

Finally, in data points, you want to determine whether points are part of the rest of the data points or not, which you can do through algorithms like outlier and anomaly detection. In **Figure 6**, you can see the clustering with the

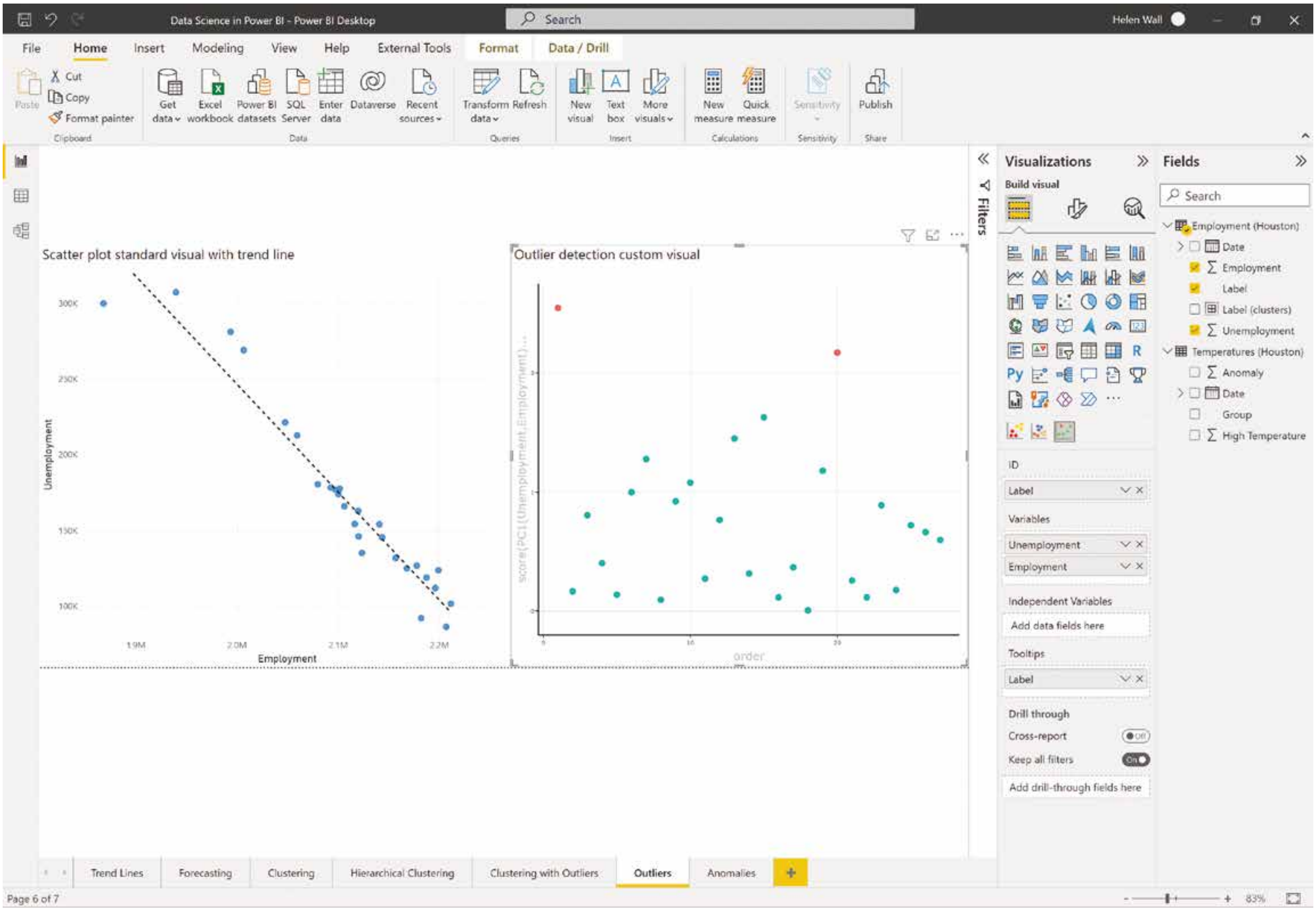


Figure 7: Outlier detection custom visual

outlier detection custom visual compared to the built-in clustering algorithm. You can see the outliers denoted by small gray Xs in the visual on the right that fall outside the two clusters marked in teal and red.

Grouping data together in clustering and determining the points outside these clusters as outliers represents one way to find outliers, but there are other ways to do it. You can also determine outliers using the outlier detection custom visual you see in **Figure 7**. This visual lets you separate the outliers from the rest of the points (the main group) using a z-score calculation to determine their sigma thresholds. The farther out points represent the outliers in red and the rest of the points aren't part of the outlier group. Like the clustering with outliers visual, the outlier detection visual also runs R code behind the scenes without you having to write any of it.

Besides outliers, anomalies are data points that also don't fit into the expected pattern of behavior for data points. On a high level, outliers represent deviations from where you are, and anomalies represent deviations from where you should be. You can see the outcome of running the built-in algorithm to find anomalies in the top line chart of **Figure 8**, which you access through the analytics pane of the selected visual. For Power BI to automatically find the anomalies for you, put a time-series field on the x-axis or it will gray out the algorithm in the analytics pane so you can't access them.

Notice that you can change the input parameters for finding the anomalies by changing the sensitivity of the algorithm. A higher sensitivity number makes the identified anomalies more sensitive to swings, which means that you'll see the algorithm identify more data points as anomalies. If you'd like to format the anomalies themselves, you can change their shape and color (from gray to orange like you see in this example).

In the lower visual in **Figure 8**, you can see the outcomes of running an anomaly detection algorithm directly with an R script. You can see it reflected in the outcome of a standard stacked column chart visual where you use conditional formatting so that orange can mark the anomalies while the rest of the dates display as a blue color. The algorithm itself can run as a standard R visual, but you can also use the R script integration options in the Power Query Editor to add a column for the anomaly detection model in **Figure 1**.

With the applied step for running an R script in the list of these steps on the right, the code below shows what this R

What to Look for?

Although this might seem to project the overall objectives of algorithms like those found in machine learning, on a high level, you're looking to find trends, grouping, and outliers and anomalies in data points. These don't necessarily exist as standalone things that you're looking for in data either. For example, you can find trends and groups in data, and then the points that allow you to easily find the points that are outliers or anomalies.

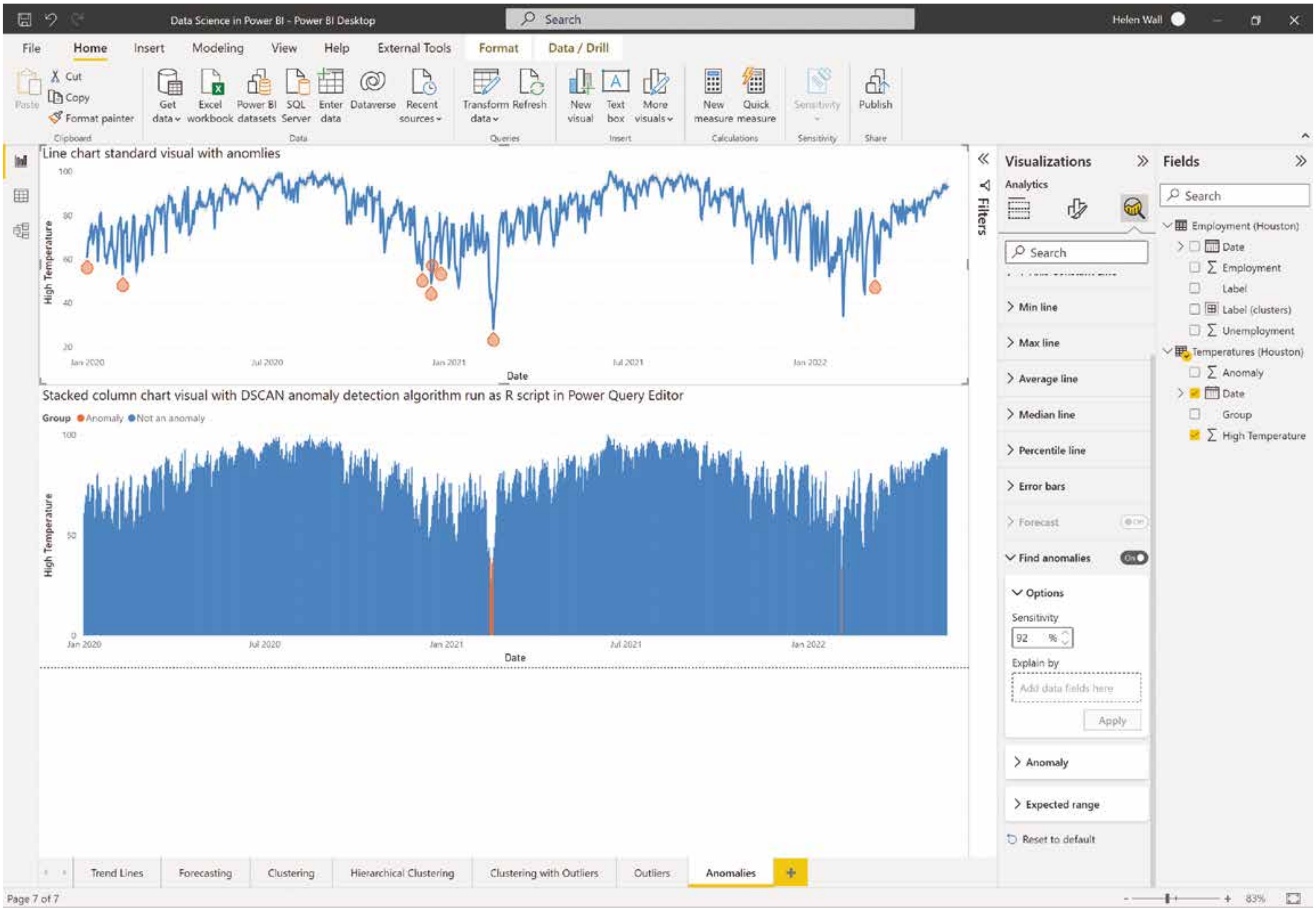


Figure 8: Anomaly detection

code looks like. Once you import the fpc library, you then set the seed so you can run the anomaly detection algorithm with the dbscan function. Then assign the outcome results cluster column as a new column in the existing dataset variable and assign the entire dataset to a new results variable in Power Query.

```
# 'dataset' holds the input data for this
#  script

library(fpc) #loading package
set.seed(220)  #setting seed
results <- dbscan(dataset$High,eps=2,MinPts=1)
dataset$Anomaly <- results$cluster
#add cluster to dataset data.frame
outcome <- dataset
```

Why would you choose one approach for clustering or anomaly detection over another (for example, built-in algorithms versus writing your own R code). There isn't a single right answer for this. You might want different levels of control over the outcomes, or you might want to see a certain level of efficiency or speed that one approach provides. Like with everything else in data science, there isn't one right approach for the way to do something, but rather a selection of options to choose from.

Additional algorithms to explore include logistic regression, principal components analysis (PCA), classification, and much more. I go into these topics in much greater depth in several of my LinkedIn Learning courses: https://www.linkedin.com/learning/instructors/helen-wall.

Helen Wall

shutterstock/Arunas Gabalis

ONLINE QUICK ID 2209091

Getting Started with Cloud Native Buildpacks

Peter Mbanugo
p.mbanugo@yahoo.com
www.pmbanugo.me
@p_mbanugo

Peter Mbanugo is a technical writer and software engineer who codes in JavaScript and C#.
He is the author of “How to build a serverless app platform on Kubernetes”.
He has experience working on the Microsoft stack of technologies and also building full-stack applications in JavaScript.
He’s a co-chair on NodeJS Nigeria, a Twilio Champion, and a contributor to the Knative open-source project. You can find his OSS contributions at github.com/pmabnugo.

When he isn’t coding, he enjoys writing the technical articles that you can find on his website or other publications, such as on Pluralsight and Telerik.

Cloud Native Buildpacks transform your source code into images that can run on any cloud. They take advantage of modern container standards such as cross-repository blob mounting and image layer "rebasing," and, in turn, produce OCI-compliant images. You use an image because it’s a lightweight, standalone, executable package of software that includes everything you need to run an application: code, runtime, system libraries, and settings.

When you tell Docker (or any similar tool) to build an image by executing the Docker build command, it reads the instructions in the Dockerfile, executes them, and creates an image as a result. Writing Dockerfiles that produce secure and optimized images isn’t an easy feat. You need to know and stay updated about best practices or, if you're not careful, you may create images that take a long time to build. They may also not be secure.

Rather than investing time in optimizing images, you may want to focus on the business logic of your software. Fortunately, there’s a tool that can read your source code and output an optimized OCI compliant image. This is what Cloud Native Buildpacks can do for you. You can use this tool in your software delivery process to automatically produce images without needing a Dockerfile.

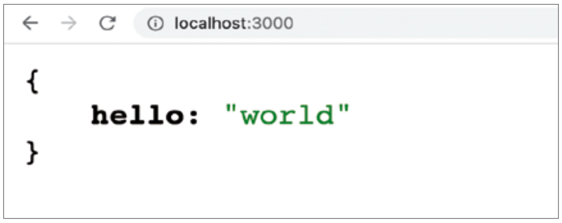
This article introduces you to Cloud Native Buildpacks and shows you an example of how to use them in GitHub Actions. By the end of the article, you’ll have a CI pipeline that builds and publishes an image to Docker Hub.

What Are Cloud Native Buildpacks?

Cloud Native (technologies that take full advantage of the cloud and cloud technologies) Buildpacks are pluggable, modular tools that transform application source code into container images. Their job is to collect everything your app needs to build and run. Among other benefits, they replace **Dockerfile** in the app development lifecycle, enable swift rebasing of images, and provide modular control over images (through the use of builders).

How Do They Work?

Buildpacks examine your app to determine the dependencies it needs and how to run it, then packages it all as a runnable container image. Typically, you run your source code through one or more buildpacks. Each buildpack goes through two phases: the detect phase and the build phase.



```
{
    hello: "world"
}
```

Figure 1: The JSON response

The detect phase runs against your source code to determine whether a buildpack is applicable or not. If it detects an applicable buildpack, it proceeds to the build stage. If the project fails detection, it skips the build stage for that specific buildpack.

The build phase runs against your source code to download dependencies and compile your source code (if needed), and set the appropriate entry point and startup scripts.

Containerize a Node.js Web App

Let’s create an image for a Node.js WSb application. You’re going to build a minimal REST API using Node.js. I prepared a starter repo at https://github.com/pmbanugo/fastify-todo-example, which you will fork and modify. Follow the steps below to clone and prepare the application:

1. Clone your fork of the repository.
2. Check out the **code-magazine** branch.
3. Open the terminal and run **npm install** to install the dependencies.
4. Open the project in your preferred code editor/IDE.

The project is a Web API built using a Fastify framework with just one route. Try out the application by opening the terminal and running the command **npm start**. The application should start and be ready to serve requests from localhost:3000. Open your browser to localhost:3000 and you should get a JSON response, as depicted in **Figure 1**.

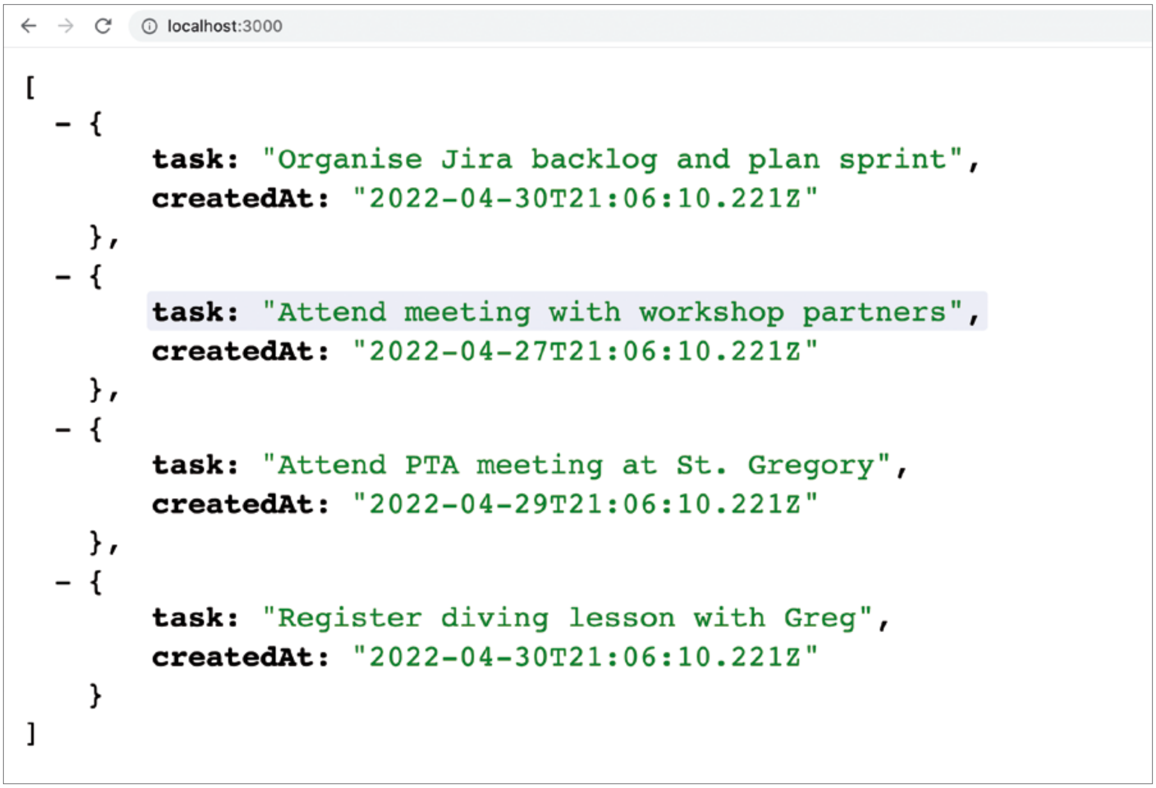
You want to modify the response so that the JSON data in **todo.json** is returned. Open server.js and replace **reply.send({ hello: "world" })** on line 7 with the code below:

```
const data = Object.entries(todos)
               .map((x) => x[1]);
reply.send(data);
```

Restart the server and open localhost:3000 in the browser. You should now get a list of todo items returned as a JSON array, as shown in **Figure 2**.

Building and Running a Container Image

Let’s build a container image of the Node.js Web app and run it locally. You don’t need a Dockerfile; instead you’ll use the pack CLI to build the image and Docker to run the container. If you don’t have Docker installed, go to docker.com to download and install Docker Desktop. You can install the pack CLI using Homebrew by executing the command **brew install buildpacks/tap/pack**. If you don’t use Homebrew, you can find more installation options at https://buildpacks.io/docs/tools/pack/#install.



```
[
  - {
        task: "Organise Jira backlog and plan sprint",
        createdAt: "2022-04-30T21:06:10.221Z"
    },
  - {
        task: "Attend meeting with workshop partners",
        createdAt: "2022-04-27T21:06:10.221Z"
    },
  - {
        task: "Attend PTA meeting at St. Gregory",
        createdAt: "2022-04-29T21:06:10.221Z"
    },
  - {
        task: "Register diving lesson with Greg",
        createdAt: "2022-04-30T21:06:10.221Z"
    }
]
```

Figure 2: The todo items returned as a JSON aray

Open your terminal and run the command **pack build todo-fastify --builder paketobuildpacks/builder:base** to build a container image using **paketobuildpacks/builder:base** as the builder image. The builder is an image that contains all the components necessary to execute a build, which includes the buildpacks and files that configure various aspects of the build. If you look through the output of the command, you should notice that during the detect phase, six buildpacks were detected to take part in the build phase (see F**igure 3**). These six buildpacks are then used to build and export an image.

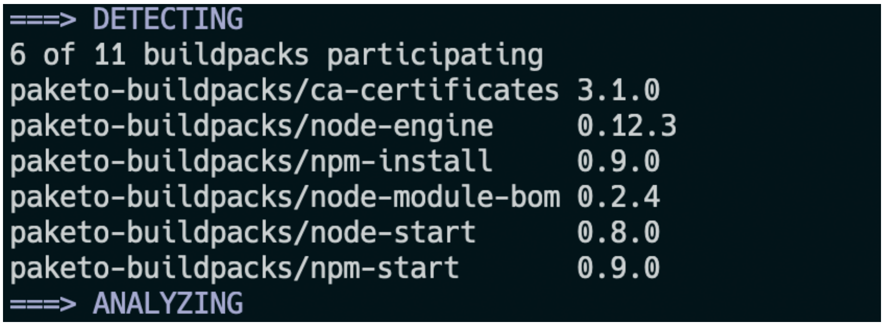
After the image is built, you'll run it using Docker. Run the command **docker run -d --rm -p 8080:3000 todo-fastify** to start the container and open localhost:8080. It should return the same JSON array as you get when running it without Docker. Stop the container using the command **docker stop CONTAINER_ID**. Replace CONTAINER_ID with the value that was returned when you started the container.

Rebuilding The Image

You're going to add another route that returns an item based on its key. Open server.js and add the code snippet below after line 10.

```
fastify.get("/:id", function (request, reply) {
  const data = todos[request.params.id];
  reply.send(data);
});
```

The new route gets the **id** from the request params, uses it to get a specific item from the **todos** object, and then returns the item as JSON.



```
===> DETECTING
6 of 11 buildpacks participating
paketo-buildpacks/ca-certificates 3.1.0
paketo-buildpacks/node-engine     0.12.3
paketo-buildpacks/npm-install     0.9.0
paketo-buildpacks/node-module-bom 0.2.4
paketo-buildpacks/node-start      0.8.0
paketo-buildpacks/npm-start       0.9.0
===> ANALYZING
```

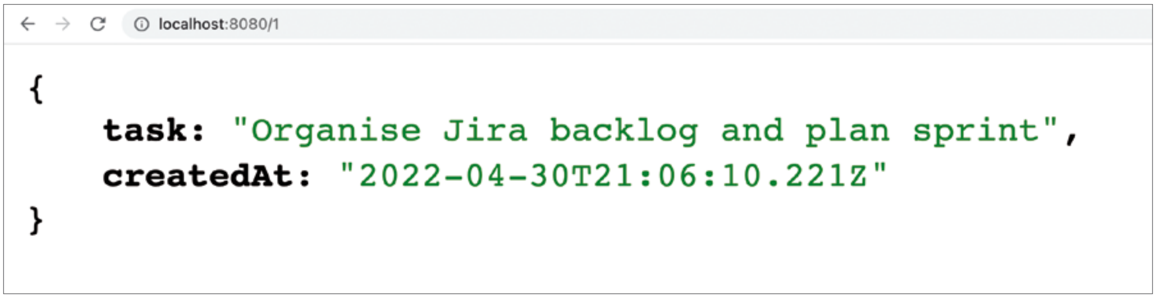
Figure 3: The six buildpacks have been detected.

Now that you've modified the code, you need to rebuild the image and run the container to test that the application still works. Open your terminal and run the command **pack build todo-fastify --builder paketobuildpacks/builder:base** to build the image. You should notice that the second build (and subsequent builds) are much faster because the images needed for the build processes were downloaded and cached in the initial run.

Now run the command **docker run -d --rm -p 8080:3000 todo-fastify** to start the container. Open http://localhost:8080/1 in your browser. You should get a JSON response similar to what you see in **Figure 4**.

Building an Image from a CI Pipeline

You can build images in your continuous integration pipeline using Cloud Native Buildpacks. With GitHub Actions, there's a Pack Docker Action (https://github.com/marketplace/actions/pack-docker-action) that you can use. When you com-



```
{
    task: "Organise Jira backlog and plan sprint",
    createdAt: "2022-04-30T21:06:10.221Z"
}
```

Figure 4: The JSON response

bine it with the Docker Login Action, you can build and publish to a registry in your workflow. There's a similar process on GitLab using GitLab's Auto DevOps, and you can read about it on https://docs.gitlab.com/ee/topics/autodevops/stages.html#auto-build-using-cloud-native-buildpacks.

I included a GitHub Actions workflow as part of the starter files in the repository you forked. You'll find it in the **.github/workflows/publish.yaml** file. The workflow builds an image and publishes it to Docker Hub whenever you push new commits to your GitHub repository.

Let's take a look at the publish.yaml file to understand what it does.

The build-publish job defines two environment variables.

```
env:
  USERNAME: '<USER_NAME>'
  IMG_NAME: 'todo-fastify'
```

IMG_NAME holds the name of the image, in this case, called *todo-fastify*. The **USER_NAME** variable is the Docker registry's namespace where the image is stored. Replace the value with your Docker Hub username.

There are four steps in this job, namely Checkout, Set App Name, Docker login, and Pack Build:

```
- name: Checkout
  uses: actions/checkout@v2
- name: Set App Name
  run: 'echo "IMG=$(echo ${USERNAME})/
      $(echo ${IMG_NAME})" >> $GITHUB_ENV'
- name: Docker login
  uses: docker/login-action@v1
  with:
    username: ${{ env.USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}
- name: Pack Build
  uses: dfreilich/pack-action@v2
  with:
    args: 'build ${{ env.IMG }} --builder
     paketobuildpacks/builder:base --publish'
```

The **Checkout** step clones and checks out the branch. After that, the **Set App Name** step adds a new environment variable named **IMG**. The value is formed by concatenating **USERNAME** and **IMG_NAME** variables.

The **Docker login** step authenticates the workflow run against the Docker registry because the final step builds and publishes the image. The **Pack Build** step uses the **dfreilich/pack-action** action to build the application and publish the image to the Docker registry. This action uses the Pack CLI behind the scenes, which, in turn, depends on Docker to build and publish to a registry.

The **args** supplied to **dfreilich/pack-action** tells it to run the **build** command using the paketobuildpacks/builder:base builder image. The **--publish** flag instructs the pack CLI to publish to the registry after the build process is complete.

The Docker login step needs a DOCKERHUB_TOKEN secret. Go to Docker Hub and create an access token. Then add a GitHub secret named **DOCKERHUB_TOKEN** with its value set to your Docker Hub's access token.

Now commit your changes and push your commits back to your GitHub remote. You should see the workflow run and when it's done, the image should be in your Docker registry repository.

Builder and Buildpacks

A builder is an image that contains buildpacks and the detection order in which builds are executed. There are different buildpacks from different vendors that you can use, such as those from Heroku and Google. Use the links below to check out some available builders and buildpacks:

- **Heroku**: hub.docker.com/r/heroku/buildpacks
- **Google**: github.com/GoogleCloudPlatform/buildpacks
- **Paketo**: paketo.io/docs/concepts/builders/

Visit www.buildpacks.io if you want to read more about Cloud Native Buildpacks.

Conclusion

I've shown you how to build images locally using the pack CLI, and also how to use it within GitHub Actions. You need a builder to build an image, and you used paketobuildpacks/builder:base as the builder image.

Peter Mbanugo

(Continued from 74)

patterns (and conform where necessary) to specific facts and circumstances. Organizations, in my opinion, aren't—or at least shouldn't be—any different.

In any successful consulting engagement, strategy and tactics must meet somewhere in the middle. Leadership creates the policy that the rank-and-file staff must execute. And good and actionable policy requires leadership. The benefits we strive to achieve for our client's benefit can only be as good as the client's ability to take our recommendations.

As **Figure 1** illustrates, the central unifying role that touches all other roles is the chief executive (in black). Organizations are run by people. As much as we would like to have clear consensus from the group, there often needs to be one person who makes the call. Ideally, that call is informed by the input of the other "chiefs." The ones in yellow on my model are primarily split between Administration and Operations: the chief administrative officer (CAO) and chief operations officer (COO).

An organization's administrative function is concerned with **what** gets done. Operations is concerned with **how** things get done. The technical counterpart that supports the what and the how are the chief information officer (CIO) and the chief technical officer (CTO). The CIO sets forth the technical strategy of what's to be done. The CTO sets forth how that strategy is to be executed.

The third category of roles, in green, sets forth the four major functions I've identified as being common to any successful organization. The chief human resources officer (CHO) is all about the health, well-being, and development of an organization's personnel. An organization can only be as good as its people, and more specifically, how it treats its people. The chief compliance and legal officer (CCO) is the one who makes sure the rules, whether they be internal, external, legal, or regulatory are followed. The chief financial officer (CFO) is concerned with capital. How are projects financed? Will it be through organic growth and internally generated cash? Or will it come by way of external sources like debt or equity investment? Finally, there's the chief marketing officer (CMO). It isn't enough to have great ideas, products, and services. The world needs to know about them to purchase them!

The foregoing list of roles is by no means exclusive. The model is simply my conception of how a canonical organization should be organized. It's far more important that each role be given its proper due. Organizations best situated to grow and improve through consulting have these roles and they don't operate in a vacuum, they operate cooperatively, and they act as a check and balance for the other roles. For example, the CCO would be a check and balance on the CIO/CTO to ensure that the Devops scheme supports the representations made by the CFO in the organization's public reporting.

Invariably, any technical consulting engagement will touch on one or more of these areas. How an organization makes decisions and evaluates options, irrespective of specific technology is, in my opinion, the primary determinant of whether the recommendations we make will be successful. A secondary determinant is the organization's recognition of the work it must undertake to make our recommendations feasible.

The successful consultant makes the determination early on where the organization's maturity level is. It's through this recognition that consulting delivers value, truly a partnership wherein both parties exert equal effort. And quite often, before the recommendations may be implemented, there may be other preparatory work required that may require the work of other consulting organizations. Successful consultants gladly pass on clients that don't understand or appreciate that partnership equation.

John V. Petersen

CODE COMPILERS

Sep/Oct 2022
Volume 23 Issue 5

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
Joydip Kanjilal
Sahil Malik
Jeremy Miller
Paul D. Sheriff
Vassili Kaplan
Peter Mbanugo
John Petersen
Shawn Wildermuth

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

On Consulting and Organizations

Many articles have been written about modern design, architectures, and languages. What about modern consulting and organizations? By "modern," I mean to convey a notion of what's needed today, perhaps at the expense of what would have been regarded as sound practice yesterday. Just as we must continually examine and refine our technical platforms, so too must we do the same for how we approach problem solving. If the ends are the technical solutions we build, then the means to that end is found in consulting services.

Consulting, in recent years, has become a loaded term, meaning different things to different people. At one end of the spectrum is the journeyman contract programmer. At the other end of the spectrum are global firms like KPMG and PWC that offer a wide range of services. Both ends of the spectrum are referred to as consultants. Is such broad application of the term "consultant" correct? The answer entirely depends on the services provided.

If we're providing advisory services such that requested recommended courses of action are the product, then yes. But if all we're doing is providing labor, then no. In other words, if all we're doing is slinging code at the client's direction, that's staff augmentation, not consulting. The client has defined the problem and the solution. The need we're filling in such cases is the labor to implement the solution. If at any point, advice is sought in **how** to solve a given problem, that's consulting. The latter presents a case where we bring our experience and skill to render **affirmative recommendations** for the given context.

The water's edge to good consulting cannot just be "it depends." With consulting, the primary work product includes the recommendations and the basis for making them. Such recommendations must be grounded, actionable, and feasible. By grounded, I mean based on the reality of constraints.

Anything that isn't grounded is the stuff of aspirations. Aspirations are good things because they present the better place we wish to be in the future. Once upon a time, I aspired to be a lawyer. To be a lawyer, it meant I needed to pass the bar examination. To pass the bar examination, it meant I had to take the bar examination, which meant I had to be eligible to take the exam. To be eligible to take the bar examination in most jurisdictions, I must have graduated from an accredited law school. In other words, to achieve that aspirational goal, three things were required: a plan, time, and work.

Think of the last time your customer or employer decided **we want to be Agile**, without any real idea of how to get there. Strategy without tactics or a plan may very well be aspirational. But without any appreciation for the work required to create the plan and work the plan, the aspiration is nothing more than a pipe dream.

A consultant's job, in part, is to steer clear of pipe dreams. I'm often reminded of the following passage in The Pragmatic Programmer by Andrew Hunt and David Thomas:

A tourist visiting England's Eton College asked the gardener how he got the lawns so perfect. "That's easy," he replied. "You just brush off the dew every morning, mow them every other day, and roll them once a week." "Is that all?" asked the tourist. "Absolutely," replied the gardener. "Do that for 500 years and you'll have a nice lawn, too."

A responsible consultant honestly conveys to their client their obligations as freely as the benefits the client hopes to realize. Advice, as great and as well thought out as it may be, will only be as good as the client's ability to implement that advice.

Good Consulting Starts with KYC

KYC stands for "know your client." How are they organized? What are their values? What are their strengths and weaknesses? How committed are they to achieving their aspirational goals? How open are they to change? To know your client means to get into the weeds and embed with them. If there's a shop floor, don a hardhat and walk the floor. For every line of code proposed to be written, will you and your team know how that code furthers the organization's goals and objectives?

Of course, every client organization isn't a monolith. Understanding what the organization does by the "Collective whole" is only one part of the equation. How organizations run, that's a function of their people.

Organizations, after all, are run by people, each with their own agendas, strengths, weaknesses, biases, and opinions. How often have you been confronted with the tension when the person you need to work with is worried that your consulting engagement is a threat to their job? The two things you must accept in such cases are that people are going to believe what they believe and it isn't the consultant's job to fight that battle. All you can do is carry out the engagement with fidelity and professionalism.

To that end, we must **know and understand the client**. A good first step toward that knowledge is in knowing and understanding how the organization is led. There's an old saying that a fish rots from the head down. The same can be said of an organization, which can only be as good as its leadership. Taking a standards-based patterns and practices approach, I employ the following model and then apply the client's specifics to the model:

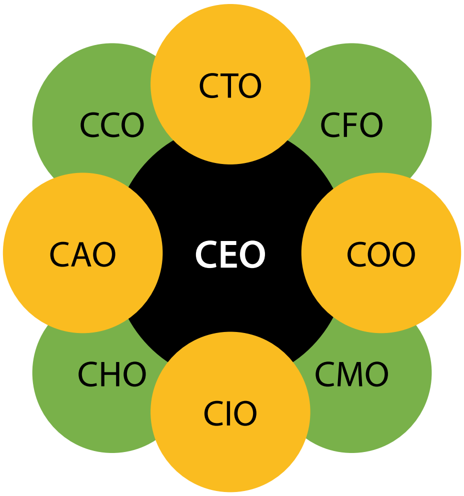


Figure 1: A standard "C-suite" model

The model is my conception of the prototypical C-suite, in terms both of roles and in relationships to one another. If we're to rely on the empirical evidence of past engagements and then apply that experience to the current engagement, there must be a constant. In our technology solutions, constants are patterns and practices around coding, design, and architecture. We apply those

(Continued on page 73)

CODE
CONSULTING
TICKET
1 HOUR FREE CONSULTATION
GET YOUR
FREE HOUR

shutterstock/Lucky-photographer