MS Graph, Power BI, Insomnia, Python, C#

CODE

JUL
AUG
2021

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

Title

Title
Text Title

Title
Text Title

Title
Text Title

0 7>
7 25274 56130 8

DevInte

rsection

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. Bill Me option is available only for US subscriptions. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

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

LEAD Tools

Don't Go Down with Sunk Costs

It all started with a comment from a client, "I can definitely help work on these reports when we go live." That single statement caused me to throw away a significant amount of work on a current project that we're planning on shipping this summer. This comment made a huge amount of sense, as reports are the lifeblood of many companies (including this one) and we don't want to be in the way of that company progressing.

When I heard this comment for the first time, I thought to myself "No problem. We can show the client how to create new reports and compile them into our code base." That was fine, until yesterday! Yesterday I was fighting the report writer component we'd chosen for our application and after an hour of battling this tool, I threw in the towel. I thought to myself, "If I'm fighting this, our client has zero hope of success. It's time to rethink this idea."

Yesterday I made the decision to throw away the reporting work we'd done to date and moved our reports into SQL Server Reporting Services (SSRS). I proceeded to install SSRS and the requisite Visual Studio tooling so I could begin the migration for the first report we targeted. By the end of the day, I'd migrated the bones of the report over and I finished wiring this report into our application in the early hours of today. I felt a huge burden lifted from my shoulders.

As you may have come to expect, there's an important lesson in this brief tale of changing out reporting solutions. The important lesson here is that we're avoiding the sunk-cost fallacy.

Individuals commit the sunk cost fallacy when they continue a behavior or endeavor as a result of previously invested resources (time, money, or effort) (Arkes & Blumer, 1985). This fallacy, which is related to loss aversion and status quo bias, can also be viewed as bias resulting from an ongoing commitment. Citation: https://www.behavioraleconomics.com/resources/mini-encyclopedia-of-be/sunk-cost-fallacy/

The world seems hell-bent on fully implementing the sunk-cost fallacy and this axiom is what leads to costly project overruns. In the case of software development, the sunk-cost fallacy has the tendency to cause serious long-term technical debt. In the words of the legendary basketball player Dikembe Mutombo, "Not in my house." At this stage of development, we decided that the short term sunk-cost we've incurred vastly outweighs the long-term stability and maintainability of our project. After thinking about this a bit more, I decided to think about what types of decisions developers make daily that have the potential for either savings or loss. Here are a few…

In the Interest of Time

Sometimes developers take shortcuts in order to "just get something out the door." This is known as In the Interest of Time coding. See, that phrase has an accurate acronym--ITIOT. I pinged my "brain trust" to give me ideas and my senior developer Greg said this:

> In software development, I often have to make the decision to "just make it work" or make it "work for the next guy." I've completely changed everything to make the code "make sense" to the next developer that might have to look at the code. That dev might just be me!

There are no quick fixes and that code that you "spiked" will live on much longer than you can ever anticipate. If you ever find yourself doing ITIOT coding, make sure you go back and make the proper fix as soon as possible. Pay off that quick loan ASAP.

Getting Stuck on Features is a Smell

Over 30+ years of writing code, I've learned to recognize a smell when I get stuck on a implementing a feature. This smell tells me that the feature is poorly defined, it's an incorrect solution, or it just can't be implemented as requested. Ignoring this type of smell is a common source of long-term technical debt. If it was difficult to implement, it will likely be difficult to maintain.

In nearly all cases when I encountered this smell, I returned to the client or stakeholder with the issue and, more often than not, a better solution was derived, and we were able to get past the "stuck-ness." Sometimes a 15-minute phone call can save countless hours of development.

Instrumentation is Key

A few years back, my team was responsible for re-writing a website for a credit card company. When we deployed the website, we had several stability issues that were tough to batten down. Luckily for us—and I mean LUCKY—we discovered a set of telemetry tools that we could "bolt on" to our application to pinpoint bottlenecks and other stability issues. This deployment taught me the importance of having logging and telemetry built into our applications from the beginning. It's amazing how useful a logging system can be when operating an application. Thanks to my bud JVP for reminding me of this important aspect.

No Code Remains Test Code

Sometimes I begin code with the word Test (or Hack or Junk or Spike) in the name of the program, class, or application and often the name sticks with it throughout production. Take a look at this list of function names: getDailyReportFormTest(), getGaugeChecklistFormTest(), getMasonryFormTest(). There are around 30 of these with the word Test() in a code base that's now in its eighth year. Although this is a small technical debt, it's still a debt. Remember: Code lives forever.

I hope that some of these life lessons help you pay down technical debt early or not take the debt on in the first place. These types of decisions may be costly or difficult and or just tough to implement. Be bold and make the call as soon as you can. When it comes to my reporting issue, I've seldom felt better about a development decision. Make the call: You'll feel better when you do.

PS: A big thanks to my brain trust for helping add good ideas to this editorial.

Rod Paddock

Errata

An error was found after our May/June publication in Joydip Kanjilal's article on JWT authentication in ASP.NET Core 5 (https://www.codemag.com/Article/2105051/Implementing-JWT-Authentication-in-ASP.NET-Core-5) and has been corrected in the online version of this article and in GitHub (https://github.com/joydipkanjilal/jwt-aspnetcore).

CODE
CONSULTING
TICKET
1 HOUR FREE CONSULTATION
GET YOUR
FREE HOUR

Let's Talk About Microsoft Graph

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.

If you work in the Microsoft ecosystem, it's hard to ignore something as big as Microsoft Graph. Imagine that you're new to the Microsoft ecosystem and your boss just attended a Microsoft conference, and everyone's talking about Microsoft Graph, as they often are. Your boss feels that your product must integrate with Microsoft Graph because apparently, it'll help people and organizations get more done. That can't be a bad thing, right? It's your job now, as a developer/architect, to figure out this amazing technology called Microsoft Graph. This is where you start hitting the search engines trying to educate yourself.

I tried to find a definition for Microsoft Graph. Wikipedia (https://en.wikipedia.org/wiki/Microsoft_Graph) likes to define it like this: "Microsoft Graph is a Microsoft developer platform that connects multiple services and devices."

I guess that's nice. It's a developer platform, I'm a developer, I'm on some kind of platform, like maybe Java or C#. I'm not 100% sure yet what this means, but I'm intrigued. I narrow my search to Microsoft docs.

I found the following definition in Microsoft's official docs (https://docs.microsoft.com/en-us/graph/overview):

"Microsoft Graph is the gateway to data and intelligence in Microsoft 365. It provides a unified programmability model that you can use to access the tremendous amount of data in Microsoft 365, Windows 10, and Enterprise Mobility + Security. Use the wealth of data in Microsoft Graph to build apps for organizations and consumers that interact with millions of users."

Wow, so this is a developer platform, and a gateway to data and intelligence. As a developer, the line about the unified programmability model makes some sense. But I'm not sure if I want to interact with millions of users. I think I'll need to read up on what this Microsoft 365 thing is that they're talking about. I've heard of Office 365. Did they rename it? More search bingeing is in order.

I land on this definition of Microsoft 365: "A productivity cloud that delivers innovative and intelligent experiences, rich organizational insights, and a trusted platform to help people and organizations get more done."

Well, I'm glad it's a trusted platform to help people and organizations get more done. I certainly wouldn't want an untrusted platform that would help people and organizations get less done.

Gosh! We seem to have a department of confusing documentation at work here. Just tell me already what Graph is and how I integrate with it. I figure that you're just as frustrated, so I wrote this article. If you've never heard of Microsoft Graph, and your boss has the sudden urge to use it, what does it mean to you, the developer?

What is Microsoft Graph?

Before I go much further, I have no doubt that no matter what words I pick to describe Microsoft Graph, someone is going to poke holes into my wordsmithing capabilities. That's okay, this is how I understand Graph. But wait a second! Before I explain what Microsoft Graph is, you need to know what Azure AD and the Microsoft identity platform are. Let's take a short detour into Azure Active Directory and the Microsoft identity platform first.

Azure Active Directory (Azure AD) is a cloud-based identity and access management service. The **Microsoft identity platform** is an ecosystem that's a superset of Azure AD. When you log into Office 365, you're logging into Azure AD. When you log into Microsoft Teams, you're logging into Azure AD. Given how flexible and extensible Azure AD is, there are many variants of the user experience that the end user may see. Some may be on iOS. Some are in the browser. Some are federated to ADFS (Active Directory Federation Services). Some use SAML, some use OpenID Connect. Modern authentication, which is another ill-defined umbrella term, is quite flexible. Azure AD is Microsoft's flavor of modern authentication and so much more.

Now back to Microsoft Graph. Microsoft Graph is a bunch of programmable features that are protected by the Microsoft identity platform and are accessible from any platform. And when I say a bunch of programmable features, it's mainly three: APIs, connectors, and Data Connect.

APIs

APIs are simple REST APIs available at https://graph.microsoft.com. To be fair, there are a LOT of APIs here. A lot of stuff in the Microsoft cloud is available through various endpoints under this URL. You can really go to town with what you can do with these APIs. For example, you can access your mail. You can access your colleague's mail protected by permissions, of course. You can access calendars. You can invite users. You can… I really should stop. You can do a lot and I really mean a lot. Here's the currently released functionality https://docs.microsoft.com/en-us/graph/api/overview?view=graph-rest-1.0 and here's the functionality currently in beta https://docs.microsoft.com/en-us/graph/api/overview?view=graph-rest-beta. Take a moment to glance through the possibilities. You can really program the heck out of the Microsoft ecosystem with MS Graph.

> You can really program the heck out of the Microsoft ecosystem with MS Graph.

These APIs are all protected by the Microsoft identity platform. Let me really simplify this for you: If you've ever called an OpenID Connect-protected API, calling a Microsoft Graph API is exactly the same. In fact, with the SDKs that Microsoft provides, it's even easier. And by SDKs, I mean not just the SDKs that allow you to authenticate to any Azure AD-protected endpoint, such as MSAL, but there are also MS Graph SDKs that really eliminate the friction for you.

As a developer, here's what you need to know. MS Graph has a bunch of REST APIs that you can call from any platform, and Microsoft SDKs make it very easy to use the APIs.

Connectors

Microsoft Graph connectors are how your non-Microsoft systems provide data to the Microsoft cloud. There are a lot of connectors available for Microsoft Graph, as can be seen here https://docs.microsoft.com/en-us/microsoftsearch/connectors-gallery. These connectors make it possible for all those data sources to input data into the Microsoft cloud, and make it available for services such as Microsoft Search. As an example, there's a connector available for MS-SQL. This means that Microsoft search can now make MSSQL searchable. How neat is that?

Data Connect

These APIs and connectors are great. But as an organization invested in the Microsoft ecosystem, your lifeline—your data—is sitting in the cloud. You can call a bunch of APIs, but sometimes you need lots of data to do further insightful work. Maybe you're trying to write an AI model that analyzes your user's emails, for instance.

You can imagine that calling API after API under the users' delegated permission can get cumbersome very quickly. In fact, it's not going to scale at all. It's precisely to get around this problem that Microsoft Graph Data Connect exists. It allows you to work with the data in popular Azure datastores, such as Azure Data Lake or Azure Blob storage. You can then analyze that data using Azure Data Lake analytics or Azure SQL database, and really your imagination is the limit at that point.

I used a particular phrase here: "users' delegated permission." This simply means that you're trying to call the API on the user's behalf. I'll talk more about this shortly. For now, just imagine that if I want to read your email, I must do it on your behalf. That's exactly what the users' delegated permission allows me to do. It allows me to perform an action, such as reading your email, on your behalf. And yes, there's a permission model built around this, which requires something called "consent" from the user. Afterall, if I'm reading your email, you must consent to it, right? Or an administrator can consent on your behalf. But I'm getting ahead of myself here. I'll explain consent shortly.

My point is that in the back of your mind, you might be thinking that for bulk data processing, there must be some kind of controls built inside of Microsoft Graph Data Connect. After all, you don't want anybody reading just anybody's email; you want some controls on this ecosystem. Random access would entirely defeat the purpose of consent. You'll be glad to know that Microsoft Graph Data Connect has a lot of controls built in so an administrator can set up various rules, and exceptions that define exactly what data is exposed when using Microsoft Graph Data Connect.

Application Permissions vs. Delegated Permissions

A very large and key part of Microsoft Graph are the APIs that it exposes. These APIs are protected by the Microsoft identity platform. To understand how you call these APIs, you must have a basic understanding of how APIs work in the Microsoft identity platform.

These APIs are accessible using standard OpenID Connect mechanisms. This means that in order to access this API, you have to pass in an access token. The access token is a string you pass into the Authorization header in your HTTP request to the API. It typically has a whole bunch of information, such as the validity, who it was issued to, a signature, and much more.

A key but optional part of that access token is the user's identity. Pay special attention to this: The user's identity is optional.

Why would the user's identity in an access token be optional?

Think about the various kinds of APIs that Microsoft Graph can expose. I'll give you two examples: I want to read all users' emails versus I want to read the logged-in users' emails.

Reading all users' emails is something that a background process, such as a daemon or a CRON job, would do. Here, the user's identity isn't important, but the application's identity is important. However, if I say that I wish to read the logged-in users' emails, I must know who the logged in users are.

In this example, when I wish to call an API from an application without a signed in user present, such as background services or daemons, I'd use an application permission.

In contrast, when I wish to call an API from an application where the signed-in user is present, I'd use a delegated permission. Microsoft Graph leverages another capability of the Microsoft identity platform called "administrator consent." Put simply, certain delegated permissions are low risk and can therefore be consented by non-administrator users. Certain permissions are higher risk and they require administrator consent. All application permissions require admin consent, but some delegated permissions don't require admin consent.

There's another feature of the Microsoft identity platform called consent policies. Consent policies allow you to control this behavior for your organization to a large extent.

Effective Permissions

Before I move off the topic of permissions, there's one more important thing for you to know, and that's the concept of effective permissions. As the name suggests, effective permissions are the permissions that your app has when it makes a request to an API. But there's a key difference in how effective permissions work in delegated permissions and application permissions.

For delegated permissions, the effective permissions of your app are the least privileged intersection of the delegated permissions that the app has been granted by consent and of the privileges of the currently signed in user. In other words, your app when using delegated permissions will never have more privileges than the signed in user.

Application permissions, on the other hand, are simply the permissions that have been consented to for the application.

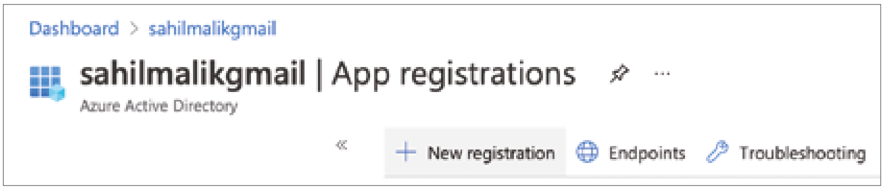

Dashboard > sahilmalikgmail
sahilmalikgmail | App registrations
Azure Active Directory
New registration
Endpoints
Troubleshooting

Figure 1: Create a new app registration.

How Do You Call Microsoft Graph?

I think you already know the answer to this. You call a REST API. If only things were this simple! Because the APIs are authenticated, you first need to get an access token. As you can imagine, the access token has permissions to **do** something versus not being allowed to do something else. Also, the access token may or may not have user identity. Then there is the whole idea of throttling, and Graph SDKs, that I won't get into in this article.

There are a few distinct patterns emerging here. I think I'll need an access token. To get an access token, I'll also need to somehow specify what permissions the access token will have. These are specified in an app registration, which is a concept unique to the Microsoft identity platform, although other OpenID Connect platforms have similar equivalent concepts as well.

This means that there are three distinct steps:

- Set up an app registration.
- Set up permissions.
- Get an access token.

With the access token, I can call Microsoft Graph.

Set Up an App Registration

To call Microsoft Graph, or, for that matter, any API, your application must be granted permissions to call that certain API. In other words, Azure Active Directory needs to know about your application. This process of informing Azure Active Directory everything about your application that Azure Active Directory needs to care about so it can provide authentication services to it is called **app registration**.

As you can imagine, an app registration contains a lot more information than just permissions. For example, in an application that users are signing into, you need to know a reply URL. This reply URL is the whitelisted URL to which Azure Active Directory will send the tokens. Perhaps your application logic depends on certain claims. Your app registration contains information about the claims your app expects.

For my purposes here, let's go ahead and set up an Azure Active Directory app registration. Because I intend to target both delegated permissions, which run on the user's behalf, and applications permissions, which run on the application's behalf, I'll configure the app registration accordingly. In other words, the app registration must allow for an application to sign in and for a user to sign in.

It's possible to set up an application registration through the Azure portal, Azure CLI, or through PowerShell. For the purposes of this article, I'll stick with the portal.

To register an application, go ahead and visit portal.azure.com, and navigate to the Azure Active Directory link on the left-hand side. On this page, under the Manage section, look for **App registrations** and click **New registration**, as shown in **Figure 1**.

Clicking that button shows you a form where Azure Active Directory asks you for the basic information it needs to create this app registration. Specifically, you'll be asked for three bits of information.

First, you'll be asked for the name of this application. This is a name that should make sense to the administrators and the users. For example, this could be "fancy email application". Go ahead and provide the name "MSGraphClient".

Second, it asks you what kind of accounts can sign into this application. There are four choices here.

- A single tenant application allows an account from the current Azure Active Directory to sign in. This is the simplest choice, and this is what I will pick in this scenario. I'll briefly describe what the other choices do as well.
- The second choice is to make the application multitenant. This means that users from other Azure active directories can also sign into my application. Of course, this is controlled by consent.
- The third choice is that you can allow either multitenant users, or Microsoft accounts to use your application. It's worth noting that there may be significant API differences when you're signing in using a Microsoft account versus an Azure Active Directory account. For example, not every feature is exposed to personal Microsoft accounts.
- Finally, you can choose to target your application to only personal Microsoft accounts.

The third question that Azure Active Directory asks you during an app registration is an optional question. It asks you for one or more redirect URLs. This is a whitelisted list of URLs to which Azure Active Directory can send the tokens. Note that these tokens wield a lot of power and responsibility. An access token is like cash: If you find it lying on the ground and you pick it up, it's yours. Therefore, Azure Active Directory must be very careful where to send these access tokens. Depending upon the specific OpenID Connect grant you use, this may be sent either as the result of a post request or posted by Azure Active Directory to a whitelisted URL. Therefore, Active Directory must ask you for redirect URLs so it knows which URLs are safe. Additionally, it ties down what you can specify in that redirect URL, such as that it must be HTTPS and it can't use wildcards. I'll configure this later, so just leave this blank for now and click the register button.

I just mentioned that an access token is like cash. It belongs to whoever discovers it. There is, however, another standard called **proof of possession**, that proves that the requestor who requested the access token is the one sending the access token. This effectively makes the access tokens a little more secure because it gives the called API confidence that this access token wasn't stolen. Further discussion about this is out of scope for this article.

Once the app is registered, you're shown a bunch of information. A key information here is a GUID called the Client ID, sometimes also referred to as the Application ID. Note that this is different from the Object ID. Note down the Tenant ID and Client ID shown on this screen. The Tenant ID is a GUID that represents your Azure Active Directory and is shared by all app registrations registered in your Azure Active Directory.

Set Up Permissions

With my app registration in place, let's add some permissions to it. Specifically, I wish to inform Azure Active Directory that my app or the user using my app has the ability to call certain APIs. To do so, visit the Azure portal again, and under Azure Active Directory, look for App registrations and locate your MSGraphClient app again. Go ahead and click on it. You'll note that there's a menu on the left-hand side, as shown in **Figure 2**.

Here, click on API permissions. You'll notice, as shown in **Figure 3**, that there's already a permission called User.Read added for your app. This allows the app to read the absolute basic profile of the user. To be precise, you don't need this permission in an app registration. You could go ahead and remove it, and the app registration is still valid. However, this permission lets you read the absolute basic profile information of the user that signs in. It's innocuous, so leave it alone.

There are several other interesting things going on in **Figure 3**. For example, notice that permissions are grouped under Microsoft Graph. Microsoft Graph is just one of the APIs, incidentally, authored by Microsoft, that's available for you to use. There are several other APIs available for you as well. In fact, you can author your own APIs, and then allow your own applications to be able to call those APIs. This is out of scope for this article.

Also notice that under the type column, it's clearly specified which permissions are application permissions and which are delegated permissions. In this case, the out-of-the-box permission of User.Read is a delegated permission. Can you guess why that's a delegated permission? Well, you're reading the profile of the logged-in user. This requires you to have the user's identity. Therefore, it's a delegated permission. Additionally, there's a simple description for the permission, and there's another interesting column called admin consent required. Now this is interesting because not all permissions are equal. Some of them are a little more sensitive than others. For permissions that are deemed slightly more sensitive in nature, the logged-in user consenting must be an administrator. And finally, there's the status column. The status column displays consent grants. That's best explained with an example, when you add a new permission.

At the top of **Figure 3**, you should see an **Add a permission** button. Go ahead and click on it. This should open a pane on the right-hand side and Microsoft Graph should appear prominently on the top. Additionally, there are a number of other APIs that Microsoft exposes that should also be visible in that same pane. If you've authored custom APIs, there should be a tab called **My APIs** available over there as well. For now, go ahead and click on Microsoft Graph, and you should be prompted to pick between delegated permissions and application permissions. For the purposes of this article, I'll work with both Delegated and Application permissions. This is a great way to see both in action. You already have a delegated permission called User.Read added, so you need to add an additional application permission. How about a permission that lets me read all users' profiles? Go ahead and click on application permissions tab, and under the user category, look for a permission called User.Read.All. Select it and click on the **Add Permissions** button. Your app registration's permissions should now look like **Figure 4**.

Notice anything strange in **Figure 4**? Under the status column, there's a "not granted for" status written for the newly added permission. When you clicked on the add permissions button, you didn't actually add the permission. You merely requested for the permission to be added. In order to actually add the permission and make it usable, you need to grant consent. There are two ways to grant consent. One way is to click on the grant admin consent button at the top of **Figure 4**. This is necessary for application permissions because application permissions, when called from a headless process, don't have the ability to show an interactive user interface in order to grant consent. The other option is that the user during the log-in process can grant consent. For the purposes of this article, while you are in the Azure

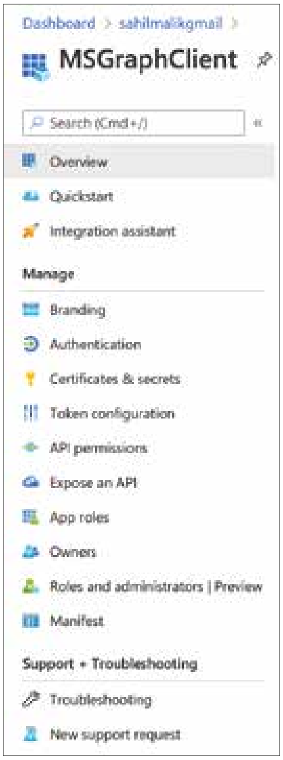
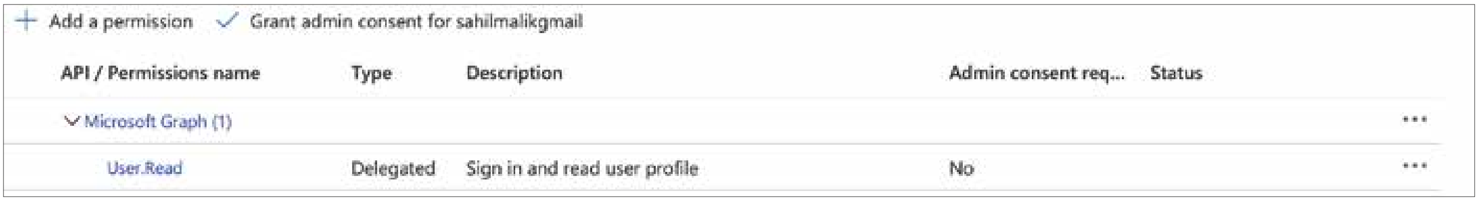
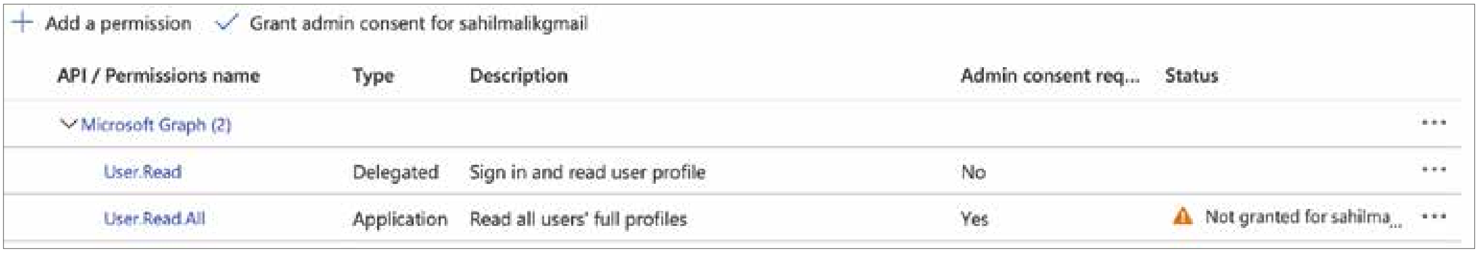


Figure 2: The management menu of an App Registration



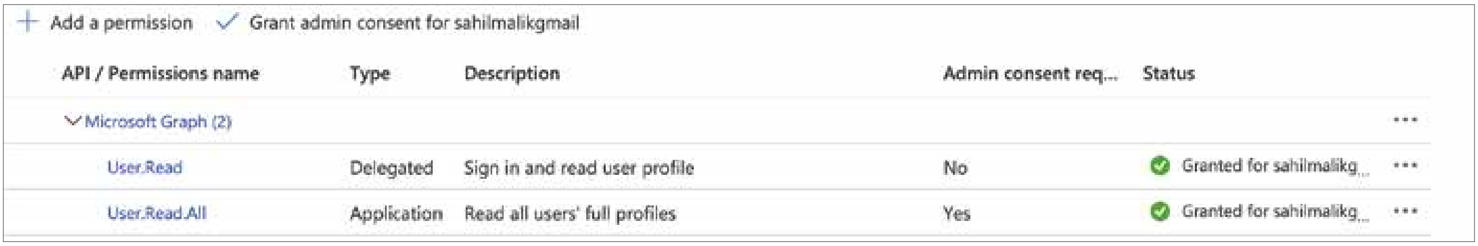
API / Permissions name	Type	Description	Admin consent req...	Status
˅ Microsoft Graph (1)				
User.Read	Delegated	Sign in and read user profile	No	

Figure 3: The permissions of an app



API / Permissions name	Type	Description	Admin consent req...	Status
˅ Microsoft Graph (2)				
User.Read	Delegated	Sign in and read user profile	No	
User.Read.All	Application	Read all users' full profiles	Yes	⚠ Not granted for sahilma...

Figure 4: A newly added permission



API / Permissions name	Type	Description	Admin consent req...	Status
∨ Microsoft Graph (2)				···
User.Read	Delegated	Sign in and read user profile	No	✓ Granted for sahilmalikg... ···
User.Read.All	Application	Read all users' full profiles	Yes	✓ Granted for sahilmalikg... ···

Figure 5: Consent granted

portal, go ahead and click on the grant admin consent button that you see on top of **Figure 4**. Verify that the consent is granted. This can be seen in **Figure 5**.

At this point, the application has the necessary permissions. But before I go much further, I think it's worth mentioning another animal called the Azure AD Graph. The permission you added, for instance User.Read, is shorthand for https://graph.microsoft.com/User.Read. Although that reads like a URL, it's merely a unique string that allows you to identify a permission. It's mere co-incidence, and perhaps some planning on Microsoft's part, that Microsoft Graph is also exposed at https://graph.microsoft.com. There exists another User.Read in the Microsoft ecosystem, and that's https://graph.windows.net/user.read. Pay close attention: One ends in **.com** and the other ends in **.net**. Again, these are just unique strings. Graph.windows.net represents AAD Graph, which is on its way to deprecation. You shouldn't use it. Unfortunately, you'll find a lot of parallel permissions between Microsoft Graph and AAD graph, and that could get confusing. But as AAD graph is on its way to deprecation, pay close attention and make sure that you're using Microsoft Graph and not AAD graph.

With that out of the way, it is time to call Microsoft Graph.

Get an Access Token

To call Microsoft Graph, or for that matter any OpenID Connect-protected API, you're going to need an access token. The access token is put in the authorization header in the following format:

```
Bearer <access token>
```

The obvious question is: How do you get an access token that will work with Azure Active Directory? The access token is intended for Microsoft Graph, but it's issued to the application, or the application and user pair. The process of requesting the access token between these two is different. OpenID Connect has various flows that allow you to request access tokens. Some of them work with application identities, and some of them work with application and user identity. Describing each flow is out of scope for this article, but I assure you that it's a very interesting topic and I hope to cover it in a future article.

Listing 1: Requesting an access token

```
curl --location --request POST
 'https://login.microsoftonline.com/<tenantid>/oauth2/v2.0/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'client_id=<guid>' \
--data-urlencode 'scope=https://graph.microsoft.com/.default' \
--data-urlencode 'client_secret=secret' \
--data-urlencode 'grant_type=client_credentials'
```

Listing 2: Call the API

```
curl --location --request GET 'https://graph.microsoft.com/v1.0/users' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer '
```

For the purposes of this article, I'll show you a simple trick to get an access token. I'll show this in two parts. First, I'll call the API under application permissions and therefore request an access token on behalf of the application. Second, I'll call the delegated permission and request an access token on behalf of the user.

Call the Application Permission API

To call an API using an access token with application permission, you'll need to first ask Azure Active Directory for such a token. One way to get this token is using a standard OpenID Connect grant, called the **client credential grant**. Client credential grant is a simple POST request to the Azure Active Directory token endpoint with the following information:

- Who are you?
- What do you need this token for?
- Prove your identity with a credential.

In this scenario, the question of who you are is answered by the identity of the application. In your app registration, you'll see something called the **application ID**. Sometimes we also refer to this as **client ID**. Go ahead and copy this information from your app registration.

In this scenario, the question of what you need this token for is answered with Microsoft Graph. In the case of Azure Active Directory, there's a special scope called **.default**. That's what you need to use with client credential flow. So the scope you're requesting an access token for is https://graph.microsoft.com/.default. Specifying this scope causes Azure Active Directory to return an access token that's valid for all previously consented permissions.

Finally, you can prove your identity by either providing a secret or using a certificate. Although the secret is a simpler choice, a certificate is a safer choice. This is because when you use a secret, you're required to send the secret over the wire. When you use a certificate, you send a string that merely proves your possession of the accurate certificate. It's worth mentioning that Azure Active Directory, when using client credential flow, won't validate expired certificates, or certificates from a certificate authority, or certificate revocation lists. The certificate is merely a credential. To provide a secret, go to your app registration area, and under Certificates and Secrets, go ahead and add a secret. This secret is shown to you only once and you should treat it like a password. In fact, production applications prefer to use managed identity. Where you can't use managed identity, put this secret in a key vault and use managed identity to read from the key vault.

Now that you have a client ID, a scope, and a secret, you can make a simple POST request to get an access token using application permissions, as shown in **Listing 1**.

Sending the request shown in **Listing 1** should return you a JSON object, one of the nodes of which is the access token. Copy and paste that access token, which you'll use to call the API. You can see how to call the API in **Listing 2**.

You can verify that this call returns the users in your organization.

Call the Delegated Permission API

To call Microsoft Graph, or, for that matter, any API under the delegated permission, you first need to obtain an access token that uses delegated permissions. Again, there are many ways to do so, but I'll use something called the auth code flow using PKCE. I'll leave the description of all these deep identity related topics or a future article. For now, just follow along.

At a high level, this flow requires you to request a code. Using that code, you can request an access token. There's some protection involved using something called the **code challenge,** so Azure AD has some confidence that the party requesting the access token is the party that originally requested the code.

First let's request the code. Because there's a user involved here, you'll have to perform this operation in a browser. This allows the user to sign in, and therefore the token will be issued on behalf of the user. This token will be sent back to a whitelisted URL, so in your app registration, under authentication, choose to add a Web application with http://localhost as a reply URL for a Web application. Also choose to enable ID tokens for hybrid flows. This can be seen in **Figure 6**.

Now you need to create a URL for the user to sign in. The URL can be seen in **Listing 3**. There's a lot going on in **Listing 3** and frankly, in most cases, SDKs such as MSAL will abstract it for you. Wherever you have an SDK such as MSAL available, you should use it. But because I'm doing this by hand here, I need to craft up a URL.

In **Listing 3**, you may see a lot of things that may be unfamiliar. I'll leave all those details for a future article where I get to talk about authentication in depth. For now, replace the strings in various place holders, such as the GUID for the Client ID and the Tenant ID, and then open a browser window in private mode and visit the URL from **Listing 3**. That URL should ask you to log in. After you provide your credentials, you're shown an ugly window, like **Figure 7**.

It may look like a request failed from **Figure 7**, but the reality is that the request worked. In a real-world application, you'd have something listening on localhost, or whatever redirect URL you specified in your Web application authentication settings. Because I don't have anything listening there, I'll have to perform this step manually. Copy and paste the entire URL from your browser window that should

Listing 3: The login URL

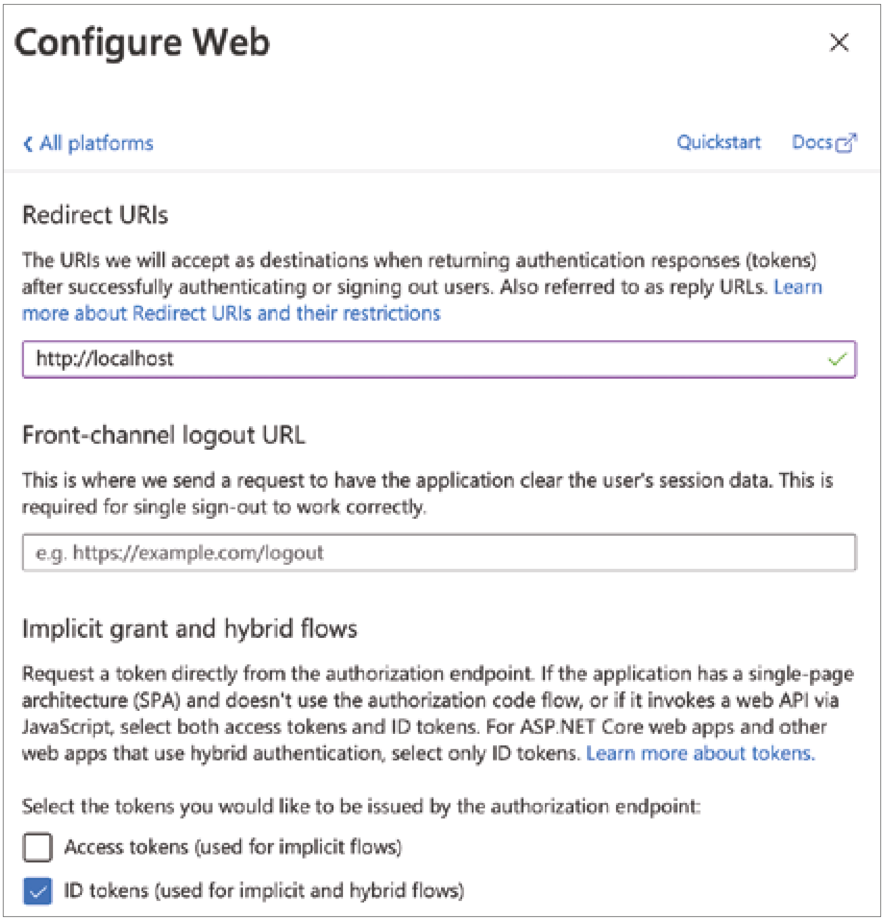
```
https://login.microsoftonline.com/<tenantid>/oauth2/v2.0/authorize?
 client_id=<guid>&
 response_type=code%20id_token&
 redirect_uri=http://localhost&
 response_mode=fragment&
 scope=openid&
 state=1245&
 nonce=abcde&
 code_challenge=n4bQgYhMfWWaL-qgxVrQFaO_TxsrC4Is0V1sFbDwCgg&
 code_challenge_method=S256
```

Listing 4: Requesting an access token

```
curl --location --request POST
 'https://login.microsoftonline.com/<tenantid>/oauth2/v2.0/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'redirect_uri=http://localhost' \
--data-urlencode 'client_id=<guid>' \
--data-urlencode 'grant_type=authorization_code' \
--data-urlencode 'code=<code>' \
--data-urlencode 'code_verifier=thisisasecret' \
--data-urlencode 'scope=https://graph.microsoft.com/.default' \
--data-urlencode 'client_secret=<secret>'
```

Listing 5: Call the API

```
curl --location --request GET
 'https://graph.microsoft.com/v1.0/me\
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer '
```



Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating or signing out users. Also referred to as reply URLs. Learn more about Redirect URIs and their restrictions

http://localhost ✓

Front-channel logout URL

This is where we send a request to have the application clear the user's session data. This is required for single sign-out to work correctly.

e.g. https://example.com/logout

Implicit grant and hybrid flows

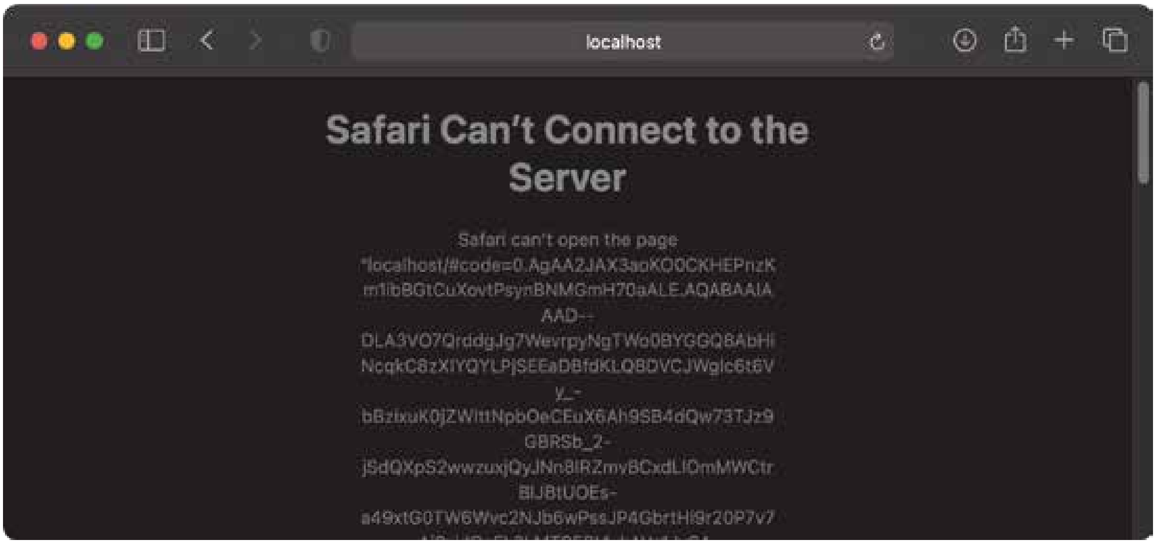
Request a token directly from the authorization endpoint. If the application has a single-page architecture (SPA) and doesn't use the authorization code flow, or if it invokes a web API via JavaScript, select both access tokens and ID tokens. For ASP.NET Core web apps and other web apps that use hybrid authentication, select only ID tokens. Learn more about tokens.

Select the tokens you would like to be issued by the authorization endpoint:

☐ Access tokens (used for implicit flows)

☑ ID tokens (used for implicit and hybrid flows)

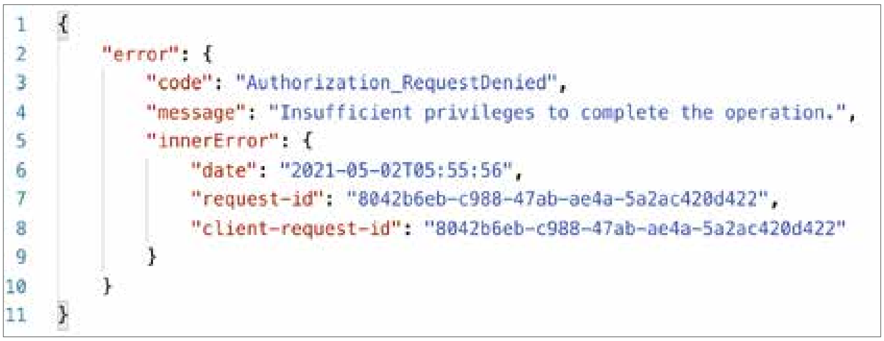
Figure 6: The app's authentication configuration



Safari Can't Connect to the Server

Safari can't open the page "localhost/#code=0.AgAA2JAX3aoKO0CKHEPnzKm1ibBGtCuXovtPsynBNMGmH70aALE.AQABAAIAAAD--DLA3VO7QrddgJg7WevrpyNgTWo0BYGGQ8AbHiNcqkC8zXIYQYLPjSEEaDBfdKLQBDVCJWglc6t6Vy_-bBzixuK0jZWIttNpbOeCEuX6Ah9SB4dQw73TJz9GBRSb_2-jSdQXpS2wwzuxjQyJNn8lRZmyBCxdL!OmMWCtrBIJBtUOEs-a49xtG0TW6Wvc2NJb6wPssJP4GbrtHl9r20P7v7

Figure 7: You got a code.



```
{
    "error": {
        "code": "Authorization_RequestDenied",
        "message": "Insufficient privileges to complete the operation.",
        "innerError": {
            "date": "2021-05-02T05:55:56",
            "request-id": "8042b6eb-c988-47ab-ae4a-5a2ac420d422",
            "client-request-id": "8042b6eb-c988-47ab-ae4a-5a2ac420d422"
        }
    }
}
```

Figure 8: You can't call application permission with a delegated permission token.

look like **Figure 7**. That entire URL should have a bunch of query parameters, and the one you're interested in is **code**. Copy and paste its value. Another interesting parameter here is the **id_token**, which proves the user's identity. If you're interested, go ahead, and visit https://jwt.ms to decode that token. You should see some interesting claims, including the user's identity.

Next, you'll redeem this code for an access token. The request for an access token can be seen in **Listing 4**.

Sending this request should send you back, among other things, an access token. This is a short-lived token that you can now use to make requests to Microsoft Graph. Feel free to decode this token at jwt.ms and verify that it contains the user's identity.

To call MS Graph now under delegated permission, use the code shown in **Listing 5**.

Verify that the request shown in **Listing 5** shows the current user's data. There's a very subtle difference between **Listing 2** and **Listing 5**. The only difference is the URL you are calling. Now let's try and be a little naughty. The access token that I have can call the granted delegated permissions. Let's try to use this access token to call the API from **Listing 2**.

You should see an error, as shown in **Figure 8**.

This makes sense, because the logged-in user doesn't have the ability to read all users' profiles, because you didn't give such a permission consent. If you're curious, there is a delegated permission concern that does allow you to do this. And that's User.Read.All under delegated permissions.

Similarly, you can try the reverse—try calling the **me** endpoint from **Listing 5** with the access token you obtained for application permissions. Because there's no **me**, no logged in user, the request won't work.

Summary

The word "graph" is annoyingly overused in the Microsoft ecosystem and, perhaps the entire tech industry. Graph fatigue aside, let me assure you that Microsoft Graph is a very key part of the Microsoft ecosystem. It will serve you well, so get familiar with it.

Of course, there's a lot more I can say about Microsoft Graph. There are some interesting APIs. There's the whole DevOps aspect. And then there are various tips and tricks you can use—they help you discover new APIs, the right permissions, and just make you more productive in general.

And then there's the authentication bit that you should know about.

More on such topics in future articles. Happy coding.

Sahil Malik

ONLINE QUICK ID 2107031

How to Use the Fetch API (Correctly)

In my last two articles, "Using Ajax and REST APIs in .NET 5" (https://bit.ly/3bRzfJ6) and "Build a CRUD Page Using JavaScript and the XMLHttpRequest Object" (https://codemag.com/Article/2105031/Building-a-CRUD-Page-Using-JavaScript-and-the-XML-HttpRequest-Object), I introduced you to using the XMLHttpRequest object to make Web API calls to a .NET 5 Web server.

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

Paul has been in the IT industry over 34 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 28 courses in the www.pluralsight.com library (http://www.pluralsight.com/author/paul-sheriff) on topics ranging from LINQ, JavaScript, Angular, MVC, WPF, ADO.NET, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

Whether you use jQuery, Angular, React, Vue, or almost any other JavaScript framework to make Web API calls, most likely, they use the XMLHttpRequest object under the hood. The XMLHttpRequest object has been around as long as JavaScript has been making Web API calls. This is the reason it still uses callbacks instead of Promises, which are a much better method of asynchronous programming.

In this article, you'll learn to use the Fetch API, which is a promise-based wrapper around the XMLHttpRequest object. As you'll see, the Fetch API makes using the XMLHttpRequest object easier to use in some ways but does have some drawbacks where error handling is concerned. To make working with the Fetch API a little easier, a set of IIFEs (closures) are created in this article. Using a closure makes your code easier to read, debug, and reuse. You don't need to have read the previous articles to follow this one. However, the .NET 5 Web API project is created from scratch in the first article, so reference that article if you want to learn to build a CRUD Web API using .NET 5.

Download Starting Projects

The best way to learn to use the technologies presented in this article is to follow along and type in the samples. I've created a download with two starting applications, a .NET 5 Web API project, and a Web Server project (either MVC or node). Download these projects at www.pdsa.com/downloads and click on the link entitled "CODE Magazine - How to Use the Fetch API (Correctly)". After downloading the ZIP file, unzip it into a folder where you'll then find three folders. The **\Samples** folder contains the finished samples for both MVC and node. The **\Samples-WebAPI** is the .NET 5 Web API project. The **\Samples-Start** folder contains an MVC and a node project, one of which you'll use to follow along with this article.

In addition to the source code, you also need the Microsoft AdventureWorksLT sample database. I've placed a version of it on my GitHub account that you can download at https://github.com/PaulDSheriff/AdventureWorksLT. Install this database into your SQL Server.

Navigate into the folder **Samples-WebAPI** and load that folder in Visual Studio Code or Visual Studio 2019. Open the **appsettings.json** file and modify the connection string to point to your SQL Server where you installed the AdventureWorksLT database. Run this project and when the browser appears, type in http://localhost:5000/api/product. If you have everything installed correctly, you should get an array of JSON product objects displayed in the browser. Leave the Web API project running as you make your way through this article.

If you're most familiar with **MVC**, navigate into the folder **\Samples-Start\AjaxSample-MVC**. Load this folder into another instance of Visual Studio Code or Visual Studio 2019. Click on the **Run > Start Debugging** menu item to ensure that your browser launches and displays a Product Information page.

If you're most familiar with **node**, navigate into the **\Samples-Start\AjaxSample-Node**. Load this folder into another instance of Visual Studio Code. Open a terminal window and type in **npm install** to load all dependencies on your computer. Next, type in **npm run dev** to start the *lite-server* and display a browser with a blank Product Information page. One thing to note when using the node version is that if you open the **index.html** page in VS Code, you see that it reports four errors. They're not really errors; it's just that VS Code doesn't understand the templating engine you're using. The templating engine is explained later in this article.

Application Architecture

As you read this article, you're going to learn how to put together a CRUD application using the Fetch API. I prefer to show you a more robust, real-world example rather than just a simple sample. To that end, I highly recommend that you create separate .js files as I'm doing in this article, so you have reusable code for any additional pages, and for future applications.

Figure 1 shows you the overall architecture for the application you're going to build. The **site.js** file is used on almost all pages in your site and contains a global *appSettings* object. The *appSettings* object contains properties to hold information that you're going to need for your entire application. I've already placed a few properties in here for you. The most important one is the *apiUrl* property that contains the URL for where your Web API server is located.

```
var appSettings = {
  "apiUrl": "http://localhost:5000/api/",
  "msgTimeout": 2000,
  "networkErrorMsg":
    "A network error has occurred.
      Check the apiUrl property to
        ensure it is set correctly."
}
```

The **ajax-common.js** file contains an Immediately Invoked Function Expression (IIFE) assigned to a global variable named *ajaxCommon*. You put methods into this closure to handle generic Ajax calls and error handling. This file is referenced on any page where you make Ajax calls. The **product.js** file is where you write methods to work specifically with the product page (in this case, the index page). The index page is going to display a table of products, and allow you to add, edit, and delete product information by calling the Web API. If you have more pages, like a customer page or an employee page, create **customer.js** and **employee.js** files, respectively, for the functionality of each of those pages.

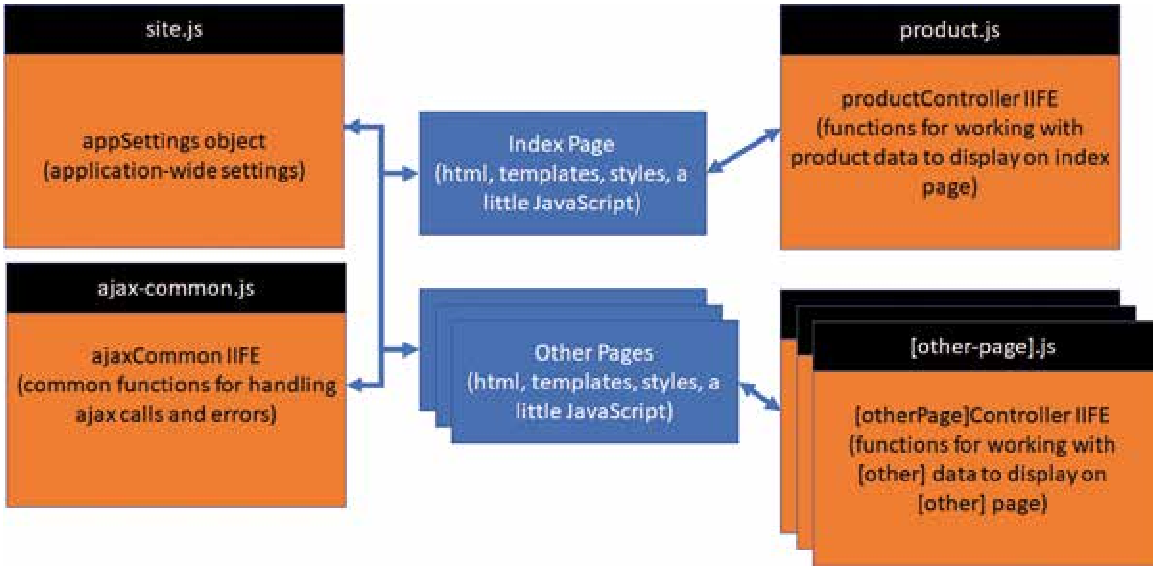

site.js
appSettings object
(application-wide settings)
ajax-common.js
ajaxCommon IIFE
(common functions for handling
ajax calls and errors)
Index Page
(html, templates, styles, a
little JavaScript)
Other Pages
(html, templates, styles, a
little JavaScript)
product.js
productController IIFE
(functions for working with
product data to display on index
page)
[other-page].js
[otherPage]Controller IIFE
(functions for working with
[other] data to display on
[other] page)

Figure 1: A good application architecture separates functionality into different closures.

Set Options in the productController

The *appSettings* object can be used everywhere in your application because it's declared outside of any closure. However, a better approach is to pass values from the *appSettings* object into each closure that needs the settings. This allows you to modify the settings for each page if needed. Open the *product.js* file, located in the **\scripts** folder in the node project and in the **\wwwroot\js** folder in the MVC project, and just below the comment that reads **// Private Variables**, add the following literal object.

```
let vm = {
  "options": {
    "apiUrl": "",
    "urlEndpoint": "product",
    "msgTimeout": 0
  }
};
```

Instead of having multiple variables with a closure, I prefer to create a literal object called *vm,* which stands for "View Model". Within the *vm* object is where you add as many properties as needed for your page. Throughout this article, you're going to add quite a few, but this first one is for the options you wish to pass in from outside of the closure. To set the values in the *vm* object, add a public function within the *return* literal object located at the end of the closure.

```
return {
  "setOptions": function (options) {
    if (options) {
      Object.assign(vm.options, options);
    }
  }
};
```

Adding the setOptions property in the **return** object defines the method setOptions() as a public method that can be called by referencing the *productController* variable. Don't type this in anywhere, but the code snippet below is an example of how you can call this setOptions() method and set one or more of the properties in the *vm.options* property.

```
productController.setOptions({
  "apiUrl": "http://localhost:5000/api/",
  "urlEndpoint": "product"
});
```

The *apiUrl* property is self-explanatory, as it represents the Web API server name where all your controllers are located. The *urlEndpoint* property is what's added on to the end of the *apiUrl* to provide the specific controller within your server to call for this page. If you put these two together, you end up with the URL http://localhost:5000/api/product. Within the *productController,* add one more method called get() that looks like the following code snippet.

```
function get() {
  let msg = vm.options.apiUrl +
            vm.options.urlEndpoint;
  msg += " - ";
  msg += JSON.stringify(vm.options);

  displayMessage(msg);
}
```

The code above is going to allow you to display the data within the *vm.options* object so you can ensure that your setOptions() method is working as you expect it to. Add the get() method to the *return* object to expose this as a public method from the *productController* variable.

```
return {
  "setOptions": function (options) {
    if (options) {
      Object.assign(vm.options, options);
    }
  },
  "get": get
};
```

Display Messages

Near the top of the index page, you'll find two <div> elements defined with bootstrap row and column classes, as shown in the following code snippet. Within the <div> elements are two labels; one to display informational messages and one to display error messages. Both labels are styled using a style from the **site.css** file in the project. They're both also styled with **"d-none"**, which is a bootstrap class to make each of these labels invisible.

```
<div class="row">
  <div class="col">
    <label id="message"
           class="infoMessage d-none">
    </label>
    <label id="error"
           class="errorMessage d-none">
    </label>
  </div>
</div>
```

Create a new method named displayMessage() in the *productController* closure to write text into the message label when you want to display an informational message to the user. If a message is passed in, the message is set into the label's text area, and the label is made visible by removing the class **"d-none"**. If an empty message is passed in, the label is hidden by adding the **"d-none"** class to the label.

```
function displayMessage(msg) {
  if (msg) {
    $("#message").text(msg);
    $("#message").removeClass("d-none");
  }
  else {
    $("#message").addClass("d-none");
  }
}
```

Open the **index.cshtml** or the **index.html** file and at the bottom of the page, modify the *window.onload* function to look like the following code snippet. You're building a literal object with only that set of properties you want to update in the *vm.options* property within the *productController* closure.

```
window.onload = function () {
  productController.setOptions({
    "apiUrl": appSettings.apiUrl,
    "msgTimeout": appSettings.msgTimeout
  });
  productController.get();
}
```

Try It Out

Save all your changes and run the project to display the index page. You should see the full URL to the product controller being displayed in the <label> on the index page. You also see the *options* property with its properties displayed. This proves that the setOptions() method did set the *options* property correctly when called from the index page.

```
http://localhost:5000/api/product -
{
  "apiUrl":"http://localhost:5000/api/",
  "urlEndpoint":"product",
  "msgTimeout":2000
}
```

Getting Started with the Fetch API

The fetch API is similar to using the jQuery's $.ajax() method. You make a request to a Web API endpoint and a promise object is returned in either a fulfilled or a rejected state. In the get() method in your *productController*, modify the get() method to look like the following.

```
function get() {
  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint)
    .then(response => response.json())
    .then(data =>
      displayMessage(JSON.stringify(data)))
    .catch(error => {
      displayError("*** in the catch()
         method *** " + error);
    });
}
```

This code uses the fetch() function to make a call to the Web API Product controller class. When the Ajax call is fulfilled, the *response* parameter is passed to the first .then() method. Extract the body of the *response* object using the .json() method. The result from calling the .json() method is an array of product objects retrieved from the Web API. This array is passed to the second .then() method as the *data* parameter. Within the second .then() method is where you do something with the data, such as display it in an HTML table, or fill in a drop-down list. For now, you're just putting that data into the informational message label.

The .catch() method from the fetch() function is called when a network error occurs while attempting to make the Web API call. In the .catch() method, call a method named displayError() that can display an error message in the error label. In the *productController* closure, type in the code shown below.

```
function displayError(msg) {
  if (msg) {
    $("#error").text(msg);
    $("#error").removeClass("d-none");
  }
  else {
    $("#error").addClass("d-none");
  }
}
```

As shown previously, there's a label with an **ID** of error. When you receive an error, display that error in this label, as it's styled with a red background and white lettering, so it stands out to the user. If an error message is passed in, the message is set into the label's text area, and the label is made visible by removing the class **"d-none"**. If an empty message is passed in, the label is hidden by adding the **"d-none"** class to the label.

Try It Out

Save the changes and run your project. You should see an array of product objects displayed in the message label.

The Fetch API Exception Handling is Erratic

The Promise object returned by fetch() doesn't reject an error when an HTTP error status is returned (400 or greater) like most normal APIs do. For example, 400 and 500 status codes don't cause a rejection, but a 404 may or may not cause a rejection of the promise. A network failure, a CORS error, and a few other types also cause a rejection. This section of the article illustrates each of these scenarios. To force the get() method to go into the .catch() method, remove one of the zeros (0) from the port number 5000 in the *apiUrl* property in the *appSettings* object located in the **site.js** file.

```
http://localhost:500/api/
```

Save your changes and run the project. Open your browser tools (F12) to get to the console window and you should see an error message that looks like the following, if you're using the Chrome browser.

```
Failed to load resource: net:
  :ERR_CONNECTION_REFUSED
```

In the error message label on the index page, you should see something that looks like the following message.

```
*** in the catch() method ***
TypeError: Failed to fetch
```

Because the port number doesn't exist, a network error is detected by the Fetch API. Because the fetch() function is unable to reach the Web API server, the .catch() method is called.

Another way to cause a rejection of the promise is to get a 404 (not found) status code returned from the Web API server. Put the *apiUrl* property back to the normal port number of 5000. In the get() function, add on a "/9999" to the fetch() call.

```
fetch(vm.options.apiUrl +
      vm.options.urlEndpoint + "/9999")
```

Save your changes and run the project. Open your browser tools to get to the console window and you should see the following error message reported:

```
Failed to load resource: the server responded
   with a status of 404 (Not Found)
```

In the error message label on your page, you should see the following message:

```
*** in the catch() method ***
SyntaxError: Unexpected token C in JSON at position 0
```

The product ID of 9999 doesn't exist in the database, so the Web API server returns a 404 status code with the text "Can't find Product with ID=9999". So, why did you end up in the .catch() block? After all, you did get to the Web API server, so it wasn't a network error. If you look at the error message returned by the Fetch API, it says there was a SyntaxError. The problem is that what's returned by the 404 isn't JSON data, but a simple text string. When you call the response.json() method on a text string, an exception is thrown because trying to perform a JSON.parse() on a text string causes control to go to the .catch() method. Don't worry, you're going to learn how to solve this problem in the next section of this article.

What happens in the case of a bad request (400) or an internal server error (500), or any of the many other HTTP error status codes? The answer is that you won't know until you try each one. I'm going to show you how to create a method to handle the most common errors and display what it's appropriate. Let's take a look at a 400 status code. Add a "/a" to the end of your URL in the get() method as shown below:

```
fetch(vm.options.apiUrl +
      vm.options.urlEndpoint + “/a”)
```

Save your changes and run the project. Open your browser tools to get to the console window and you should see the 400 error message reported by your browser.

```
Failed to load resource: the server
responded with a status of 400 (Bad Request)
```

In the error message label on your page, you should see a JSON object that looks like the following.

```
{
  "type":"https://tools...",
  "title":"One or more validation
           errors occurred.",
  "status":400,
  "traceId":"00-…",
  "errors": {
     "id": ["The value 'a' is not valid."]}
}
```

Notice that you did not go into the .catch() method, but instead ended up with the literal object reported by the second .then() method. As you can see, trying to handle errors using the Fetch API can be very confusing because it seems very random what type of error calls the catch, and which ones try to process the response object passed back.

Exploring the Response Object

To help with handling exceptions while using the Fetch API, you need to learn more about the *response* object that you see in the first .then() method. When you get the *response* object in the first .then() method, there are a few properties that are important to you; *ok*, *status*, and *statusText*. If the call is successful, the *ok* property is set to a true value, the *status* property is set to the HTTP status code, and the *statusText* property is set to the corresponding message of the status code. For example, if the *status* property is set to 200, the *statusText* property is set to "OK".

If the *ok* property is set to false, this means the call failed for some reason other than a network error. The *status* and *statusText* properties are still set with the corresponding HTTP status code and message. Depending on the HTTP status code, you're going to use two different methods to retrieve the data associated with that status code. For example, if you receive a 404 status code, use the response.text() method to retrieve the actual text message sent back from your Web API controller. If you receive a 400 status code, use the response.json() method to retrieve a JSON object filled with additional properties about what went wrong. For a 500 status code, use the response.json(), however, you'll then find the actual message returned from the Web API method within the *message* property on the object returned.

So with this in mind, re-write your get() function (**Listing 1**) to return response.json() if the *ok* property is true, and return response.text() if the *ok* property is false. It's better to get the text version of the data when *ok* is false so you don't cause an error attempting to convert the response to JSON when it could be text. If you return text data to the second .then() method, you can always parse it to JSON depending on the number in the *status* property. Let's test getting each of the various successful and error codes you looked at previously.

Get a 200 Status

Change the *apiUrl* property back to "http://localhost:5000/api/". Modify the get() function to look like the code shown in **Listing 1** and save your changes. Run the project and you can see the array of product objects displayed in the message label.

Get a 404 Status from Your Web API Method

Change the fetch() call in the get() method to add to the URL a "/9999". The value "9999" is an invalid product ID so the Web API server returns a 404 (Not Found) status code.

```
fetch(vm.options.apiUrl +
      vm.options.urlEndpoint + “/9999”)
```

Save the change, run the project, and you should see the text "Can't find product with ID=9999." displayed in the message label. This message is being returned from the Web API Get(int id) method.

Get a 404 Status from a Non-Existent API Endpoint

The other kind of 404 status code is when you call an API that doesn't exist. Change the fetch() call in the get() method to look like the following code snippet.

```
fetch(vm.options.apiUrl + “prod/9999”)
```

Save the changes and run the project. Running this fetch() function produces an empty string in the message label. You're going to learn how to take care of this in the next section of this article.

Get a 400 Status

The next status code to test is a 400 (Bad Request). You can force this error to occur by passing in a letter to the Get(int id) method. Because the Web API method doesn't accept a letter, submitting this on your URL line causes the 400. Change the call to the fetch() function to look like the following.

```
fetch(vm.options.apiUrl +
      vm.options.urlEndpoint + “/a”)
```

Save the changes, run the project, and you should see text that looks like the following in the message label. The return value is a JSON object, but it's being reported as text.

```
{
  "type":"https://tools...",
  "title":"One or more validation
          errors occurred.",
  "status":400,
  "traceId":"00-…",
  "errors": {
     "id": ["The value 'a' is not valid."]}
}
```

From just the few status codes you tried here, you can see that the code in **Listing 1** handles only the 200 and 404 calls correctly. Of course, these two status codes are about 95% of use cases for a typical business application. However, to be complete, you should also handle 404 for non-existent API endpoints, 400 for bad requests, and 500 for other exceptions that might be thrown by the Web API server.

Create Helper Functions

To handle the various status codes returned by the Fetch API, it's important to preserve a few properties from the *response* object so you can check them when you get into the second .then() method. To accomplish this, add a new literal object to the *vm* object in the *productController*. Create a property called *lastStatus* just below the *options* property you added earlier in this article.

```
let vm = {
  "options": {
    "apiUrl": "",
    "urlEndpoint": "product"
  },
  "lastStatus": {
    "ok": false,
    "status": 0,
    "statusText": "",
    "response": null
  }
};
```

The *ok* property is set to either a true or false value. The *status* property is set to the HTTP status code (200, 404, etc.) from the last request. The *statusText* property is set to the text that goes along with the HTTP status code such as "OK" or "Not Found". The *response* property is populated in the second .then() method as you're going to see in the next code listing.

Add the processResponse() method shown below to the *productController*. In this method, copy the properties from the *response* object into the properties of the *lastStatus* object. Because the *lastStatus* object is created outside of any methods within *productController*, this object is available to all methods. Once you have the properties set, check the *ok* property to determine if you should return the results from response.json() or response.text() back to the first .then() method.

Listing 1: Check the response.ok property to determine if the Ajax call was successful or not.

```
function get() {
  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint)
    .then(response => {
      if (response.ok) {
        return response.json();
      }
      else {
        return response.text();
      }
    })
    .then(data =>
      displayMessage(JSON.stringify(data)))
    .catch(error => {
      displayError("*** in the catch()
                    method *** " + error);
    });
}
```

```
function processResponse(resp) {
  // Copy response to lastStatus properties
  vm.lastStatus.ok = resp.ok;
  vm.lastStatus.status = resp.status;
  vm.lastStatus.statusText = resp.statusText;
  vm.lastStatus.url = resp.url;

  if (vm.lastStatus.ok) {
    return resp.json();
  }
  else {
    return resp.text();
  }
}
```

When working with the Fetch API response object, you need to be aware that once you've processed the body of the *response* object using the .json() or the .text() methods, you can't read the body again. This is a one-time operation. Modify the get() function to look like **Listing 2**. In the first .then() method, you pass the *response* object to the processResponse() method you just created. In the second .then() method, either the JSON or the text data is passed into the *data* parameter. The first thing you should do is to assign the *data* parameter into the *response* property of the *lastStatus* property. This preserves the original data in case it's needed.

If the *lastStatus.ok* property is true, you do something with the data returned from the Web API. For now, you're just going to display it into the message label. Later in this article, you're going display that product data in an HTML table. If the *lastStatus.ok* property is false, call an ajaxCommon.handleError() method passing in the *lastStatus* object. You're going to write the handleError() method shortly. If the .catch() method is called because of a network error, pass the error object to an ajaxCommon.handleAjaxError() method that you're going to write soon.

It's now time to add the two methods handleAjaxError() and handleError() into the *ajaxCommon* closure. Open the **ajax-common.js** file and just below the **// Private Functions** comment block in the *ajaxCommon* closure, create the handleAjaxError() method as shown below. Because the .catch() method is only called when something catastrophic happens, this code is going to assign the generic error message from the *appSettings* object to the variable *msg*. It then logs the *error* parameter and the *msg* variable to the console. The *msg* variable is returned from this method so you can display it in the error label if you wish.

```
function handleAjaxError(error) {
  let msg = appSettings.networkErrorMsg;

  console.error(error + " - " + msg);

  return msg;
}
```

Listing 2: Create separate functions to handle processing the response, and handling errors.

```
function get() {
  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint)
    .then(response =>
      processResponse(response))
    .then(data => {
      // Fill lastStatus.response
      // with the data returned
      vm.lastStatus.response = data;

      // Check if response was successful
      if (vm.lastStatus.ok) {
        displayMessage(JSON.stringify(data));
      }
      else {
        displayError(ajaxCommon
          .handleError(vm.lastStatus));
      }
    })
    .catch(error => displayError(
       ajaxCommon.handleAjaxError(error)));
}
```

Listing 3: Based on the HTTP status code you need to handle the error response differently.

```
function handleError(lastStatus) {
  let msg = "";

  switch (lastStatus.status) {
    case 400:
      msg = JSON.stringify(lastStatus.response);
      break;
    case 404:
      if (lastStatus.response) {
        msg = lastStatus.response;
      }
      else {
        msg = `${lastStatus.statusText}
               - ${lastStatus.url}`;
      }
      break;
    case 500:
      msg = JSON.parse(
        lastStatus.response).message;
      break;
    default:
      msg = JSON.stringify(lastStatus);
      break;
  }

  if (msg) {
    console.error(msg);
  }

  return msg;
}
```

Add the handleError() method, as shown in **Listing 3**, just below the handleAjaxError() method you just created. This method checks the HTTP status code to determine how to handle the data returned from the Web API server. Remember, depending on the HTTP status code, you may retrieve just a simple piece of text, or a JSON object. Looking back at **Listing 2**, you can see in the second .then() method that the data passed into that .then() method is stored into the *lastStatus.response* property. It's this data that could either be text or a JSON object based on the status code. To expose the two methods publicly from the *ajaxCommon* closure, modify the *return* literal object to look like the following:

```
return {
  "handleAjaxError": handleAjaxError,
  "handleError": handleError
};
```

Try It Out

Make sure you set the *apiUrl* property in the *appSettings* object back to the valid endpoint "http://localhost:5000/api/". Save your changes and run the project. If you typed everything in correctly, you should still see the array of product objects displayed in the message label. Now, try each of the error conditions outlined in the previous section of this article to ensure that you're getting the same errors reported as before.

Display All Products in an HTML Table

Instead of displaying all the products in the message label, let's put the array of product data into an HTML table, as shown in **Figure 2**. There are many different methods you can use to create an HTML table. I'm going to use a templating engine called mustache.js, which you can find at https://github.com/janl/mustache.js. A templating engine allows you to create some HTML in a **<script id="dataTmpl" type="text/html">** tag, add some replaceable tokens in the format of {{property_name}}, then combine the HTML from this tag with data in an array of a literal object. The mustache templating engine iterates over the array of data and replaces the tokens with the data from each element of the array and places the resulting HTML into the DOM at the location you specify.

To make this work, add two new properties to the *vm* object literal, as shown in the following code snippet:

```
let vm = {
  "list": [],
  "mode": "list",
  // REST OF THE PROPERTIES HERE
}
```

Change the get() function by adding code to set the *vm.mode* property to "list" immediately after the function declaration. Within the second .then() method, remove the line of code displayMessage(JSON.stringify(data)); that's located in the **if (vm.lastStatus.ok)** block. Replace the lines of code within the if statement, as shown in the following code snippet:

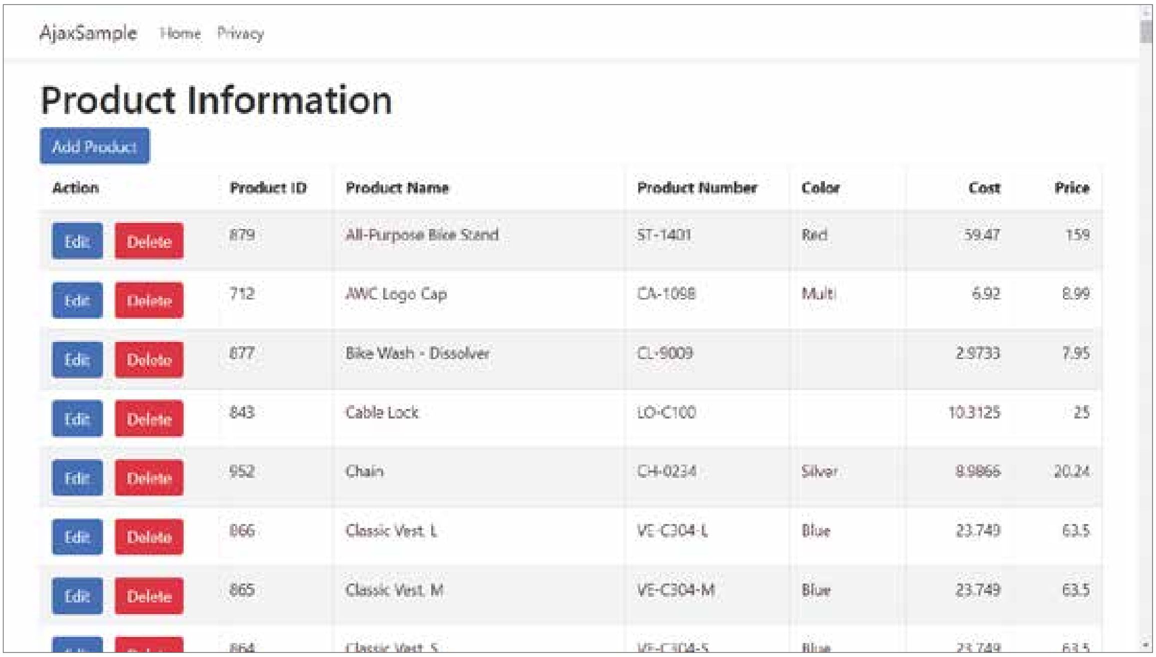
```
function get() {
  vm.mode = "list";

  // REST OF THE CODE
    if (vm.lastStatus.ok) {
      // Assign data to view model's
      // list property
      vm.list = data;
      // Use template to build HTML table
      buildList(vm);
    }
  // REST OF THE CODE
}
```

Listing 4: Create the HTML table, but leave the <tbody> blank for the templating engine to fill in.

```
<table id="products"
       class="table table-bordered
              table-striped table-collapsed">
  <thead>
    <tr>
      <th>Action</th>
      <th>Product ID</th>
      <th>Product Name</th>
      <th>Product Number</th>
      <th>Color</th>
      <th class="text-right">Cost</th>
      <th class="text-right">Price</th>
    </tr>
  </thead>
  <tbody></tbody>
</table>
```

Open the **index** page and locate the <table> shown in **Listing 4**. Notice that the <thead> element is filled in with the appropriate headers needed to describe the product data. However, the <tbody> element is blank. It's into the blank <tbody> element where you create the appropriate <tr> and <td> elements to match the <th> elements in the header with the individual property values from each row in the array of product data.

If you scroll down more in the **index** page, you'll find a <script> tag with a type of "text/html" (**Listing 5**). Inside this <script> tag, you can see a combination of HTML and



Product Information

Action	Product ID	Product Name	Product Number	Color	Cost	Price
Edit Delete	879	All-Purpose Bike Stand	ST-1401	Red	59.47	159
Edit Delete	712	AWC Logo Cap	CA-1098	Multi	6.92	8.99
Edit Delete	877	Bike Wash - Dissolver	CL-9009		2.9733	7.95
Edit Delete	843	Cable Lock	LO-C100		10.3125	25
Edit Delete	952	Chain	CH-0234	Silver	8.9866	20.24
Edit Delete	866	Classic Vest, L	VE-C304-L	Blue	23.749	63.5
Edit Delete	865	Classic Vest, M	VE-C304-M	Blue	23.749	63.5
	864	Classic Vest, S	VE-C304-S	Blue	23.749	63.5

Figure 2: Create an HTML table using a templating engine such as mustache.js.

Listing 5: Create a template to generate an HTML table within a <script> tag.

```
<script id="dataTmpl" type="text/html">
  {{#list}}
  <tr>
    <td>
      <button type="button"
              class="btn btn-primary"
         onclick="productController
                 .getEntity({{productID}});">
        Edit
      </button>
      &nbsp;
      <button type="button"
              class="btn btn-danger"
         onclick="productController
                 .deleteEntity({{productID}});">
        Delete
      </button>
    </td>
    <td>{{productID}}</td>
    <td>{{name}}</td>
    <td>{{productNumber}}</td>
    <td>{{color}}</td>
    <td class="text-right">
      {{standardCost}}
    </td>
    <td class="text-right">
      {{listPrice}}
    </td>
  </tr>
  {{/list}}
</script>
```

replaceable tokens {{property_name}} that mustache uses to generate each row of the table. The token {{#list}} refers to the *list* property you just added to the *vm* literal object. The pound sign (#) informs mustache that this variable is the array to iterate over. Think of the two tokens {{#list}} and {{/list}} as the beginning and the ending of the loop respectively. As mustache loops through each item, it starts creating each row of the table. When it finds a {{property_name}} token, it looks into the current array item and extracts the property_name, such as *productID* or *productNumber* from the current product object and replaces the value of those properties into the location of the {{property_name}} token. Mustache continues building each row of HTML as it loops through each item in the product array. What's nice about placing the HTML into a <script> tag like this is that it's much more readable than if you used a normal JavaScript loop and had to build the HTML using normal strings.

So how do you use the mustache templating engine to use the code in the <script> tag and combine that with the data you put into the *vm.list* property? In the code you just added to the get() function, you set the *vm.list* property with the array of products, and you then call a method named buildList(). Add this buildList() method in the productController using the code presented below.

```
function buildList(vm) {
  // Get HTML template from <script> tag
  let template = $("#dataTmpl").html();

  // Call Mustache passing in the template and
  // the object with the collection of data
  let html = Mustache.render(template, vm);

  // Insert the rendered HTML into the DOM
  $("#products tbody").html(html);

  // Display HTML table and hide <form> area
  displayList();
}
```

The buildList() function first reads the HTML from the script tag using the jQuery html() method and puts that HTML into the variable named *template*. Next, the Mustache.render() method is called passing in the *template* variable and the *vm* object that contains the *list* property. The render() method passes back the HTML it generated into a variable named *html*. Use the jQuery html() method to set the HTML generated by mustache into the <tbody> element in the products table.

In the **index** page, there's a <div id="list"> wrapped around the products table. The <form> tag on the page has an ID of detail. Both elements are hidden by default because they are set with the attribute of class="d-none". Either the <table> or the <form> is displayed at any one time, so you need a function named displayList() to remove the **"d-none"** class from the list and add the **"d-none"** class to the detail.

```
function displayList() {
  $("#list").removeClass("d-none");
  $("#detail").addClass("d-none");
}
```

Try It Out

Save all the changes you made and run the project. All the products should now appear in the HTML table in your browser, as shown in **Figure 2**.

Get a Single Product

The only difference in the Fetch API code between fetching all rows from the product table and a single row is to include a forward-slash and the product ID to retrieve on the URL, for example, http://localhost:5000/api/product/710. Open the **product.js** file and add a getEntity() function to look like **Listing 6**.

There are a few more methods you need to add to the *productController* to support displaying the product detail. Add a setInput() method that takes a product object and places each property's value into the appropriate <input> tag within the <form> element.

```
function setInput(entity) {
  $("#productID").val(entity.productID);
  $("#name").val(entity.name);
  $("#productNumber").val(entity.productNumber);
  $("#color").val(entity.color);
  $("#standardCost").val(entity.standardCost);
  $("#listPrice").val(entity.listPrice);
  $("#sellStartDate").val(entity.sellStartDate);
}
```

The Save and Cancel buttons aren't hidden currently, but you're going to be making them disappear later in this article. When you get a product and are displaying the detail area with the <form> element, call a displayButtons() method to ensure that those two buttons are visible using the following code.

Listing 6: Retrieve a single product using Fetch.

```
function getEntity(id) {
  vm.mode = "edit";

  // Retrieve a single entity
  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint + "/" + id)
    .then(response =>
      processResponse(response))
    .then(data => {
      if (vm.lastStatus.ok) {
        // Fill lastStatus.response
        // with the data returned
        vm.lastStatus.response = data;

        // Display entity
        setInput(data);

        // Unhide Save/Cancel buttons
        displayButtons();

        // Unhide detail area
        displayDetail();
      }
      else {
        displayError(ajaxCommon
          .handleError(vm.lastStatus));
      }
    })
    .catch(error => displayError(
       ajaxCommon.handleAjaxError(error)));
}
```

```
function displayButtons() {
  $("#saveButton").removeClass("d-none");
  $("#cancelButton").removeClass("d-none");
}
```

As mentioned previously, only the table or the detail area can be displayed at a time. Add a method named displayDetail() to hide the HTML table and display the detail area within the <form> element.

```
function displayDetail() {
  $("#list").addClass("d-none");
  $("#detail").removeClass("d-none");
}
```

If the user clicks on the wrong Edit button next to a product, they may wish to go back to the HTML table. This functionality is what the Cancel button is for. Add a new method called cancel() into the *productController*. This method first hides the <form> detail area by adding the **"d-none"** class back to the form. It then clears any messages within the message label. Finally, it calls the get() method which refreshes the data from the Web API and displays the HTML table. You don't necessarily need to call the get() method if you don't want to, you could simply call the displayList() method and have it redisplay the table of product data.

```
function cancel() {
  // Hide detail area
  $("#detail").addClass("d-none");
  // Clear any messages
  displayMessage("");
  // Display all data
  get();
}
```

The last thing to do to display the single product in the form element is to expose two of these methods publicly from the *productController* closure by modifying the *return* object. Both of these functions are called from buttons on the **index** page and thus need to be exposed publicly.

```
return {
  "setOptions": function (options) {
    if (options) {
      Object.assign(vm.options, options);
    }
  },
  "get": get,
  "getEntity": getEntity,
  "cancel": cancel
};
```

Try It Out

Save all your changes and run the project. Click on one of the Edit buttons next to a product to see the detail page appear with the product information filled into each input field. Click on the Cancel button to return to the HTML table of products.

Display a Blank Product for Adding

When you want the user to add a new product, you need to present a blank detail page to them. Use the same <form> and <input> elements you use for editing, just create an empty product object to load into those <input> elements. Create a new method named clearInput() in the *productController* as shown in the code below. Once the new product object is created with any default values, pass this object to the setInput() method.

```
function clearInput() {
  let entity = {
    "productID": 0,
    "name": "",
    "productNumber": "",
    "color": "",
    "standardCost": 0,
    "listPrice": 0,
    "sellStartDate": new Date()
      .toLocaleDateString()
  };

  setInput(entity);
}
```

On the **index** page, there's an Add button that, when clicked calls, a method named productController.add(). Create the add() method in the product controller, as shown in the code below. Notice that the mode property is set to "add" whereas in the getEntity() method you set the mode property to "edit". This will be used in the save() method.

```
function add() {
  vm.mode = "add";

  // Display empty entity
  clearInput();
  // Display buttons
  displayButtons();

  // Unhide detail area
  displayDetail();
}
```

Because this method needs to be called from outside the *productController*, modify the *return* object to include this new add() method.

```
return {
  // REST OF THE CODE HERE
  "getEntity": getEntity,
  "cancel": cancel,
  "add": add
};
```

Try It Out

Save all your changes and run the project. Click on the Add Product button just above the HTML table and you should see a blank set of input fields appear.

Create a Save Method

After the user adds or edits a product, they need to click on the Save button to send that information to the Web API for storage into the Product table. The Save button calls a method named productController.save(). Add this save() method to the *productController* as shown below. The code checks the *vm.mode* property to see if it's "add" or "edit". If the *mode* is set to "add", a method named insertEntity() is called. If the *mode* is set to "edit", a method named updateEntity() is called.

```
function save() {
  // Determine method to call
  // based on the mode property
  if (vm.mode === "add") {
    insertEntity();
  } else if (vm.mode === "edit") {
    updateEntity();
  }
}
```

You're not going to write the code to send the data to the Web API when they insert a new product yet. Instead, write some code to illustrate the sequence of events that are going to happen after the insert or update is successful. In the insertEntity() method shown below, you're going to hide the Save and Cancel buttons after successfully inserting a record. You then display a message to the user that the data was inserted. You want this message to be displayed for a couple of seconds, so you use the setTimeout() function to wait the amount of milliseconds you created in the *appSettings* object and passed into the *productController* using the setOptions() method. Once the message has been displayed for that amount of time, call get() to retrieve all of the data again and redisplay the HTML table of products. Finally, clear the message that was displayed.

```
function insertEntity() {
  // Hide Save/Cancel buttons
  hideButtons();

  displayMessage("Data Inserted.");

  setTimeout(() => {
    // Redisplay all data
    get();

    // Clear message
    displayMessage("");
  }, vm.options.msgTimeout);
}
```

The updateEntity() method follows the same design pattern as the insertEntity() method. If the update is successful, hide the Save and Cancel buttons and display a success message. This message is displayed for a couple of seconds and then the HTML table is redisplayed, and the message is cleared.

```
function insertEntity() {
  // Hide Save/Cancel buttons
  hideButtons();

  displayMessage("Data Inserted.");

  setTimeout(() => {
    // Redisplay all data
    get();

    // Clear message
    displayMessage("");
  }, vm.options.msgTimeout);
}
```

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 "How to Use the Fetch API (Correctly)" from the Item drop-down.

ADVERTISERS INDEX

Advertisers Index

CODE Consulting www.codemag.com/consulting	7
CODE Legacy www.codemag.com/legacy	15
CODE Magazine www.codemag.com/subscribe	75
CODE Mobile www.codemag.com/mobile	43
Commercial UAV Expo – Americas www.Expouav.com	76
DevIntersection www.devintersection.com	2
dtSearch www.dtSearch.com	35
LEAD Technologies www.leadtools.com	5

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

Add a method named hideButtons() to make the Save and Cancel buttons invisible. Using jQuery, select each button and add the bootstrap class **"d-none"** to each button to make them invisible. The reason you're making these buttons invisible is that after you've successfully sent the product information to be inserted or updated, you want to display any data sent back by the Web API in the detail area for a couple of seconds. You don't want the user to be able to click on the buttons again, so by making them invisible, they're unable to click on them again.

```
function hideButtons() {
  $("#saveButton").addClass("d-none");
  $("#cancelButton").addClass("d-none");
}
```

The save() method is called from the Save button on in the index page, so you need modify the *return* literal object in the productController to expose this method.

```
return {
  // REST OF THE CODE HERE
  "cancel": cancel,
  "add": add,
  "save": save
};
```

Try It Out

Save all your changes and run the project. Click on the Add Product button just above the HTML table and you should see a blank set of input fields appear. Click the Save button and you'll see a message appear above the input fields and the Save and Cancel buttons should disappear. After about two seconds, the message goes away, and the HTML table is redisplayed. Next, try clicking on an Edit button and click the Save button. Again, you should see a message appear and the Save and Cancel buttons disappear. After about two seconds, the message goes away and the HTML table is redisplayed.

Listing 7: Insert a product object by setting the method property to 'POST'.

```
function insertEntity() {
  let entity = getFromInput();

  let options = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(entity)
  };

  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint, options)
    .then(response =>
      processResponse(response))
    .then(data => {
      if (vm.lastStatus.ok) {
        // Fill lastStatus.response
        // with the data returned
        vm.lastStatus.response = data;

        // Hide buttons while
        // 'success message' is displayed
        hideButtons();

        // Display a success message
        displayMessage(
          "Product inserted successfully");

        // Redisplay entity returned
        setInput(data);

        setTimeout(() => {
          // After a few seconds,
          // redisplay all data
          get();

          // Clear message
          displayMessage("");
        }, vm.options.msgTimeout);
      }
      else {
        displayError(ajaxCommon
          .handleError(vm.lastStatus));
      }
    })
    .catch(error => displayError(
      ajaxCommon.handleAjaxError(error)));
}
```

Listing 8: Add an updateProduct() function to be able to modify a product using the Fetch API.

```
function updateEntity() {
  let entity = getFromInput();

  let options = {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(entity)
  };

  fetch(vm.options.apiUrl +
        vm.options.urlEndpoint + "/" +
        entity.productID, options)
    .then(response =>
      processResponse(response))
    .then(data => {
      if (vm.lastStatus.ok) {
        // Fill lastStatus.response
        // with the data returned
        vm.lastStatus.response = data;

        // Hide buttons while
        // 'success message' is displayed
        hideButtons();

        // Display a success message
        displayMessage(
          "Product updated successfully");

        // Redisplay entity returned
        setInput(data);

        setTimeout(() => {
          // After a few seconds,
          // redisplay all data
          get();

          // Clear message
          displayMessage("");
        }, vm.options.msgTimeout);
      }
      else {
        displayError(ajaxCommon
          .handleError(vm.lastStatus));
      }
    })
    .catch(error => displayError(
      ajaxCommon.handleAjaxError(error)));
}
```

Listing 9: Add a deleteProduct() function to be able to delete a product using the Fetch API.

```
function deleteEntity(id) {
  if (confirm(`Delete Product ${id}?`)) {
    let options = {
      method: 'DELETE'
    };

    fetch(vm.options.apiUrl +
          vm.options.urlEndpoint +
          "/" + id, options)
      .then(response =>
        processResponse(response))
      .then(data => {
        if (vm.lastStatus.ok) {
          // Fill lastStatus.response
          // with the data returned
          vm.lastStatus.response = data;

          // Display success message
          displayMessage(
            "Product deleted successfully");

          // Redisplay all data
          get();

          setTimeout(() => {
            // Clear message
            displayMessage("");
          }, vm.options.msgTimeout);
        }
        else {
          displayError(ajaxCommon
            .handleError(vm.lastStatus));
        }
      })
      .catch(error => displayError(
        ajaxCommon.handleAjaxError(error)));
  }
}
```

Inserting a Product

All you've done so far is to pass a single parameter, the URL, to the fetch() function. There's a second parameter you can pass to the fetch() function, which is a literal JSON object called the *options* object. You're going to need to use this *options* object when inserting, updating, or deleting data. The *options* object has many properties and a few are illustrated in the following code snippet. For a complete list of the *options* object properties visit https://mzl.la/3v5g32n.

```
fetch(URL, {
  // *GET,POST,PUT,DELETE
  method: 'GET',
  // cors, no-cors, *cors, same-origin
  mode: 'cors',
  // *default, no-cache, reload,
  // force-cache, only-if-cached
  cache: 'no-cache',
  // include, *same-origin, omit
  headers: {
    'Content-Type': 'application/json'
  }
})
```

Modify the insertEntity() method you created in the last section of this article to look like **Listing 7**. This method gathers the product data from the user input fields on the index page using a method named getFromInput() and puts them into a variable named *entity*. An *options* object is created and sets the *method*, *headers*, and *body* properties with the appropriate data. The *method* property is set to POST to tell the Web API server which method to invoke. The Content-Type *header* is set to application/json to inform the server to expect JSON data. The *body* property is set to the stringified version of the *entity* literal object.

The fetch() function is invoked using the full URL and the *options* object. The processResponse() method converts the *body* property in the *response* object and passes it to the second .then() method. If the *lastStatus.ok* property is set to true, use the code you learned about in the last section to hide the Save and Cancel buttons, display a success method, and display the product data sent back from the server. The reason to redisplay the product data sent back from the server is that sometimes you might have a field that's generated by SQL Server and you might want to display that new value to the user. After a specified number of seconds, the HTML table is redisplayed, and the informational message is cleared.

In order to submit the product data, the user fills in to the input fields, so you need a method named getInput(), as shown in the code below. This method uses jQuery to gather the values from each input field and build a literal product object.

```
function getFromInput() {
  return {
    "productID": $("#productID").val(),
    "name": $("#name").val(),
    "productNumber": $("#productNumber").val(),
    "color": $("#color").val(),
    "standardCost": $("#standardCost").val(),
    "listPrice": $("#listPrice").val(),
    "sellStartDate":
      new Date($("#sellStartDate").val())
  };
}
```

Try It Out

Save all your changes and run the project. Click on the Add Product button and enter the data shown in **Table 1** into the input fields.

Click on the Save button and, if you've done everything correctly, you should see the message **Product inserted successfully** appear in the message label. After a couple of seconds, the HTML table will reappear, and you should see your new product appear as the first row in the table.

Field	Value
Product ID	0
Product Name	A New Product
Product Number	NEW-000
Color	Red
Cost	20
Price	40
Sell Start Date	<Today's Date>

Table 1: Enter some valid values to insert into the Product table

Updating a Product

When you click on the Edit button on one of the rows in the table, the product data is displayed in the detail area. You then modify any of the fields and click the Save button to update the data into the Product table. Modify the updateEntity() method you wrote earlier to make the Web API to accomplish this. Locate the updateEntity() method in the productController and change it to look like the code shown in **Listing 8**.

In the updateEntity() method, you retrieve the product data input by calling the getFromInput() method. Create an *options* variable and set the *method* property to PUT to inform the Web API to call the method to update the data. The fetch() function is called using the URL endpoint and passing the product ID to update on the URL. In addition, the *options* object is passed as the second parameter to the fetch() function. The rest of the code is similar to what you just wrote for the insertEntity(). If the *lastStatus.ok* property is set to a true value, hide the Save and Cancel buttons and display a message to inform the user that the data was successfully updated. Display the product data sent back from the server in the input fields just in case any of the values have been updated by the server during the update process. Finally, after a couple of seconds, the HTML table is redisplayed, and the informational message is cleared.

Try It Out

Save all the changes you made and run the project. Click on one of the products and modify a couple of fields like the Cost and Price. Click the Save button and ensure that everything works correctly.

Deleting a Product

The final functionality to add to the **index** page is the ability for the user to delete a product. If the user clicks on the Delete button in one of the rows in the product table, you should prompt the user whether they really wish to delete that product. If they answer that they wish to perform the delete, call a deleteEntity() method in the *productController*. Add the code for the deleteEntity() method as shown in **Listing 9** to the *productController* closure.

The first line of code in the deleteEntity() method calls the confirm() function to display a confirmation dialog to the user to which they must respond with either OK or Cancel. If they answer OK, the code within the if() block is executed. Create an *options* object with the *method* property set to DELETE. Pass the product ID passed into this method on the URL line as the first parameter to the fetch() function and the *options* object as the second parameter. In the second .then() method, if the *vm.lastStatus.ok* is set to true, a success message is displayed in the message label. The get() method is called to reload the HTML table. After a couple of seconds, the message in the label is cleared. Because the deleteEntity() method needs to be accessed from the index page, you need to modify the *return* literal object and add this method to the list of methods to be made public, as shown in the code below:

```
return {
  // REST OF THE CODE HERE
  "add": add,
  "save": save,
  "deleteEntity": deleteEntity
};
```

Try It Out

Save all the changes you made and run the project. Not all of the products in the Product table can be deleted because of relationships set up in the database. You should add a new product, then delete that new product to test the delete functionality.

Summary

The Fetch API is different from the XMLHttpRequest object and the jQuery $.ajax() method call. The fetch() function is a straight-forward API to use, but as you learned, the exception handling can be a little challenging. Hopefully, you now have a good design pattern to use if you wish to use this API. If you're already using the jQuery $.ajax() method, I recommend that you keep using it and not switch to the Fetch API. You definitely have more options using the $.ajax() method. For more information on a comparison between XMLHttpRequest and the Fetch API, check out the post at https://bit.ly/3xpir6n.

Paul D. Sheriff

ONLINE QUICK ID 2107041

Eliminate Secrets from Your Applications with Azure Managed Identity

In the May/June 2021 issue of CODE Magazine, I wrote an article called "Can You Keep a Secret? Azure Can!" showing you how to store a connection string with its secrets in Azure Key Vault and then use Azure Managed Identities with .NET Core to let your application access that while debugging locally in Visual Studio or Visual Studio Code. The best part is that you don't have to be a security or SysOps guru to do this. The clear evidence is that I was able to pull it off!

That's all well and good for debugging applications in your IDE, but when it's time to deploy your app, you can take this secret sharing even further. If you're using Azure SQL for your database and your application is running in Azure, it's possible (and easy) to have an Azure Managed Identity to authenticate and access your database. The services are all part of the same ecosystem and they know how to share among themselves.

In this article, I'll begin with the application as I left off in the earlier article, and walk you through deploying it and removing all secrets from the connection string. In the end, it's simply a matter of a leveraging a new feature of the SQLClient API to use Managed Identity for authentication.

I think this option is much easier than how we had to leverage Managed Identity authentication with EF Core prior to this new feature, which was by using EF Core Interceptors. Interceptors are a great feature, but in this particular case, the new workflow is much simpler.

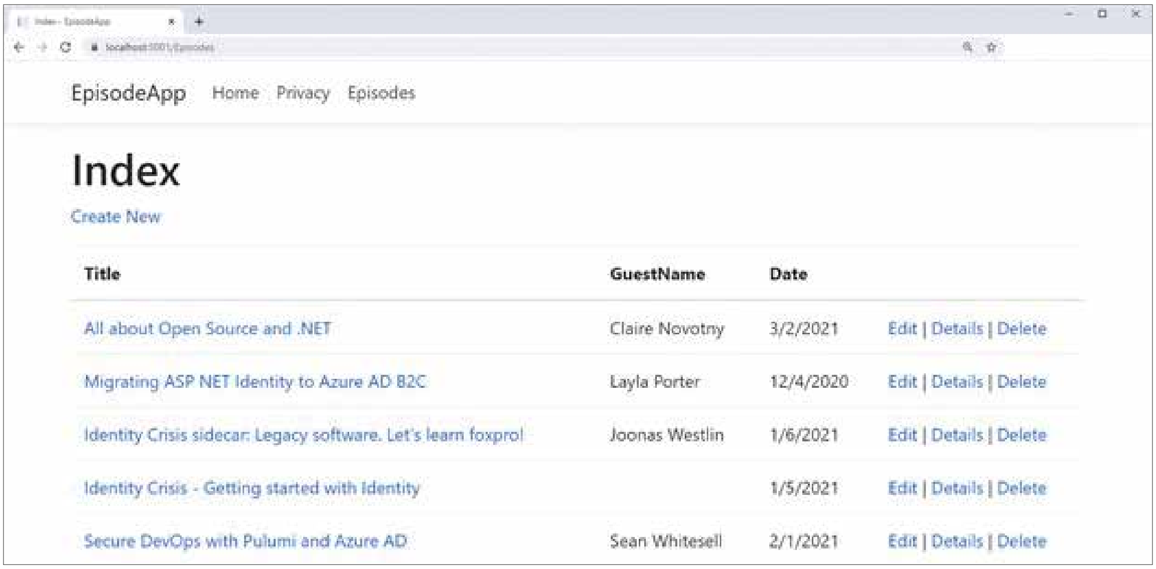
As a non-security person, going directly to the new workflow continued to confuse me. However, walking pragmatically through the steps truly helped me understand what I was doing, why I was doing it, and eased me into having a much deeper comprehension of Azure Managed Identity than I originally thought I wanted to be bothered with. I can no longer claim ignorance, which was, honestly, more a matter of fear of failure than anything else. I want to help my fellow "OMG please don't make me learn security stuff" developers adopt this new area of expertise as well.

Even as we remove the need for storing the secrets of the connection string, there are plenty of other secrets that you may want to continue storing in the Key Vault. Here, I'll consider the rest of the details of the connection string important enough to keep it in the Key Vault and not embed it in the application code.

A Quick Overview of Where We're Starting

The application I built in the previous article was to maintain a list of episodes of "The 425 Show," a Twitch stream run by the Identity Developer Advocacy team.

The ASP.NET Core Razor Pages application (**Figure 1**) uses EF Core to interact with its data.



Index

Create New

Title	GuestName	Date	
All about Open Source and .NET	Claire Novotny	3/2/2021	Edit | Details | Delete
Migrating ASP NET Identity to Azure AD B2C	Layla Porter	12/4/2020	Edit | Details | Delete
Identity Crisis sidecar: Legacy software. Let's learn foxpro!	Joonas Westlin	1/6/2021	Edit | Details | Delete
Identity Crisis - Getting started with Identity		1/5/2021	Edit | Details | Delete
Secure DevOps with Pulumi and Azure AD	Sean Whitesell	2/1/2021	Edit | Details | Delete

Figure 1: The sample app as displayed when debugging locally

Julie Lerman
@julielerman
thedatafarm.com/contact

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

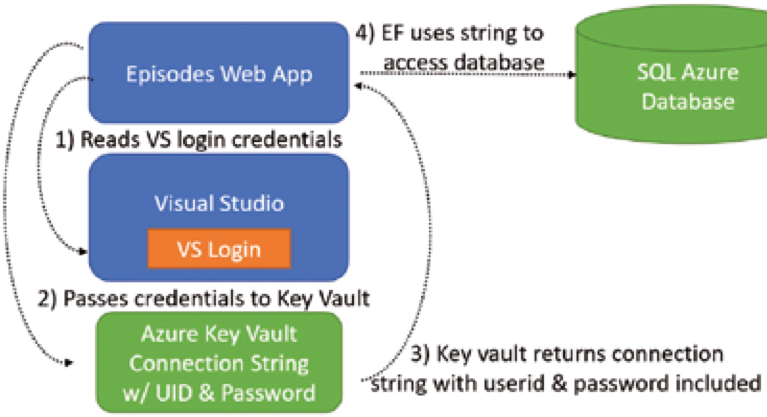

Episodes Web App
4) EF uses string to access database
SQL Azure Database
1) Reads VS login credentials
Visual Studio
VS Login
2) Passes credentials to Key Vault
Azure Key Vault Connection String w/ UID & Password
3) Key vault returns connection string with userid & password included

Figure 2: Accessing the connection string stored in Key Vault while debugging in Visual Studio

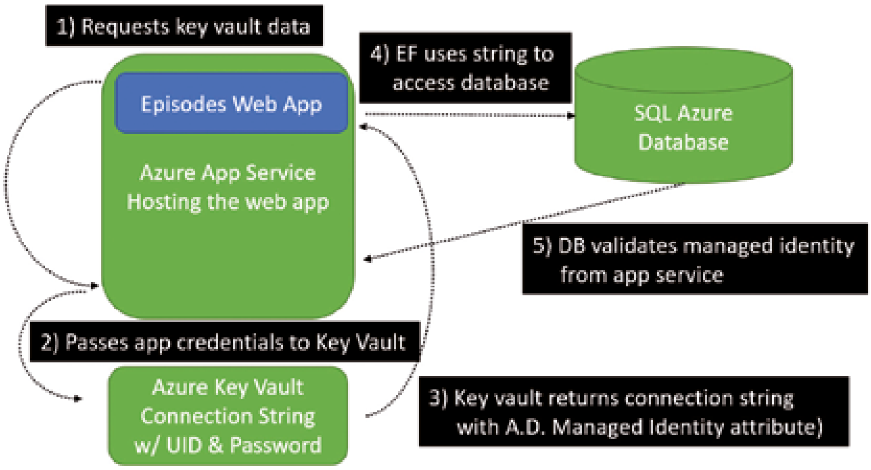

1) Requests key vault data
Episodes Web App
Azure App Service Hosting the web app
4) EF uses string to access database
SQL Azure Database
5) DB validates managed identity from app service
2) Passes app credentials to Key Vault
Azure Key Vault Connection String w/ UID & Password
3) Key vault returns connection string with A.D. Managed Identity attribute)

Figure 3: Accessing and using a credential-less connection string stored in Key Vault from an app in Azure App Service

I started out with a SQL Server LocalDB on my computer and its connection string tucked into appsettings.json. Then I moved the connection string into Azure Key Vault and using the Azure.Identity SDK for accessing secrets (a combination of Azure.Identity and Azure.Extensions.AspNetCore.Configuration.Secrets NuGet packages) I told my application to look for the connection string in Azure Key Vault. One of the critical characteristics was that these APIs are able to read the credentials with which I had signed into Visual Studio. Those are the same credentials tied to the account I use to sign into my Azure Subscription. I also told my Key Vault that those same credentials could be used to read the secrets in Key Vault. While debugging, the API was able to quietly read and then pass those credentials to Azure, access the key Vault, read the stored connection string, and then pass it back to the application, which then used the connection string to read my local database.

It may seem pretty silly to store this in Azure—the connection string to my local database, and a LocalDb at that doesn't even require a user ID or password—so that Visual Studio could locally debug my application. But that was just Step 1. I then changed that connection string to point to an Azure SQL Server and added in the user ID and password required to access the SQL Server.

And at the end of that article, I could debug my app in Visual studio and have it seamlessly read the connection string from Azure Key Vault (thanks to the Azure.Identity SDK) and then use that connection string to access the database. **Figure 2** shows the interaction between the app being debugged in Visual Studio and the Azure resources.

I still have secrets stored in the Key Vault. The connection to the database is not so precious, but the user ID and password certainly are. As I publish the app, I can leverage Managed Identity to remove even those secrets from the connection string and double-down on the security of my database.

How to Wire Up the App and the Database

To be clear, Azure Key Vault won't be responsible for allowing the deployed app to access the database. I'll still use a Managed Identity to read the connection string from the key vault (Steps 1-3 in **Figure 3**) and then I'll also be using managed identity to provide permissions for the app to talk to the database (Steps 4 and 5 in **Figure 3**).

First, you'll need to publish the Web app to an Azure Application Service. You can right-click on a project in Visual Studio, choose Publish, and walk-through publishing to Azure. I'm choosing a Linux app service because my application is .NET Core. If you want a more detailed walkthrough on publishing ASP.NET Core apps to Azure, check out the Microsoft doc's QuickStart document at https://docs.microsoft.com/en-us/azure/app-service/quickstart-dotnetcore. Note that if you're using a free subscription, Azure App Service is always free and a small single Azure SQL database is free. Key Vault is not free. But for testing in a tiny scenario like this demo, it's nominal. My US East-based subscription is $0.03USD per 10,000 transactions.

If you're not familiar with publishing ASP.NET Core apps to Azure, you might find it interesting that the publish wizard discovers and notes the connection string name in the startup configuration.

My code for setting up the EpisodesContext in the startup class specifies "EpisodesContext" as the connection string name.

```
services.AddDbContext<EpisodesContext>
(options =>options.UseSqlServer
 (Configuration.GetConnectionString
   ("EpisodesContext")));
```

As I left the application in the earlier article, the value of that connection string (which points to my Azure SQL database and contains the user ID and password) is in the Azure Key Vault and nowhere to be found in the application code.

Two Critical Changes You Need to Start with for SQL Server

Before embarking on wiring up the published application to use Managed Identity for accessing the Key Vault and the database, there are two important changes you'll need to make. Of course, I didn't make them in advance and ended up scratching my head for a while until I realized that I needed to perform these tasks, so let's get them out of the way up front.

First, note that as I'm writing this in early May 2021, the Managed Identity support in the SqlClient API is very new. It was introduced in Microsoft.Data.SqlClient 2.1.0. But it's still so new that the current version of EF Core (5.0.5) doesn't yet have a dependency on it. EF Core will bring in version 2.0.1, which doesn't have the Managed Identity support. Perhaps by the time you are reading this, EF Core will depend on the relevant version.

If not, you'll need to add a package reference into your app for the new version of SqlCLient. Again, as I write this article, that happens to be 2.1.2. I've added this to my csproj file:

```
<PackageReference
 Include="Microsoft.Data.SqlClient"
 Version="2.1.2" />
```

The second critical change is a setting in the Azure SQL Server that hosts your database. Because I was only debugging the application from my local computer, I'd added a firewall rule for my own IP address to be allowed through to the database. But now I'll have an application within the Azure ecosystem accessing it. By default, all Azure SQL Servers are locked down, so you need to explicitly tell the server to allow other Azure services to be able to access the server. Then further authentication is used to access the database.

vTo enable this, I returned to the firewall settings of the SQL Server and "flipped" the switch to Yes to allow Azure services and resources to access the server, as you can see in **Figure 4**.

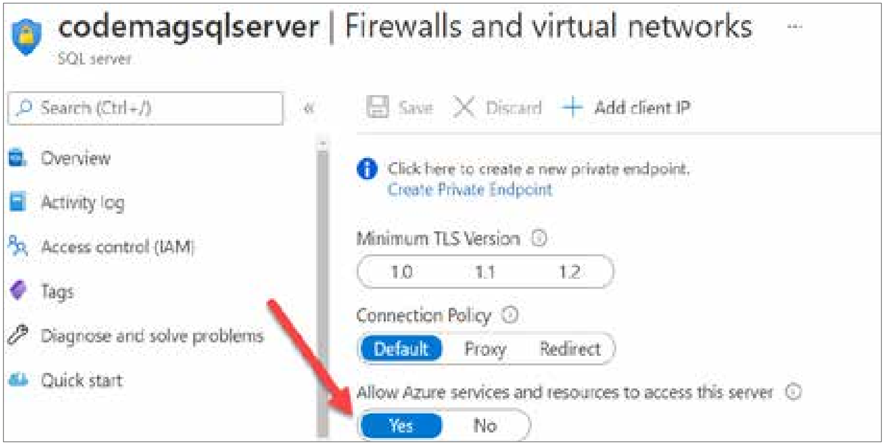
Allowing the App Service's Managed Identity to Access Other Services

The app won't work right away after it's deployed. That's because it was depending on the account I used to sign in to Visual Studio. It was this account that was configured to access Key Vault.

Instead, I need to tune the security and lock things down so that the Key Vault and database are very clearly tied to an Azure Managed Identity tied to the Episodes Azure WebApp.

I'll begin that tuning by checking in on the Identity for the App Service itself. Managed identities are accounts that are provisioned and managed by Azure AD automatically. When the App Service was created for the published application, Azure assigned it an identity. That's referred to as a "system assigned" identity, as opposed to a user assigned identity. The portal displays a handy description of a system assigned identity.

"A system assigned managed identity is restricted to one per resource and is tied to the lifecycle of this resource.



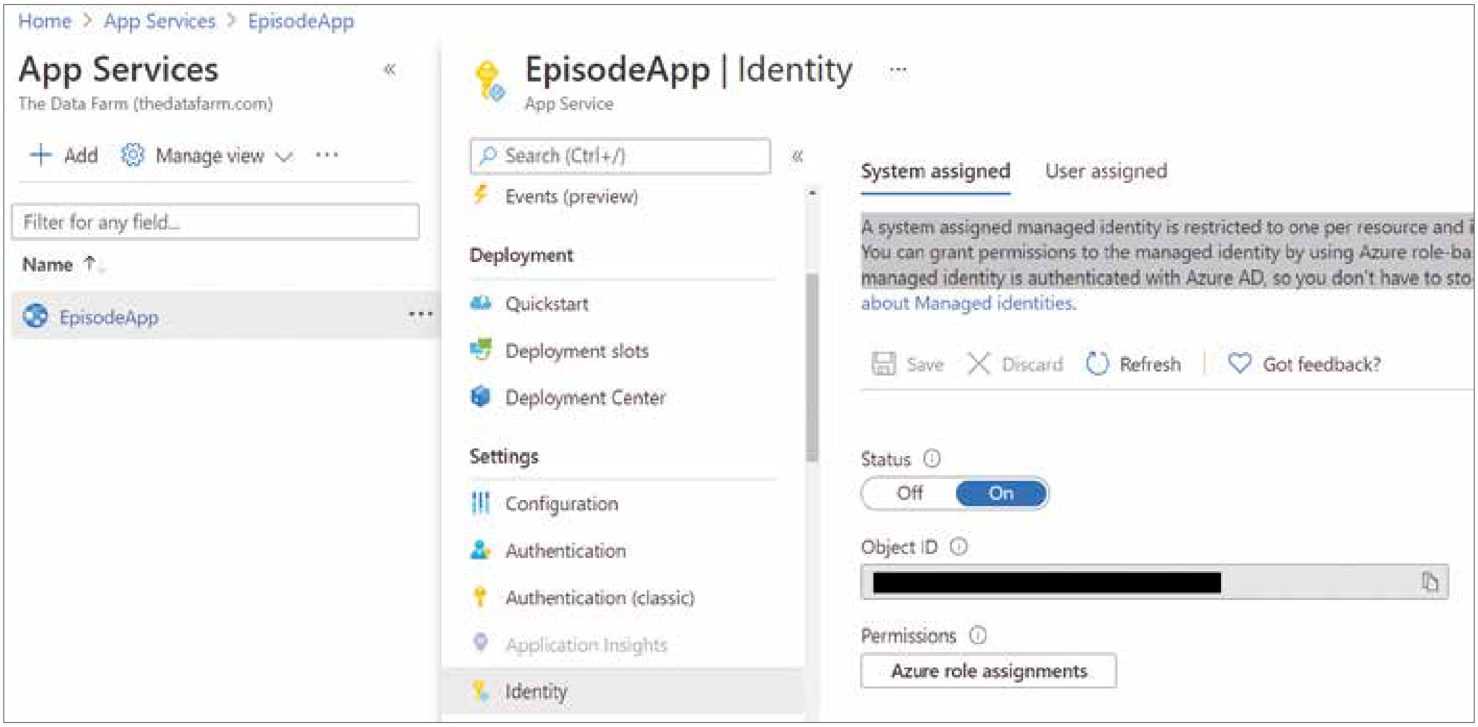
Click here to create a new private endpoint. Create Private Endpoint

Minimum TLS Version: 1.0 1.1 1.2

Connection Policy: Default Proxy Redirect

Allow Azure services and resources to access this server: Yes No

Figure 4: Allowing Azure services to access the Azure SQL Server



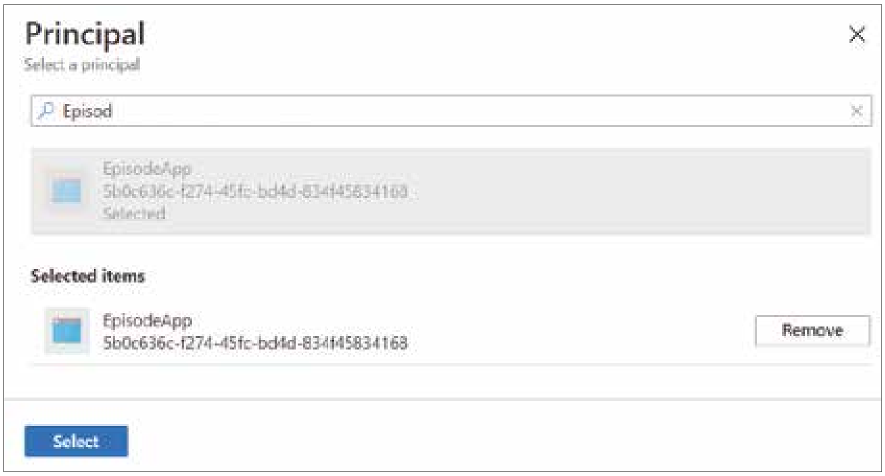
A system assigned managed identity is restricted to one per resource and i
You can grant permissions to the managed identity by using Azure role-ba
managed identity is authenticated with Azure AD, so you don't have to sto
about Managed identities.

Status: Off On

Object ID: [redacted]

Permissions: Azure role assignments

Figure 5: Inspecting the system-assigned identity of the EisodeApp



Principal

Select a principal

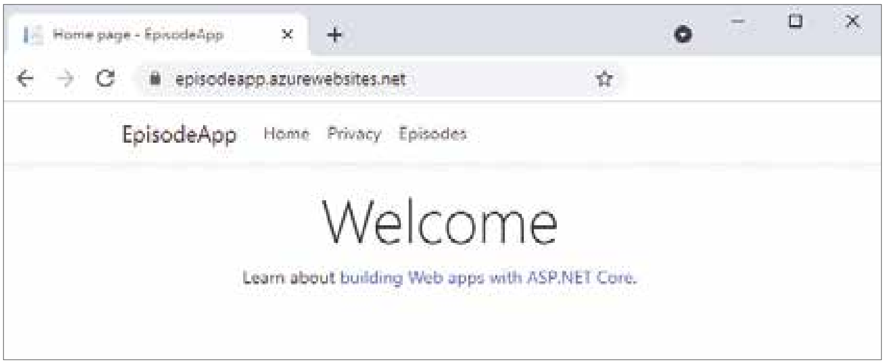
Episod

EpisodeApp
5b0c636c-f274-45fc-bd4d-834f45834168
Selected

Selected items

EpisodeApp
5b0c636c-f274-45fc-bd4d-834f45834168

Figure 6: Finding the EpisodeApp identity when creating an access policy



Home page - EpisodeApp

episodeapp.azurewebsites.net

Welcome

Learn about building Web apps with ASP.NET Core.

Figure 7: The home page of the Web app now running on Azure

You can grant permissions to the managed identity by using Azure role-based access control (Azure RBAC). The managed identity is authenticated with Azure AD, so you don't have to store any credentials in code. "

You can see the identity in the portal by opening the App Service and choosing Identity from its menu bar (**Figure 5**). You can also access identity information via the Azure CLI or PowerShell commands.

Next, you'll need to create an Access policy in Key Vault for that Managed Identity and also let the Azure SQL database know about it. If you're using the portal to set up the access, then you'll be able to search for and choose the EpisodeApp identity. That's the path I'll be following. If you're using the Azure CLI or PowerShell, then you'll need to copy that Object ID (I've covered mine in the screenshot) to include in commands.

Create an Access Policy to Read Key Vault Secrets

Hopefully, you read the earlier article and remember how to create access policies for Azure Key Vault. Here are the "Cliff Notes." Find Key Vault, choose the specific vault from the list of key vaults, and then click the Access Policies link under Settings in its menu. Finally, choose Add Access Policy. You'll only need two permissions: Get and List from the Secret permissions. Select those. The next two options are to either choose a principal or Application access. Either one will work because if you set a principal that's an app service identity, it'll recognize that it's an application and categorize it as such. I chose to set the principal, clicking on None selected, which shows a list of the top five accounts/identities in your Azure AD. You can filter down to the name of your identity; mine is EpisodeApp (**Figure 6**). Select it and then click the Select button. Then, back in the Add access policy form, click the Add button. When the portal returns to the list of access policies, you still need to save your changes. There's a save icon, harking back to the days of 3.5-inch floppy disks, at the top of the page.

Once the access policy is set up for the key vault, my app will run because it will succeed when the app's startup code attempts to hook into the key vault, which happens before the connection string is even needed. However, it wasn't instantaneous in my case. I don't know if it was a matter of time and patience—which I don't have—or the app restart I forced. **Figure 7** shows the default home page created by the template I used to build the website.

Providing the App's Managed Identity Access to the Database

If you click on the Episodes link at this point, it will still fail. Even though I now have access to Key Vault and therefore the connection string for the database, remember that there's no user ID or password in the connection string for authenticating to the database. I need to let the database know that this Azure Web App is allowed to communicate with it by using its Managed Identity.

There are two steps to achieving this:

1. Add the app service identity as a user on the database.
2. Specify read and write permissions for that user.

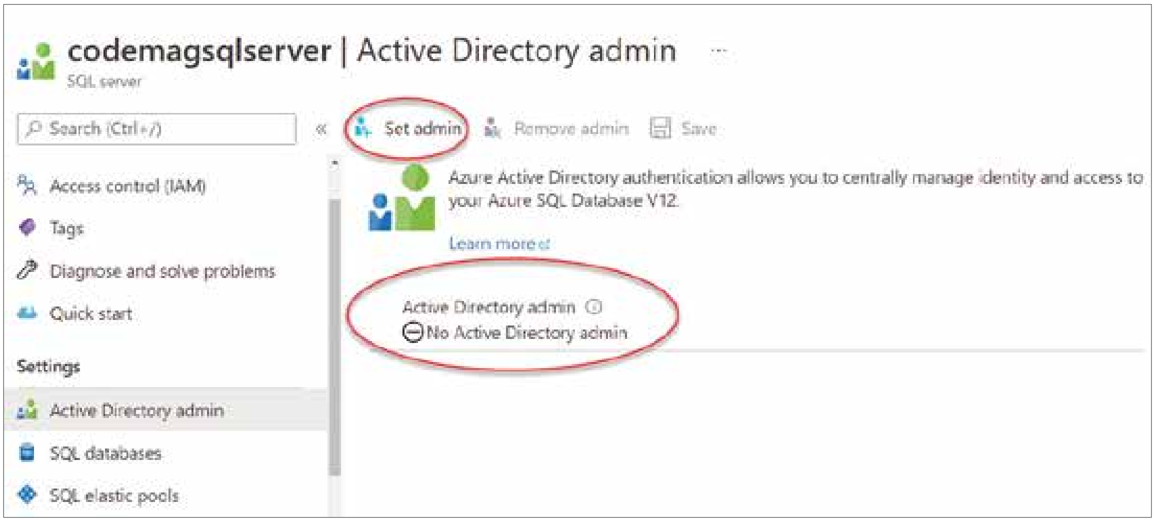
Both of these steps can be performed in TSQL through any application where you can connect to the database and execute commands. Because I already have the solution open in Visual Studio, I may as well use the SQL Server Data Tools (SSDT) in Visual Studio.

There was a wrinkle in my setup. My database had originally been created with SQL Server authentication, i.e., a user ID and password. In order to add a managed identity (the EspisodeApp identity) as a user, I have to control the database with an Active Directory account—in other words, the identity that I use to log into my Azure subscription. By default, Active Directory accounts are not given administrative privileges on Azure SQL databases. To fix this, I had to return to the database's server in the portal and under Settings, choose Active Directory admin.

There, I could see that I wasn't set up to admin the server with an Active Directory account (**Figure 8**). To remedy this, I chose **Set admin** and then selected my main Azure subscription identity (and saved that!) as an administrator of that server.

Then I was able to connect to the database from SQL Server Object Explorer using Active Directory Integrated Authentication. (**Figure 9**).

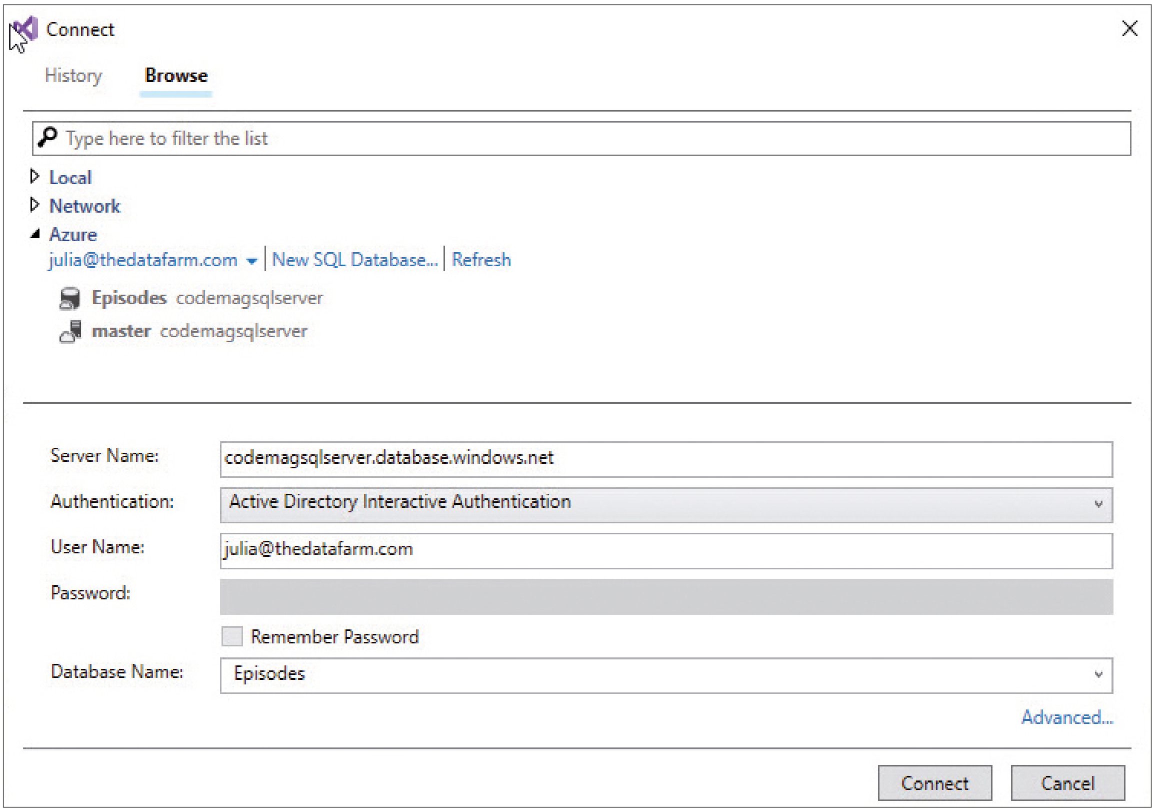
Once I have a connection to the database and a query window open, I'll execute three commands.



Azure Active Directory authentication allows you to centrally manage identity and access to your Azure SQL Database V12.

Active Directory admin: No Active Directory admin

Figure 8: Enabling the SQL Server to be administered with an Active Directory identity



Server Name: codemagsqlserver.database.windows.net

Authentication: Active Directory Interactive Authentication

User Name: julia@thedatafarm.com

Password:

☐ Remember Password

Database Name: Episodes

Figure 9: Accessing the Azure SQL database from Visual Studio using Active Directory Integrated Authentication

```
CREATE USER EpisodeApp FROM EXTERNAL PROVIDER
ALTER ROLE db_datareader ADD MEMBER EpisodeApp
ALTER ROLE db_datawriter ADD MEMBER EpisodeApp
```

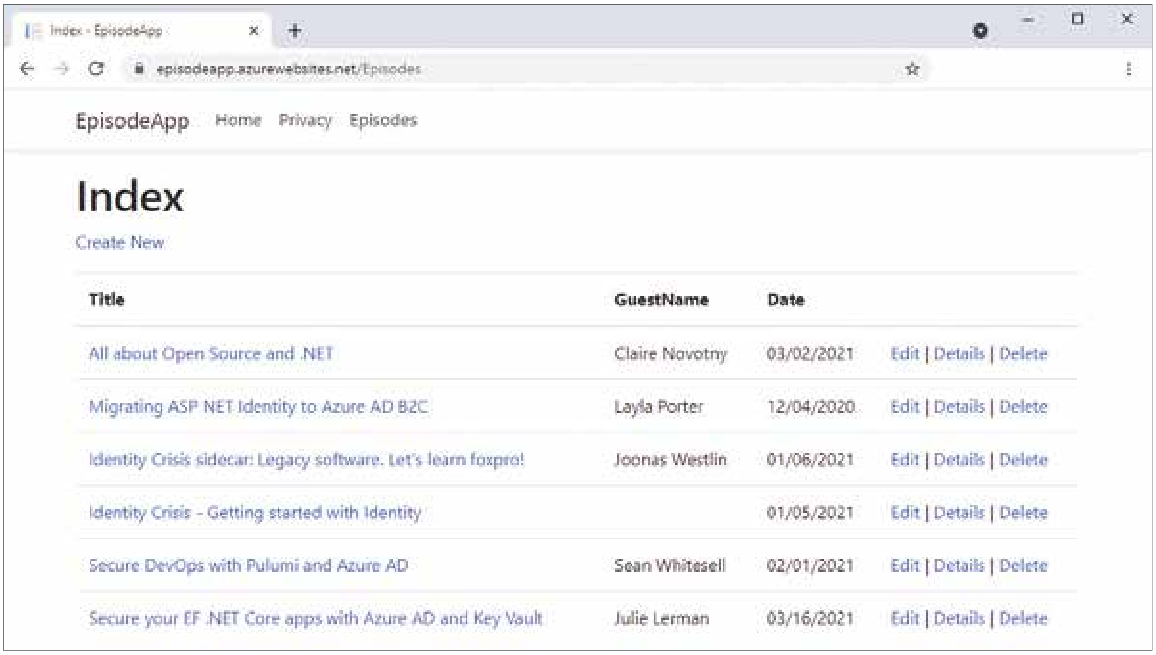
This combination associates the EpisodeApp identity as a user of the database and then allows that user to read and write data.

Signaling the Connection String to Use Managed Identity

In the last twist of this transformation, I can inform the database to use Managed Identity to authenticate the user, in this case, the Episodes Application, and grant access to the database.

This is done with an attribute in the SQL Server connection string—Authentication. Specifying Azure Active Directory with the Authentication attribute has been possible for a while, and, in fact, other APIs already supported the use of Managed Identity. It's only recently that the Microsoft.Data.SqlClient API also supported Managed Identity.

For the curious, there are now six possible values you can set in the Authentication property:



Index

Create New

Title	GuestName	Date	
All about Open Source and .NET	Claire Novotny	03/02/2021	Edit | Details | Delete
Migrating ASP NET Identity to Azure AD B2C	Layla Porter	12/04/2020	Edit | Details | Delete
Identity Crisis sidecar: Legacy software. Let's learn foxpro!	Joonas Westlin	01/06/2021	Edit | Details | Delete
Identity Crisis - Getting started with Identity		01/05/2021	Edit | Details | Delete
Secure DevOps with Pulumi and Azure AD	Sean Whitesell	02/01/2021	Edit | Details | Delete
Secure your EF .NET Core apps with Azure AD and Key Vault	Julie Lerman	03/16/2021	Edit | Details | Delete

Figure 10: The Episodes page of the app running on Azure

- Active Directory Password: You also supply UserId and Password and the database will seek them out in Azure AD.
- Active Directory Integrated: This combines using an on-premises AD with Azure AD.
- Active Directory Interactive: This mode triggers multi-factor authentication.
- Active Directory Service Principal: This involves registering the application directly with the database (non-user interactive).
- Active Directory Device Code Flow: This method is most commonly used for apps on IoT devices.
- Active Directory Managed Identity: This is what I'll use to allow Managed Identities to authenticate.

The attribute is written as

```
Authentication=Active Directory Managed Identity
```

That means my connection string will now look like this:

```
Server=
tcp:codemagsqlserver.database.windows.net,1433;
Authentication=
 Active Directory Managed Identity;
Database= Episodes
```

Keep in mind that the line wrapping is solely for the sake of this article's formatting rules.

A few points to note about this. The method used to be known as Active Directory MSI, and the API will recognize if you use "Active Directory MSI" as the value instead.

Specifying this attribute on its own in the connection string works if, like the example in this article, the identity was defined by the service. However, if you're using a user-assigned Managed Identity for authentication (created by you or another admin perhaps) then you'll need to provide the object ID of that Managed Identity in a User ID attribute.

See more details about this attribute in the docs at http://bit.ly/ADMIAuthentication.

Remember that the connection string is stored in the Azure Key Vault. The values of secrets in the key vault are immutable, so rather than editing them, the Portal gives you a way to add a new version. That path is to return to the Secrets list in your Azure Key Vault, select the secret and in the Details tab, and choose the New Version option. I typically set up the string in Notepad to make sure it's correct and then copy from there and paste into the Value text box. By default, the new (latest) version will be the only active version of the key.

With this, I can return to the Azure hosted Episodes app and browse to the Episodes list as well as edit or add episodes as I see fit (**Figure 10**).

Debug Against a Local Database

What about continuing to develop and debug from Visual Studio? Currently, the code always reads from key vault and always comes up with the new authentication mode, which will fail from Visual Studio. Even if I also put the connection string into appsettings.development.json, the Key Vault configuration will take precedence.

One approach to solving this is to conditionally read from the Key Vault only when you're not in development mode. This is controlled by the ASPNETCORE_ENVIRONMENT environment variable, which is, by default, Development on your development computer and, also by default, Production in your deployed application. If your deployed application were also in Development mode, users would see all of the detailed error and tracing information when the app fails. That's definitely not desirable.

Let's take advantage of ASPNETCORE_ENVIRONMENT to solve this problem.

I've added the localdb connection string into appsettings.development.json. Then I'll return to program.cs in the application and modify the code where, in the previous article, I instructed the app to read configurations from the Key Vault. Now I'll have it read from the key vault only if that environment variable is Production. That information is exposed through ASP.NET Core APIs as HostBuilderContext.HostingEnvironment.IsProduction().

What was formerly:

```
.ConfigureAppConfiguration((context, config) =>
{
  var builtConfig = config.Build();
  config.AddAzureKeyVault(
   new Uri(
“https://lermancodemagvault.vault.azure.net”),
   new DefaultAzureCredential());
})
```

Should now be:

```
.ConfigureAppConfiguration((context, config) =>
{
  if (context.HostingEnvironment.IsProduction())
  {
    var builtConfig = config.Build();
    config.AddAzureKeyVault(
     new Uri(
“https://lermancodemagvault.vault.azure.net”),
     new DefaultAzureCredential());
  }
})
```

I re-published the app to verify that it works both in debug mode in Visual Studio and on the Azure App Service.

Bask in the Glory of Your Totally Secure, Secret-less ASP.NET Core App Thanks to Managed Identity

That's it! I started out this process knowing nothing about Managed Identities and bringing to the table my many decades of fear of anything to do with security. Being more of a back-end person, I'd never published an ASP.NET Core app as an Azure App Service. Now I have a pretty decent understanding of Azure Active Directory, Managed Identity, and how to hook up various services in Azure to work with each other and share security information without me having to provide it in my application. I hope that I've been able to share the same confidence with you.

Although the database is secure in that it can only be used by my application, and my key vault is secure for the same reason, the application itself isn't secure because it's just a simple demo. I will definitely be locking it down prior to this article's publication, because I have seen what happens when I leave sample applications running on the Internet with anyone having the ability to enter and edit the data. You can download the code that goes with the article on the CODE Magazine website or grab it from GitHub at https://github.com/julielerman/CodeMagEpisodeApp.

Julie Lerman

ONLINE QUICK ID 2107051

Test Your REST APIs Using Insomnia REST Client

Over the past few years, APIs evolved to become the center of software development. You can take advantage of APIs to enable communication between systems and the data exchange between them. Today's applications thrive a lot on APIs—most of today's applications are API-based. You must test your APIs before releasing them for the clients or end-users to consume.

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.

It would help if you had API testing as part of your testing strategy to test your application's core business rules and help deliver better software faster. There are plenty of API testing tools around. Postman is the de facto industry-standard tool for testing and developing APIs.

Insomnia is yet another popular, fast REST client that's available for Mac, Windows, and Linux. You can use Insomnia for testing RESTful as well as GraphQL APIs. It's a free cross-platform desktop framework that incorporates a user-friendly user interface and sophisticated features, such as security helpers, code creation, and environment variables. You can take advantage of Insomnia to test HTTP-based RESTful APIs or even GraphQL APIs. This article talks about how you can fast-track API development with Insomnia REST Client.

> API is an acronym for Application Programming Interface and acts as the middle layer between the presentation layer and the database layer.

Prerequisites

If you're to work with the code examples discussed in this article, you should have the following installed in your system:

- Visual Studio 2019 (an earlier version will also work but Visual Studio 2019 is preferred)
- .NET 5.0
- ASP.NET 5.0 Runtime
- Insomnia REST Client

You can download Visual Studio 2019 from here: https://visualstudio.microsoft.com/downloads/. You can download .NET 5.0 and ASP.NET 5.0 runtime from here: https://dotnet.microsoft.com/download/dotnet/5.0. You can download and install the Insomnia REST Client from here: https://insomnia.rest.

What's API Testing?

API testing determines whether the application programming interfaces (APIs) meet functionality, consistency, efficiency, usability, performance, and security specifications. In addition, it helps uncover bugs, anomalies, or discrepancies from an API's expected behavior.

Typically, any application has three distinct layers: the presentation layer, the business layer, and the data access layer. API testing is performed at the business layer because it's the most critical of all layers in an application where the heart of the application or the business rules is stored. An API client is used to evaluate APIs for accessibility, usability, stability, reliability, and correctness.

Benefits of API Testing

Some of the benefits of API testing are:

- **Early testing**: Using API testing, you can validate your business logic even before the application is built in its entirety. API testing can also help you to find more bugs in much less time (API tests are much faster that UI tests).
- **GUI-independent:** API testing allows testing the core functionality of an application even without the need of a user interface.
- **Language-independent:** Because data is exchanged in XML or JSON format, you can use any language for test automation.
- **Improved test coverage:** Most APIs allow creating automated tests (both positive and negative tests) with high test coverage.
- **Faster releases:** API testing enables you to detect errors early in the software development life cycle, allowing for faster product releases.

Popular API Testing Tools

Some of the popular API testing tools include the following:

- Postman
- Soap UI
- Apigee
- JMeter

What Is Insomnia?

A REST client is a tool used for interacting with a RESTful API that's exposed for communication. An Insomnia REST Client is an open-source, powerful REST API client used to store, organize, and execute REST API requests elegantly. The Insomnia REST Client is an excellent alternative to Postman for sending REST and GraphQL requests with support for cookie management, environment variables, code generation, and authentication. It's available for Windows, Mac, and Linux operating systems. In addition, Insomnia incorporates a user-friendly GUI with sophisticated features such as security helpers, code creation, and environment variables.

The following are some of the features of Insomnia REST Client:

- Cross-platform support
- Ability to execute REST, SOAP, GraphQL, and GRPC requests
- Ability to store, organize, and execute REST API requests
- Ability to organize requests in workspaces and groups
- Support for query string param builder
- Ability to export and share workspaces
- Support for chained requests

Insomnia REST Client vs. Postman

Although both Postman and Insomnia have their unique features, there are certain features that are common to both, such as the following:

- Both have a free version of their software.
- Both are open-source projects.
- Both offer support for multiple workspaces.
- Both include support for GraphQL integration.
- Both have import and export of test data.
- Both use multiple ways to configure authorizations.

What's Unique about Postman

Postman is a more mature tool and the market leader in API testing tools. Some of the striking features of Postman are:

- **API documentation:** Postman is adept at generating host browser-based API documentation in real-time.
- **Monitoring:** Postman is capable of running a collection periodically to check for its performance and response.

What's Unique about Insomnia

Insomnia provides certain features that aren't supported by Postman. These features include the following:

- **Plug-ins:** Insomnia provides support for creating new plug-ins.
- **Environment variables**: Environment variables are one of the most useful features of Insomnia that can save a lot of time manually typing.
- **Code snippet generation:** Insomnia enables you to generate code snippets in 12 different languages.
- **Response format:** You can take advantage of Insomnia to view response beyond JSON and XML, i.e., you can see HTML pages, images, and even PDF documents.

Now, let's put it through its paces so you can see for yourself.

Create an ASP.NET 5 Project in Visual Studio 2019

First off, create a new ASP.NET 5 project in Visual Studio. You can create a project in Visual Studio 2019 in several ways. When you launch Visual Studio 2019, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2019 IDE.

To create a new ASP.NET 5 project in Visual Studio:

1. Start the Visual Studio 2019 Preview IDE.
2. In the "Create a new project" window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name 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. Click Create to complete the process.

This creates a new ASP.NET 5 Web application. I'll use this project throughout this article. A default controller named WeatherForecastController will be created as well. Because I won't be using this controller in this example, delete this file and update the profiles section of the launchSettings.json file with the following text:

```
"profiles": {
"IIS Express": {
  "commandName": "IISExpress",
  "launchBrowser": true,
  "launchUrl": "api/product",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Note that the launchUrl in the launchSettings.json file points to a controller named ProductController. You've yet to create this controller—you'll create it shortly.

Create a Minimalistic ASP.NET Core Web API

In this section you'll create a minimalistic RESTful API. In this example, you'll be using the following classes and interfaces:

- **Product:** This is the entity class you'll use in this application for storing Product data.
- **IProductRepository :** This interface contains the declaration of the methods used to perform simple CRUD operations using the Product entity.
- **ProductRepository:** The ProductRepository class extends the IProductRepository interface and implements its members.
- **ProductController:** This is the controller class that contains all the action methods.

Create the Model

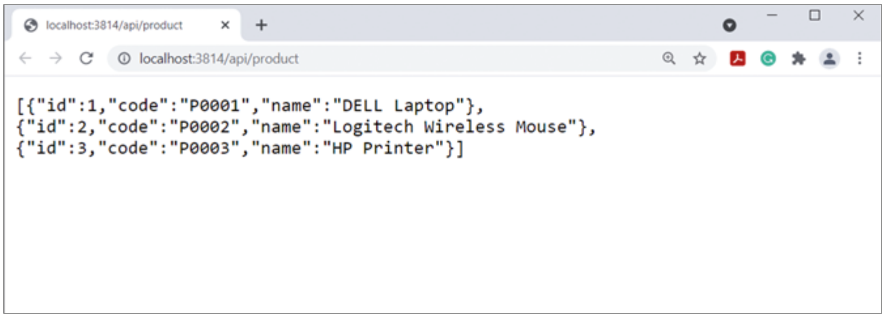
Create a file named Product.cs with the following code in there:

```
public class Product
    {
        public int Id { get; set; }
        public string Code { get; set; }
        public string Name { get; set; }
    }
```

Create the Product Repository

Create an interface named IProductRepository in a file named IProductRepository.cs with the following code in there:

```
public interface IProductRepository
    {
        Task<List<Product>> GetProducts();
        Task<Product> GetProduct(int id);
        Task<bool> AddProduct(Product product);
        Task<bool> DeleteProduct(int id);
    }
```



```
[{"id":1,"code":"P0001","name":"DELL Laptop"},
{"id":2,"code":"P0002","name":"Logitech Wireless Mouse"},
{"id":3,"code":"P0003","name":"HP Printer"}]
```

Figure 1: The RESTful API in action

The ProductRepository class pertaining to the ProductRepository.cs file extends the IProductRepository interface and implements its methods as shown in **Listing 1**.

Add the Dependencies

The following code snippet illustrates how an instance of the ProductRepository class is added as a scoped service in the Startup class so it can be used in the controller using dependency injection.

```
public void ConfigureServices
(IServiceCollection services)
{
    services.AddScoped<IProductRepository,
    ProductRepository>();
    services.AddControllers();
}
```

You can now leverage dependency injection in your controller class to retrieve an instance of ProductRepository at runtime.

Listing 1: The ProductRepository Class

```
public class ProductRepository: IProductRepository
    {
        private readonly List<Product> products =
        new List<Product>();

        public ProductRepository()
        {
            products.Add(new Product
            {
                Id = 1,
                Code = "P0001",
                Name = "DELL Laptop"
            });

            products.Add(new Product
            {
                Id = 2,
                Code = "P0002",
                Name = "Logitech Wireless Mouse"
            });

            products.Add(new Product
            {
                Id = 3,
                Code = "P0003",
                Name = "HP Printer"
            });
        }

        public Task<List<Product>> GetProducts()
        {
            return Task.FromResult(products);
        }
        public Task<Product> GetProduct(int id)
        {
            return Task.FromResult(products.
            Where(x => x.Id == id).SingleOrDefault());
        }
        public Task<bool> AddProduct(Product product)
        {
            products.Add(product);
            return Task.FromResult(true);
        }
        public Task<bool> DeleteProduct(int id)
        {
            products.Remove(products.
            Where(x => x.Id == id).SingleOrDefault());
            return Task.FromResult(true);
        }
    }
```

Listing 2: The DefaultController Class for the RESTful API

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace InsomniaRESTClient.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class DefaultController : ControllerBase
    {
        private readonly IProductRepository _productRepository;

        public DefaultController(IProductRepository
        productRepository)
        {
            _productRepository = productRepository;
        }

        [HttpGet]
        public async Task<ActionResult<List<Product>>> Get()
        {
            return await _productRepository.GetProducts();
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Product>> Get(int id)
        {
            return await _productRepository.GetProduct(id);
        }

        [HttpPost]
        public async Task<ActionResult<bool>> Post([FromBody]
        Product product)
        {
            return await _productRepository.AddProduct(product);
        }

        [HttpDelete("{id}")]
        public async Task<ActionResult<bool>> Delete(int id)
        {
            return await _productRepository.DeleteProduct(id);
        }
    }
}
```

Create the API Controller

Create a new API controller named DefaultController and replace the default autogenerated code using the code shown in **Listing 2**.

Run the Application

Now run the application by pressing Ctrl + F5 or just F5. The application starts and the Web browser displays the data shown in **Figure 1**:

Configuring Environment Variables in Insomnia

While communicating with APIs, you often need to manually type certain data across multiple requests. Here's exactly where environment variables come to the rescue. An environment here refers to a JSON object that contains data represented as key-value pairs. You can take advantage of environment variables to define a variable once and then reference its value wherever it's needed. You can access the environment manager through the drop-down menu at the top of the sidebar, as shown in **Figure 2**:

You can take advantage of the environment manager to edit the base environment, create sub environments, etc. You can click on Manage Environments to add or edit an environment. **Figure 3** shows how you can edit the Base environment.

Test RESTful API in Insomnia

In this section, I'll examine how you can make GET and POST requests using Insomnia.

Send a GET Request

To create a GET request, follow the steps outlined below:

1. Launch the Insomnia application.
2. Click on the "New Request" button.
3. In the "New Request" window that pops up, specify the name of the request and select a request method.
4. Click Create.

Now follow the steps given below to test the API using Insomnia REST Client:

1. Ensure that the Web API application is up and running.
2. Launch the Insomnia REST Client.
3. Ensure that the HTTP method GET is selected (it's the default).
4. Specify the URL in the address bar.
5. Click Send.

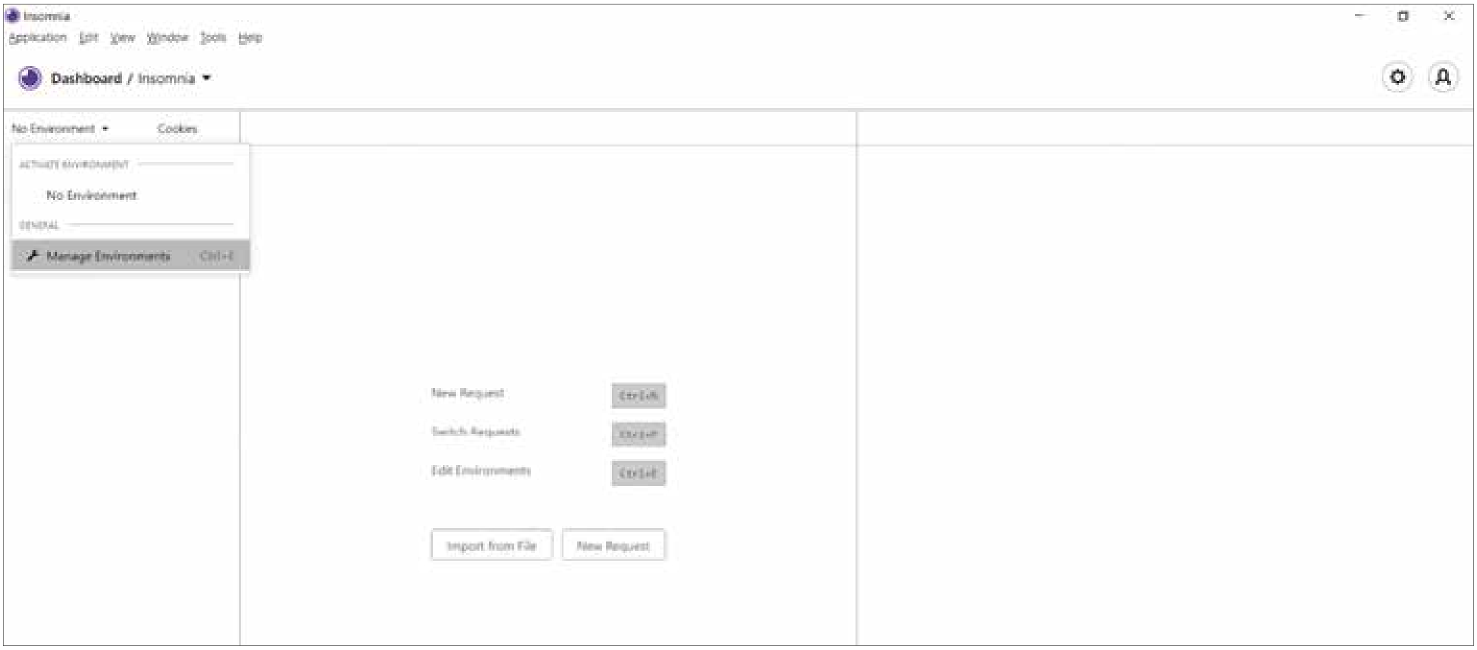
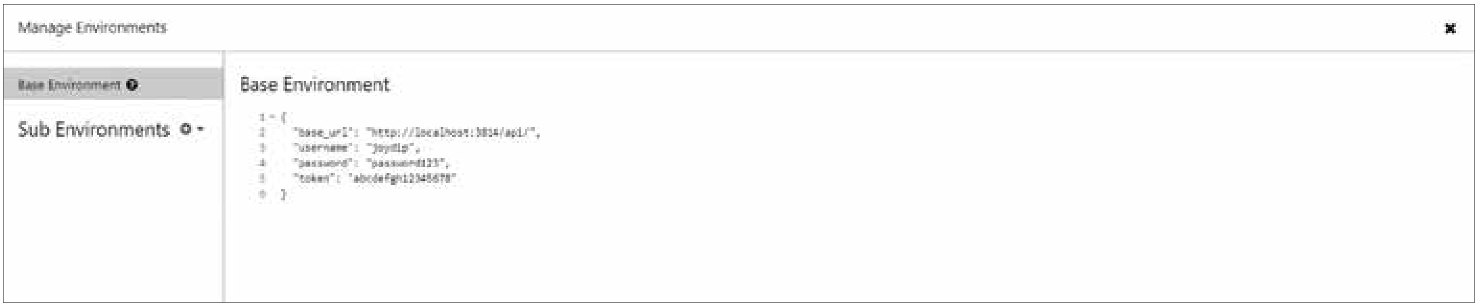
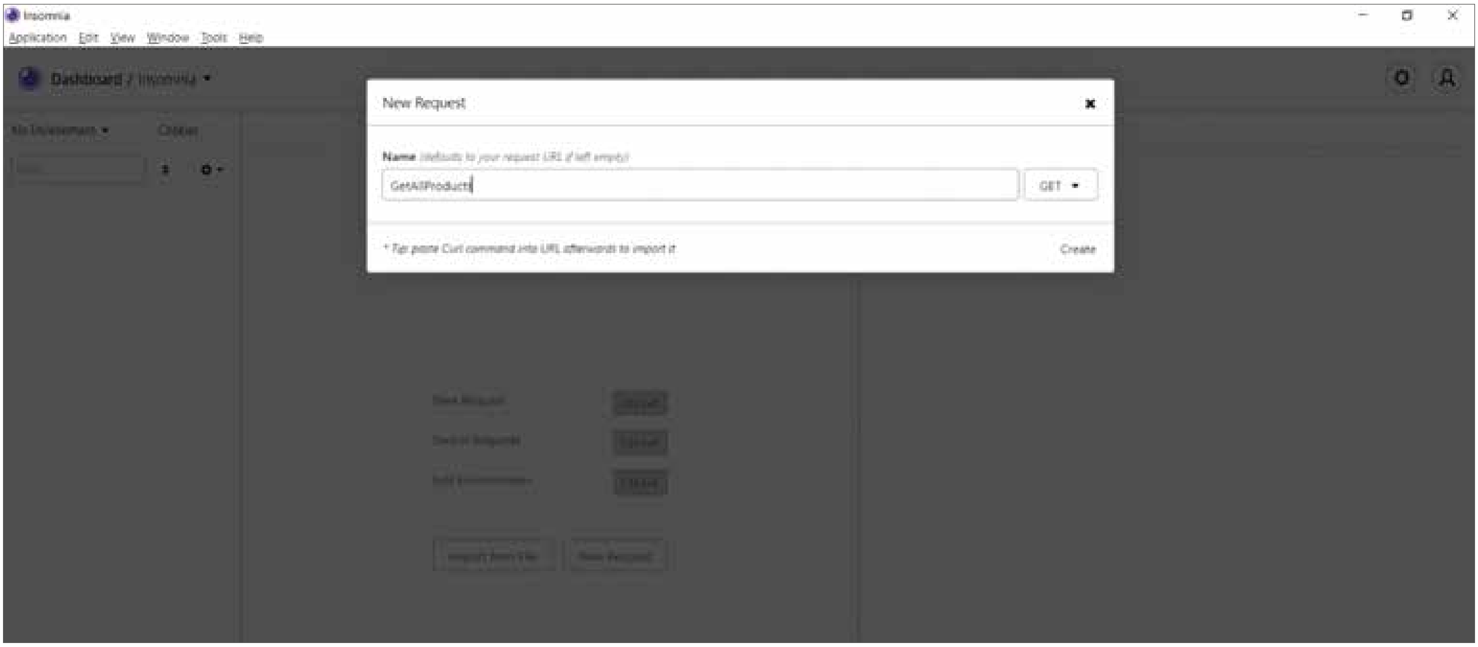


Figure 2: Manage the environment in Insomnia.



```
{
    "base_url": "http://localhost:3834/api/",
    "username": "joydip",
    "password": "password123",
    "token": "abcdefgh12345678"
}
```

Figure 3: Edit the Base environment.

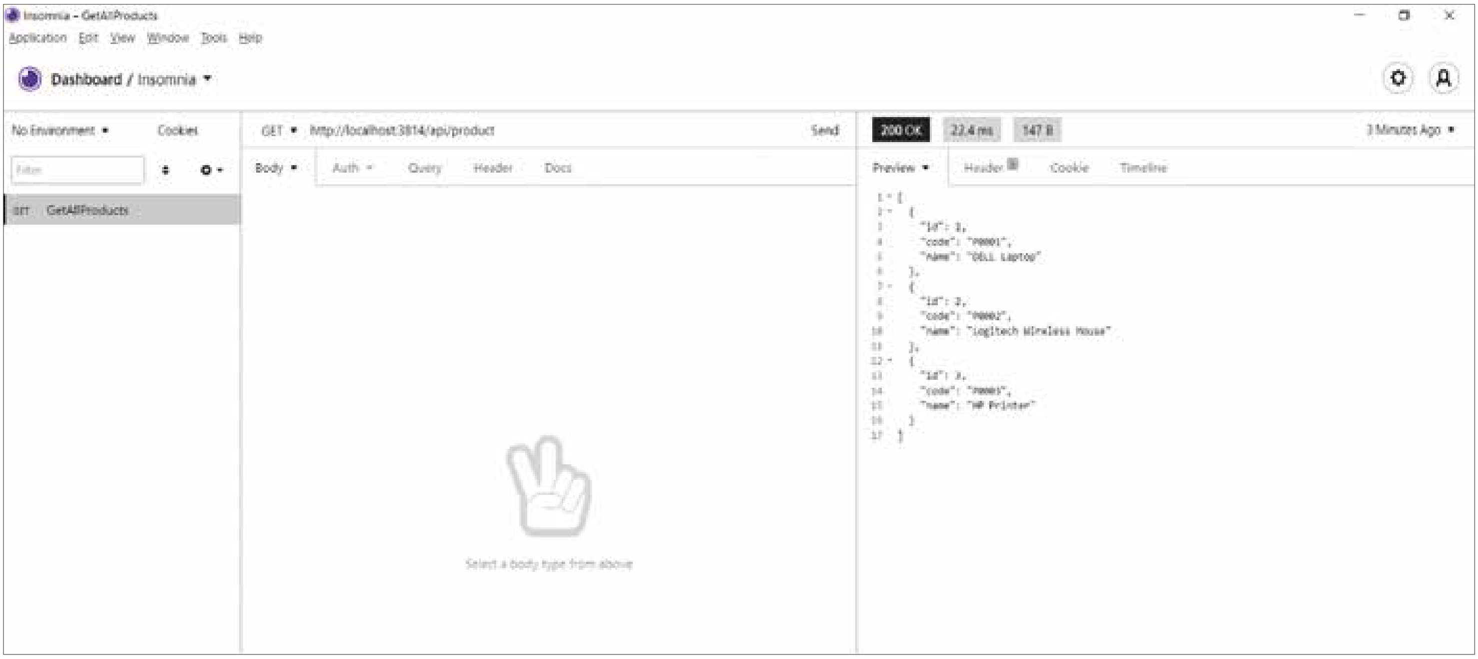


Name (defaults to your request URL if left empty): GetAllProducts

GET

* Tip: paste Curl command into URL afterwards to import it

Figure 4: Create a new request.



GET http://localhost:3814/api/product

```
[
  {
    "id": 1,
    "code": "P0001",
    "name": "DELL Laptop"
  },
  {
    "id": 2,
    "code": "P0002",
    "name": "Logitech Wireless Mouse"
  },
  {
    "id": 3,
    "code": "P0003",
    "name": "HP Printer"
  }
]
```

Figure 5: Displaying the list of products in Insomnia

Figure 5 shows the output—the list of the products is displayed in the right-side panel.

Send a POST Request

Assuming that Insomnia is up and running, to send a POST request, follow the steps outlined below:

1. Specify the URL in the address bar.
2. Specify the JSON data in the request body.
3. Click Send.

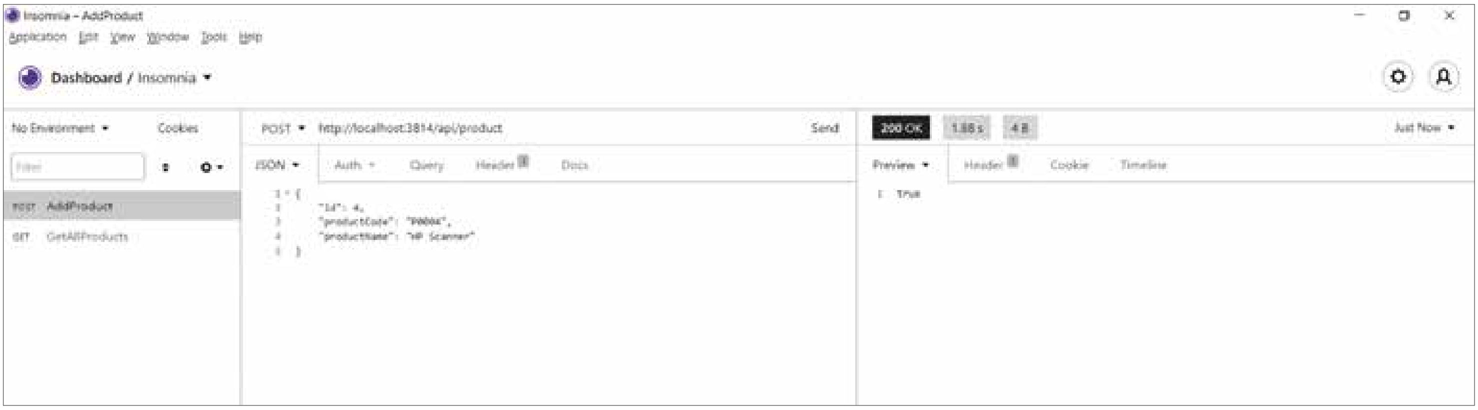
Return the response returned as true, indicating that the POST request is successful, as shown in **Figure 6**.

Create a Minimalistic GraphQL API

In this section, I'll examine how you can test a GraphQL API using Insomnia. Follow the steps mentioned earlier in this article to create another ASP.NET Core Web API project. Both of these projects can be part of the same solution. There are certain classes and interfaces you'll reuse from the previous example.

Here's the list of the classes and interfaces you'll reuse from the earlier example:

- **Product**: This is the entity class.
- **IProductRepository:** This is the interface for your repository class that contains the declarations of the methods.



```
POST http://localhost:3814/api/product
```

```
{
    "Id": 4,
    "productCode": "P0004",
    "productName": "HP Scanner"
}
```

```
true
```

Figure 6: Execute a POST request in Insomnia.

- **ProductRepository:** This is your only repository class that extends the IProductRepository interface and implements its members.

Schemas and Types

The main building blocks of GraphQL are schemas and types. A schema extends the GraphQL.Types.Schema class and represents the functionality exposed via an endpoint for the clients of the API to consume. Note that there's only one endpoint in GraphQL.

A schema comprises Query, Mutation, and a Subscription. Queries are used to consume data in an efficient manner. Mutations are used to send data to the server for performing CRUD operations. Subscriptions enable data to be sent back to the client.

With this knowledge, you can proceed with creating your minimalistic GraphQL API.

Configure the GraphQL Middleware

Support for GraphQL isn't available in ASP.NET Core by default, i.e., it isn't a built-in feature. Hence, you should install the following NuGet packages to work with GraphQL:

```
Install-Package GraphQL
Install-Package GraphiQL
```

Because support for GraphQL in ASP.NET Core is an opt-in feature and isn't enabled by default, write the following code in the Configure method of the Startup class to enable the graphql endpoint:

```
app.UseGraphiQl("/graphql");
```

Build the GraphQL Schema

To be able to query data using GraphQL, you should be able to create a type that extends ObjectGraphType<T>, as shown in **Listing 3**.

Create Your Query Type

You also need a class that retrieves data. To do this, create a class named ProductQuery that extends the ObjectGraphType class, as in **Listing 4**.

Note that when you're working with GraphQL, the client always makes an HTTP POST call and passes the query name, name of the operation, and variables. You need a POCO class to manage schema, variables, and the argument, as shown in **Listing 5**.

Listing 3: The ProductType Class

```
public class ProductType : ObjectGraphType<Product>
    {
        public ProductType()
        {
            Name = "Product";
            Field(_ => _.Id).Description("Product ID.");
            Field(_ => _.Name).Description("Product Name");
            Field(_ => _.Description).Description
            ("Product Description");
        }
    }
```

Listing 4: The ProductQuery Class

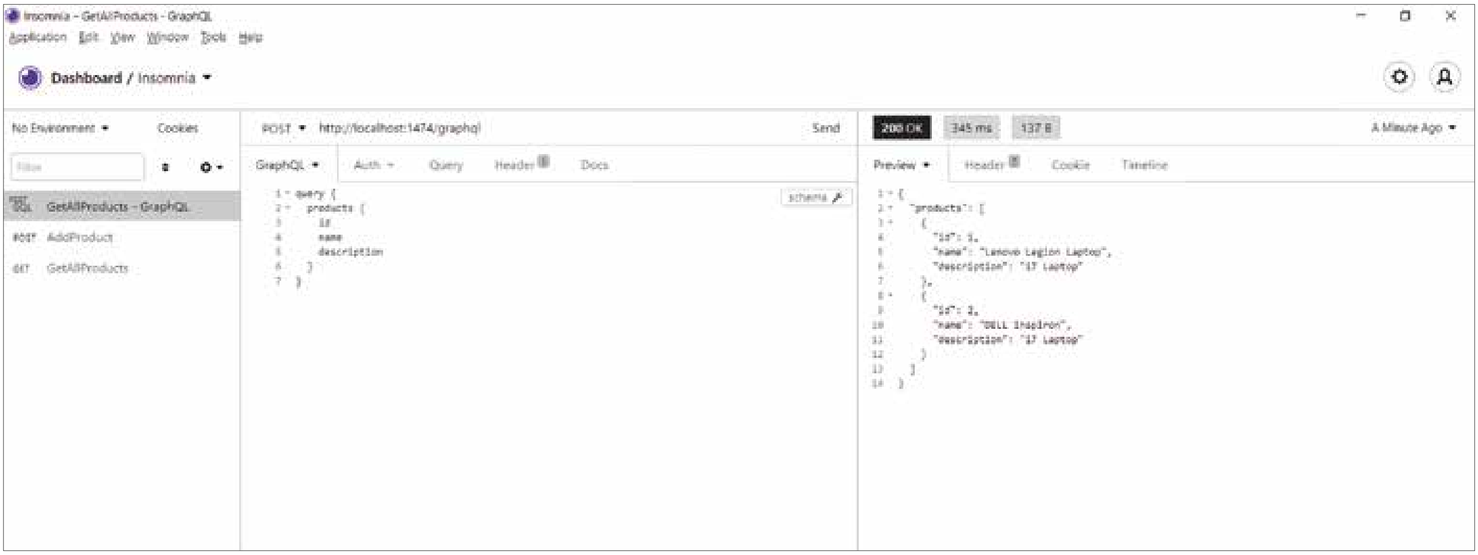
```
public class ProductQuery : ObjectGraphType
    {
        public ProductQuery(ProductRepository
        productRepository)
        {
            Field<ListGraphType<ProductType>>(
            name:"products", resolve: context =>
            {
                return productRepository.GetProducts();
            });
        }
    }
```

Listing 5: The GraphQueryDTO Class

```
    public class GraphQLQueryDTO
    {
        public string OperationName { get; set; }
        public string NamedQuery { get; set; }
        public string Query { get; set; }
        public string Variables { get; set; }
    }
```

Listing 6: The ConfigureServices method

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<IDependencyResolver>
             (_ => new FuncDependencyResolver
             (_.GetRequiredService));
            services.AddScoped<IDocumentExecuter,
            DocumentExecuter>();
            services.AddScoped<ISchema, GraphQLDemoSchema>();
            services.AddScoped<IDocumentWriter, DocumentWriter>();

            services.AddScoped<IProductRepository,
            ProductRepository>();
            services.AddScoped<ProductQuery>();
            services.AddScoped<ProductType>();
            services.AddControllers();
        }
```



```
query {
  products {
    id
    name
    description
  }
}
```

```
{
  "products": [
    {
      "id": 1,
      "name": "Lenovo Legion Laptop",
      "description": "i7 Laptop"
    },
    {
      "id": 2,
      "name": "DELL Inspiron",
      "description": "i7 Laptop"
    }
  ]
}
```

Figure 7: Executing GraphQL API in Insomnia

Listing 7: The DefaultController Class for our GraphQL API

```
[Route("graphql")]
public class DefaultController : ControllerBase
{
    private readonly ISchema _schema;
    private readonly IDocumentExecuter _executer;
    public DefaultController(ISchema schema,
    IDocumentExecuter executer)
    {
        _schema = schema;
        _executer = executer;
    }

    [HttpPost]
    public async Task<IActionResult> Post([FromBody]
    GraphQLQueryDTO query)
    {
        var result = await _executer.ExecuteAsync(_ =>
        {
            _.Schema = _schema;
            _.Query = query.Query;
            _.Inputs = query.Variables?.ToInputs();

        });

        if (result.Errors?.Count > 0)
        {
            return BadRequest();
        }

        return Ok(result.Data);
    }
}
```

Source Code

The complete source code of the demo application built throughout this article is available here: https://github.com/joydipkanjilal/insomnia

Add Services to the Container

Write the code shown in **Listing 6** in the ConfigureServices method to add services to the built-in IoC container.

Create the Controller Class

So far, so good. You now need to create the GraphQL endpoint. Create a new API controller named DefaultController with the code mentioned in **Listing 7** in there.

Test GraphQL API in Insomnia

Here's an example GraphQL query:

```
query {
  products {
    id
    name
    description
  }
}
```

Now execute the application and browse to the /graphql endpoint. You can also execute this query here using the GraphiQL tool but you'll execute the GraphQL query using Insomnia. You can easily create a GraphQL request in Insomnia either choosing the GraphQL request type while creating the request or by modifying the body type of the request using the body menu.

To execute the GraphQL endpoint using Insomnia, follow the steps outlined below:

1. Launch Insomnia.
2. Create a new request and name it GraphQL (you can provide any name).
3. Select HttpPOST as the HTTP method.
4. Select GraphQL request type.
5. Specify your GraphQL query in there.
6. Click Send.

Figure 7 shows how output looks in the Insomnia user interface.

Summary

GraphQL is technology and database agnostic, which means that it can be used with most common technologies, frameworks, or platforms. Insomnia provides support for testing both RESTful as well as GraphQL APIs. You can learn more about Insomnia here: https://support.insomnia.rest/.

Joydip Kanjilal

ONLINE QUICK ID 2107061

Building Command Line Utilities in C# and Python

A few months ago, I received an email from a friend requesting some technical help. The following text is a copy of the email he sent me (names have been excluded to protect the innocent, LOL)

Rod Paddock
craigshoemaker.net
@craigshoemaker

Craig Shoemaker is a developer, author, speaker, and Senior Content Developer for Microsoft on the Azure Functions team. From building samples, internal tools, and writing

Craig hel

Bio Text
is missing

a Pluralsight author, Craig specializes in teaching JavaScript, HTML5, and IndexedDB.

In the future, Craig wants to learn how to tell a joke.

"I have a command line tool installed through Homebrew on my laptop running High Sierra. The command is just ccextractor <filepath> and it runs fine in a standard bash terminal. I was hoping to use Automator to be able to run it on batches of files, but I'm struggling with the syntax for the Run Shell Script command. It just keeps saying ccextractor command not found. Also, the command line tool can only process one file at a time, so I guess I need some way to loop the request so it can process more than one file."

My friend, like me, is a movie aficionado with an extensive collection of movies, many of which are foreign titles with subtitles. When copying files to systems like Plex, you need this subtitle information so you can see it when you watch the films. This is where CCExtractor comes in. CCExtractor (https://www.ccextractor.org/) is an application used to extract closed captions from video files.

The problem was that my friend couldn't figure out how to use Automator (a Mac tool) to run this command on a directory of files. An attempt was also made to use Bash with no luck.

I told him that I could probably whip something up in Python, if that would work. "Are you sure that's not too much trouble?" my friend asked. "Nah, it should be pretty simple to whip up," I replied.

Here's what I did.

1. I navigated to the https://www.ccextractor.org/ site and downloaded the binaries and some 3.x GB sample files to my drive.
2. I opened my trusty text editor (https://www.sublimetext.com/) and created a new Python program.
3. After a bit of Googling, I came up with this set of code:

```
import os
import subprocess

directory_to_import =
 'D:/Data/clients/RodPaddock/CCExtractor/'
extractor_exe_path =
 'D:/Data/clients/RodPaddock/CCExtractor
/ccextract orwin'
for file in os.listdir(directory_to_import):
  if file.endswith(".mpg"):
    print(os.path.join(
    directory_to_import,  file))
    subprocess.run(
    [extractor_exe_path,
     os.path.join(
    directory_to_import, file)])
```

This code was built, debugged, and run on my Windows development box. The goal was to get it working as fast as possible on my main development box before moving it onto a Mac.

Here's a link to the Gist of the code: https://gist.github.com/rjpaddock/d53956767dd4a1fe267dee08c995c956.js.

Getting the code to run on the Mac was simple. Here's the Mac version:

```
import os
import subprocess
directory_to_import = '/Users/rodpaddock/ccextractor'
extractor_exe_path = 'ccextractor'
for file in os.listdir(directory_to_import):
  if file.endswith(".mpg"):
    print(os.path.join(directory_to_import, file))
    subprocess.run([extractor_exe_path,
    os.path.join(directory_to_import, file)])
```

As you can see, the changes were minimal, at best. I changed the path to my user directory on the Mac and got rid of the specific path to the executable. I used Homebew to install the CCExtractor on my Mac so it was in the PATH already. After installing a version of Python on my Mac, I was able to run the application as-is. No operating system-specific issues. After getting my program to work, I sent it to my friend, who simply changed the path to the files he wished to decode, and BOOM. It just worked.

Running on Windows

After marveling at how much could be accomplished with so few lines of code, I became curious to see how complex it would be to build the same application in C#. I'm using .NET Core to do this, as I want to run it cross-platform, as well. The code in **Listing 1** represents the same functionality in C#.

I'd say this wasn't too bad. Building the same application was pretty simple as a C# console application. Here's a Gist to the C# code: https://gist.github.com/rjpaddock/be601db3995082949071121d8aa992d7.

With a minimal set of code, I thought it would be fun to explore making it a bit more robust. Here's the set of features I planned to add:

- Accept an extension parameter. The original code had the extension hard-coded.
- Accept a path to the files I wished to decode.
- Accept the path to the executable as a parameter.
- Parameters should be named vs. positional, if possible.
- Run this code on Windows, Mac, and Linux.

I started with the Python program and the first feature on the list, specifying the extension as a parameter. My initial

Listing 1: The .NET Core version of my app

```
using System;
using System.Diagnostics;
using System.IO;

namespace ExtractorRunner
{
  class Program
  {
  static void Main(string[] args)
   {
    var directory_to_import =
   "D:/Data/clients/RodPaddock/CCExtractor/";
    var extractor_exe_path =
    "D:/Data/clients/RodPaddock
    /CCExtractor/ccextractorwin";
    foreach (var fileName in
     Directory.GetFiles(directory_to_import,"*.mpg"))
     {
       Console.WriteLine(fileName);
       var process = new Process()
    {
     StartInfo = new ProcessStartInfo
     {
     FileName = $"{extractor_exe_path}",
     Arguments = $"{fileName}",
     UseShellExecute = true,
    }
   };
    process.Start();
    }
   }
  }
 }
```

choice was to process **mpg** files as the default extension. My friend immediately changed it to **mp4**. With this knowledge, I realized that this would be the first thing to parameterize.

There are multiple ways this could be implemented. One way we could hack this together would be to use Python's **sys.argv[]** array, which provides positional arguments to Python programs. For instance, let's say you called the program with the following statement:

```
python copy run_cc.py .mp4
```

Then you could access the **.mp4** with **sys.arg[0]**. Although this works, it'll cause problems in the long haul if you add or remove parameters. It's also not very intuitive. It would be better to call the program with a named parameter. For example:

```
run_cc.py --extension .mp4
```

Luckily for us, Python has a built-in library to do this exact thing. This library is known as ***argparse***. To implement the first option, you need to do the following:

1. Add an **import argparse** to the imports section of the program.
2. Create an argument parser object and add an argument to it. Your code will look like this:

```
parser = argparse.ArgumentParser ()
parser.add_argument("--extension",
  help="Extension of files to convert",
  default='.mpg')
  args = parser.parse_args()
```

There's a lot going on with just these few lines of code. What this set of code does is:

- Creates an argument parser.
- Adds a parameter called **--extension** to the command line.

This parameter will be added the **args** array as a property with the name **extension**. Finally, this code specifies a help description and a default parameter value. The program code now looks like **Listing 2**.

Listing 2: The new code

```
import os
import subprocess
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--extension",
help="Extension of files  to convert",
default='.mpg')
args = parser.parse_args()

directory_to_import =
'D:/Data/clients/RodPaddock/CCExtractor/'
extractor_exe_path =
'D:/Data/clients/RodPaddock/CCExtractor/ccextractorwin'

for file in os.listdir(directory_to_import):
  if file.endswith(args.extension):
    print(os.path.join(directory_to_import, file))
    subprocess.run([extractor_exe_path,
  os.path.join(directory_to_import, file)])
```

Listing 3: The Python code

```
import os
import subprocess
import argparse

parser = argparse.ArgumentParser()
 parser.add_argument("--extension",
 help="Extension of files to convert",
  default='.mpg')

parser.add_argument("--directory",
  help="Directory to process",
  default='.')

args = parser.parse_args()

extractor_exe_path =
'D:/Data/clients/RodPaddock/CCExtractor/ccextractorwin'
for file in os.listdir(args.directory):
  if file.endswith(args.extension):
    print(os.path.join(args.directory, file))
    subprocess.run([extractor_exe_path,
    os.path.join(args.directory, file)])
```

The next step is to add a parameter to specify the directory you wish to read files from. Call the parameter **--directory** and default it to (.) the current working directory. A sample call would be as follows:

```
python run_cc.py
    --extension .mp4
    --directory  "D:/Data/clients/RodPaddock/CCExtractor/"
```

Your Python code now looks like **Listing 3**.

Finally, let's get rid of the EXE path. I'm going to cheat a bit on this one. I'm simply going to add the directory where the CCExtractor application is located to my system's PATH statement. This will take care of that issue much like Homebrew did on the Mac.

To change your PATH statement in Windows, open the Environmental Variables from the Windows Start menu. find PATH in the System variables and add the path to wherever you extracted the ccextractor application. The **Figure 1** demonstrates how this should look.

Now the final Python program looks like **Listing 4**.

The next step is to implement the same functionality in the C# application. Python has an argument parser built into its native libraries, but the .NET platform doesn't. Not to fear, there's a third-party library that you can install to add this needed functionality. This library is called **CommandLineParser** and can be installed via a **NuGet** package. You can install this library via the NuGet console by issuing the following command:

```
Install-Package CommandLineParser -Version 2.8.0
```

Listing 4: The final Python program

```
import os
import subprocess
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--extension",
  help="Extension of files to convert",
  default='.mpg')
parser.add_argument("--directory",
help="Directory to process",
default='.')

args = parser.parse_args()

# should be added to the system PATH statement
extractor_name = 'ccextractorwin'
for file in os.listdir(args.directory):
  if file.endswith(args.extension):
    print(os.path.join(args.directory, file))
    subprocess.run([extractor_name,
    os.path.join(args.directory, file)])
```

Listing 5: The new processing logic

```
var directory_to_import =
  "D:/Data/clients/RodPaddock/CCExtractor/";
var extractor_exe_path =
  "D:/Data/clients/RodPaddock/CCExtractor/ccextractorwin";
foreach (var fileName in
  Directory.GetFiles(options.Directory,
  $"*{options.Extension}"))
{
  Console.WriteLine(fileName);
  var process = new Process()
  {
    StartInfo = new ProcessStartInfo
    {
    FileName = $"{extractor_exe_path}",
    Arguments = $"{fileName}",
      UseShellExecute = true,
     }
   };
  process.Start();
 }
```

Once you've installed this library, you need to build a class that will hold your parsed command line parameters. This class will be augmented with ***Attributes*** provided by the command line parser. The first parameter to add is the dynamic extension. To do this, add the following class code to the program:

```
public class Options
{
  [ Option(longName:"extension",
    HelpText = "Extension of files to convert",
    Default = ".mpg")]
    public string Extension { get; set; } = "";
}
```

This code has a string property called Extension. When you pass in the **–extension** parameter to your application, it's stored on this property. The more interesting aspect of this class is the [Option] attribute.

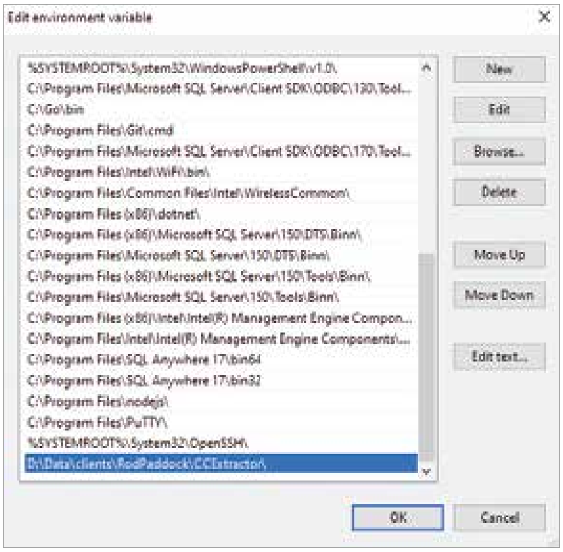
```
[Option(longName:"extension",
 HelpText
= "Extension of files to convert",Default =
    ".mpg")]
```

The longName property tells the **CommandLIneParser** library to parse an argument with the name **–extension** onto the Extension parameter. The **HelpText** and **Default** properties are self-explanatory.

Now that you've created this class, you can call the command line parser to populate your arguments onto an instance of the Options class. This code demonstrates how to do this:

```
var parsed=
   Parser.Default.ParseArguments<Options>(args);
var options=((Parsed<Options>) parsed).Value;
```

This code takes the **args** collection passed to your application, parses them, and returns a parsed object. After parsing the argument collection, you need to cast the **Value** property of the parsed object into an instance that you can



```
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\130\Tool...
C:\Go\bin
C:\Program Files\Git\cmd
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tool...
C:\Program Files\Intel\WiFi\bin\
C:\Program Files\Common Files\Intel\WirelessCommon\
C:\Program Files (x86)\dotnet\
C:\Program Files (x86)\Microsoft SQL Server\150\DTS\Binn\
C:\Program Files\Microsoft SQL Server\150\DTS\Binn\
C:\Program Files (x86)\Microsoft SQL Server\150\Tools\Binn\
C:\Program Files\Microsoft SQL Server\150\Tools\Binn\
C:\Program Files (x86)\Intel\Intel(R) Management Engine Compon...
C:\Program Files\Intel\Intel(R) Management Engine Components\...
C:\Program Files\SQL Anywhere 17\bin64
C:\Program Files\SQL Anywhere 17\bin32
C:\Program Files\nodejs\
C:\Program Files\PuTTY\
%SYSTEMROOT%\System32\OpenSSH\
D:\Data\clients\RodPaddock\CCExtractor\
```

Figure 1: Changes to PATH statement in Environmental Variables Screen

Listing 6: The final version of the new Directory option

```
using System;
using System.Diagnostics;
using System.IO;
using CommandLine;

namespace ExtractorRunner
{
  class Program
  {
    static void Main(string[] args)
    {

    var parsed = Parser.Default.ParseArguments<Options>(args);
    var options = ((Parsed<Options>)  parsed).Value;

      var extractor_exe_path =
          "ccextractorwin";
      foreach (var fileName in
          Directory.GetFiles(options.Directory,
          $"*{options.Extension}"))
      {
        Console.WriteLine(fileName);
        var process = new Process()
        {
          StartInfo = new  ProcessStartInfo
          {
            FileName =  $"{extractor_exe_path}",
            Arguments =  $"{fileName}",
            UseShellExecute =  true,
          }
        };
        process.Start();
      }

    }
    public class Options
    {
      [Option(longName:"extension",
          HelpText =  "Extension of files to convert",
        Default = ".mpg")]
      public string Extension { get; set; } =  "";

      [Option(longName: "directory",
      HelpText =  "Directory to process",
      Default = ".")]
      public string Directory { get; set; } =  ".";
    }
  }
}
```

use in your programming code. Your processing logic now looks like **Listing 5**.

Notice that the GetFiles() function now uses the Extension property of your Options class.

The next step is to add the directory to your Options class. To do this, simply add another property to your class with the appropriate name and options. Your class code will now look like this:

```
public class Options
{
  [Option(longName:"extension",
   HelpText = "Extension of files to convert",
   Default = ".mpg")]
   public string Extension { get; set; } = "";

  [Option(longName: "directory",
   HelpText = "Directory to process",
   Default = ".")]
   public string Directory { get; set; } = ".";
}
```

Notice that the DefaultValue property is a single period (.). This tells the GET files routine to simply process the current directory.

Now you can incorporate your new Directory option into your application code. **Listing 6** is what the final version will look like.

One item of note is that the path to the EXE is just the name of the application. This is because in the last post, I decided to add the Ccexteactorwin.exe file to the system PATH via the System Environment variables screen.

You can now run your code from Visual Studio. When testing your code, you can call your application with arguments by opening your Project Properties Window, selecting the Debug section, and specifying the command line parameters in the Arguments section. **Figure 2** shows that.

The running program will now spawn a new process that looks like **Figure 3.**

At this point, we have a pair of programs written in Python and C#. These programs are used to run the CCextractor program with extension and path parameters. The next step in the evolution is to run the code on other platforms, namely macOS and Linux. I'll demonstrate running code on both of those platforms.

Running on macOS

Before you start working on the code, you'll need to get your Mac set up to install the CCExtractor application and Python 3 code.

Installing the extractor is simple and is done via the Homebrew infrastructure used by Mac developers. To install the CCExtractor, do the following:

Install Homebrew if it isn't already installed. Run this script (copied from https://brew.sh/) from a terminal window.

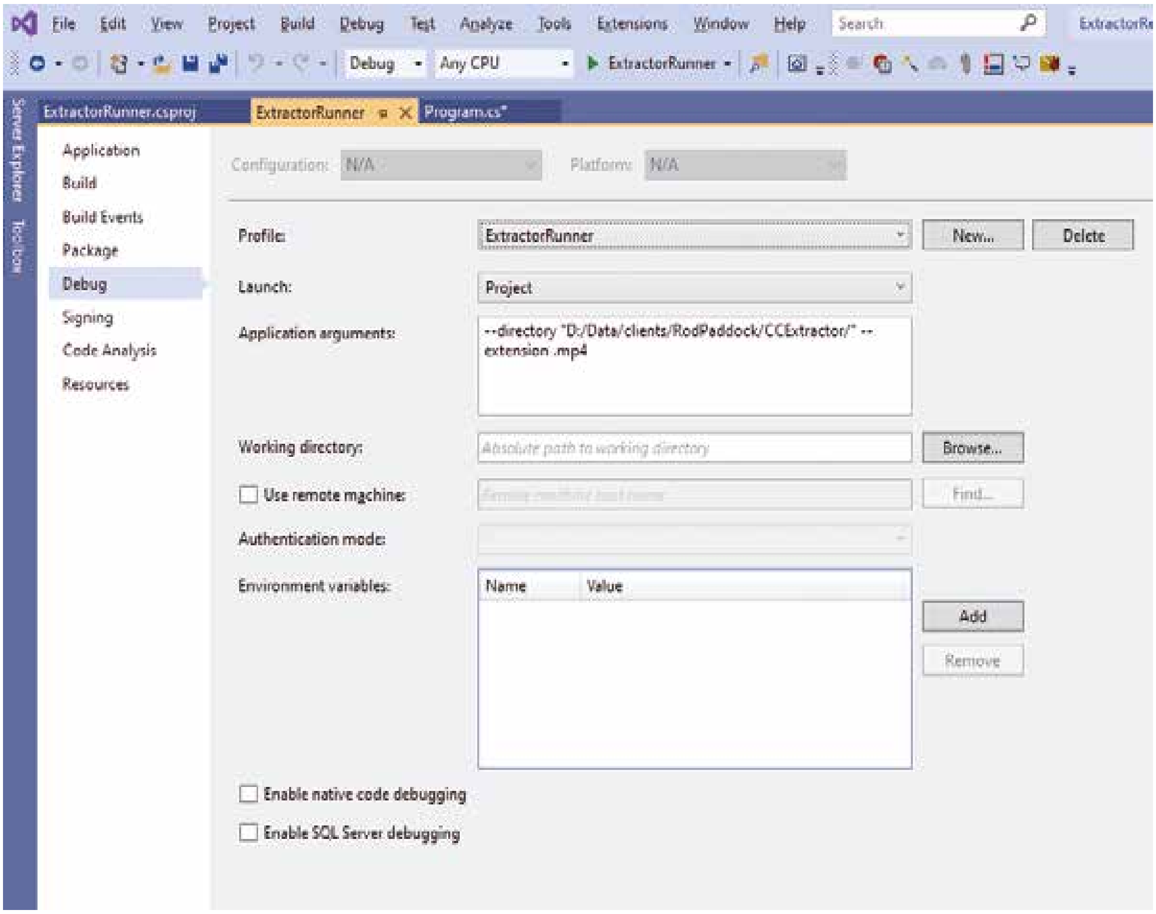
```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com
/Homebrew/install/master/install.sh)"
```

1. Install the CCExtractor program by issuing the following command:

```
brew install CCExtractor
```

2. Test it by typing CCExtractor from the terminal window. You should see a screen of help information.
3. Now insure that Python3 is installed. From a terminal window type: **python3**

If Python 3 is installed, you'll see the Python's interactive window. If not, you may be promoted to install the Command Line tools for OSX. If so, run that installer. If the Command Line Tools installer doesn't run, directions for installing Python 3 can be found here: https://docs.python-guide.org/starting/install3/osx/.



Configuration: N/A
Platform: N/A

Profile: ExtractorRunner

Launch: Project

Application arguments: --directory "D:/Data/clients/RodPaddock/CCExtractor/" --extension .mp4

Working directory: Absolute path to working directory

☐ Use remote machine:

Authentication mode:

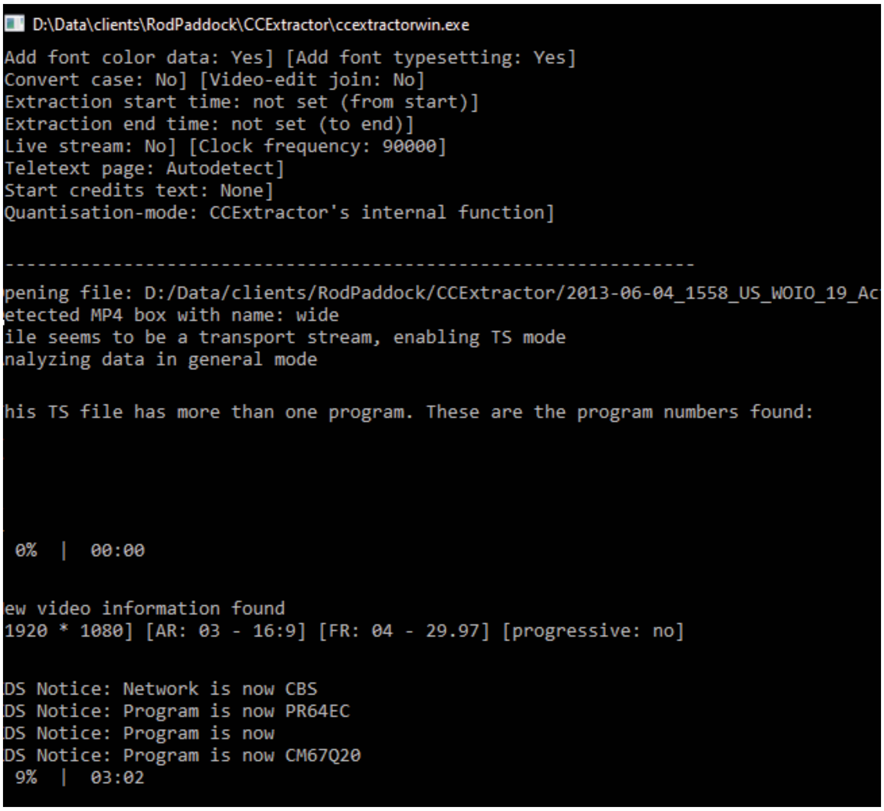
Environment variables:

Name	Value

☐ Enable native code debugging

☐ Enable SQL Server debugging

Figure 2: Specifying parameters in the arguments section



D:\Data\clients\RodPaddock\CCExtractor\ccextractorwin.exe

```
Add font color data: Yes] [Add font typesetting: Yes]
Convert case: No] [Video-edit join: No]
Extraction start time: not set (from start)]
Extraction end time: not set (to end)]
Live stream: No] [Clock frequency: 90000]
Teletext page: Autodetect]
Start credits text: None]
Quantisation-mode: CCExtractor's internal function]

---------------------------------------------------------------
pening file: D:/Data/clients/RodPaddock/CCExtractor/2013-06-04_1558_US_WOIO_19_Ac
etected MP4 box with name: wide
ile seems to be a transport stream, enabling TS mode
nalyzing data in general mode

his TS file has more than one program. These are the program numbers found:



 0%  |  00:00

ew video information found
1920 * 1080] [AR: 03 - 16:9] [FR: 04 - 29.97] [progressive: no]

DS Notice: Network is now CBS
DS Notice: Program is now PR64EC
DS Notice: Program is now
DS Notice: Program is now CM67Q20
 9%  |  03:02
```

Figure 3: The new process

Now it's time to test the code. Clone this repo:

```
https://github.com/rjpaddock/ExtractorRunner.git
```

From a terminal window, change into the folder where you cloned that repo and run the following command:

```
python3 run_cc.py
--extension mpg --directory
[[INSERT YOUR DIRECTORY HERE]]
```

You'll be presented with the error information in **Listing 7**.

This error is because the name of the CCExtractor application is different in the Windows environment. Check out the last line. What's the fix for this?

To fix this, you need to call a different executable based on the operating system. Luckily for us, Python has a built-in library for just such a thing. To check which platform your code is running on, import the platform library at the top of your python file:

```
import platform
```

Next, add the following code to your script:

```
extractorname  = ''
if platform.system() == 'Windows':
  extractor_name = 'ccextractorwin'
elif platform.system() == 'Darwin':
  extractor_name = 'ccextractor'
elif platform.system() == "Linux":
  extractor_name = 'ccextractor'
```

Now run the application. Your script should start processing files with no error. **NOTE:** The code in the Repository already has this change applied. You're welcome.

The next step is to get the C# code up and running on the Mac. This process was much easier than I anticipated, as Microsoft has created a Mac version of Visual Studio. The first step is to install Visual Studio Mac from the Microsoft website: https://visualstudio.microsoft.com/vs/mac/.

When installing the application, make sure to install it with the .NET Core option selected, as shown in **Figure 4**.

Once the installer completes, open the ExtractorRunner solution from the folder you pulled code into. Open the options dialog for the project and set the command line parameters you've been using to test, as shown in **Figure 5**.

Run your code now. You'll now see an error in the console window of your application, like that in **Figure 6**.

This is very similar to the Python error and requires the same solution. .NET Core also included a set or libraries to determine your operating system. Add the following snippet to the top of your program:

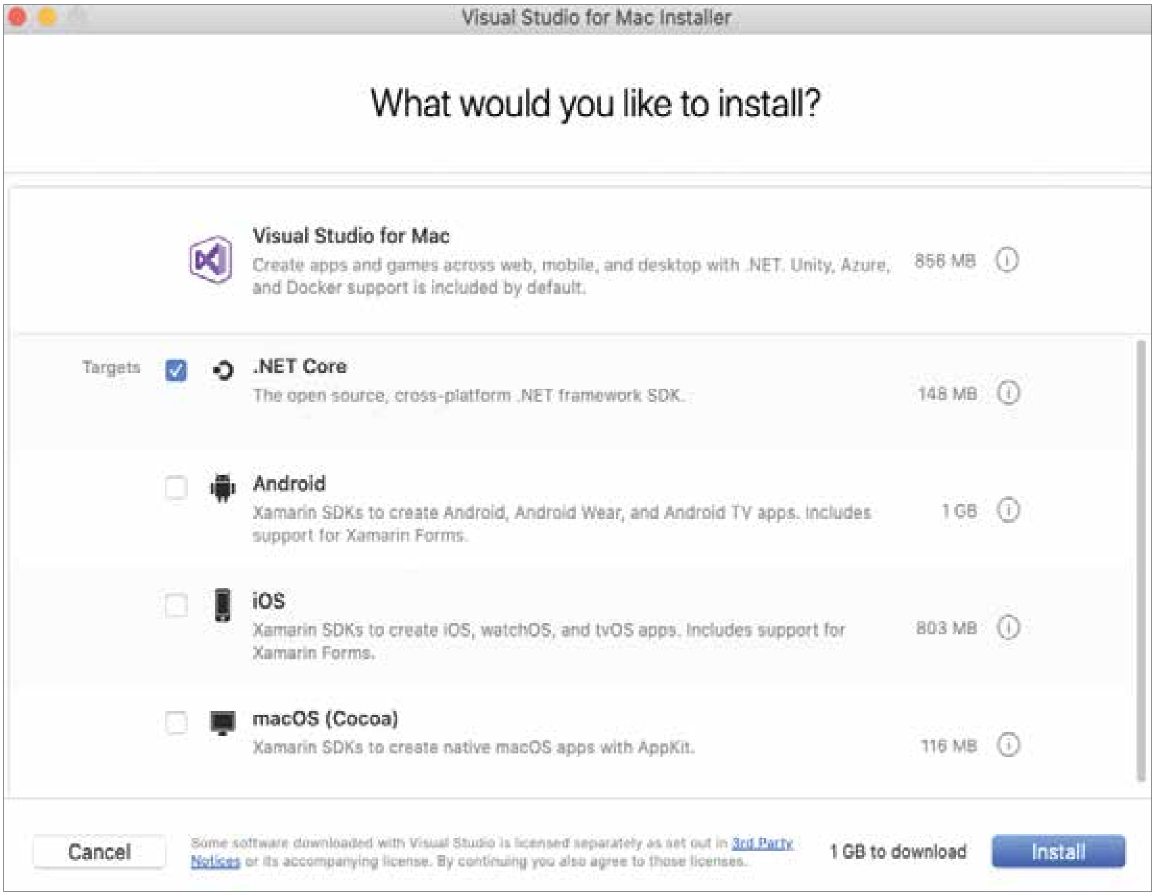
```
using System.Runtime.InteropServices;
```

Now add the following block of code to your C# program:

```
 var extractor_exe_path  = "";
if (RuntimeInformation
  .IsOSPlatform(OSPlatform.Windo ws))
  {
```

Listing 7: The error information

```
Traceback (most recent call last):
  File "run_cc.py", line 15, in <module>
subprocess.run([extractor_name,
  os.path.join(args.directory, file)])
  File "/Library/Developer/CommandLineTools/
  Library/Frameworks/Python3.framework/
  Versions/3.8/lib/python3.8/subprocess.py",
  line 489, in run with Popen(*popenargs, **kwargs)
   as process:
  File "/Library/Developer/CommandLineTools
  /Library/Frameworks/Python3.framework
  /Versions/3.8/lib/python3.8/subprocess.py",
  line 854, in __init__
  self._execute_child(args, executable,
  preexec_fn, close_fds,  File "/Library/Developer
  /CommandLineTools/Library
  /Frameworks/Python3.framework/Versions
  /3.8/lib/python3.8/subprocess.py",
  line 1702, in _execute_child
  raise child_exception_type(
  errno_num, err_msg, err_filename)
  FileNotFoundError:
  [Errno 2]
  No such file or directory: 'ccextractorwin'
```



Visual Studio for Mac Installer

What would you like to install?

Visual Studio for Mac
Create apps and games across web, mobile, and desktop with .NET. Unity, Azure, and Docker support is included by default. 856 MB

Targets
☑ .NET Core
The open source, cross-platform .NET framework SDK. 148 MB

☐ Android
Xamarin SDKs to create Android, Android Wear, and Android TV apps. Includes support for Xamarin Forms. 1 GB

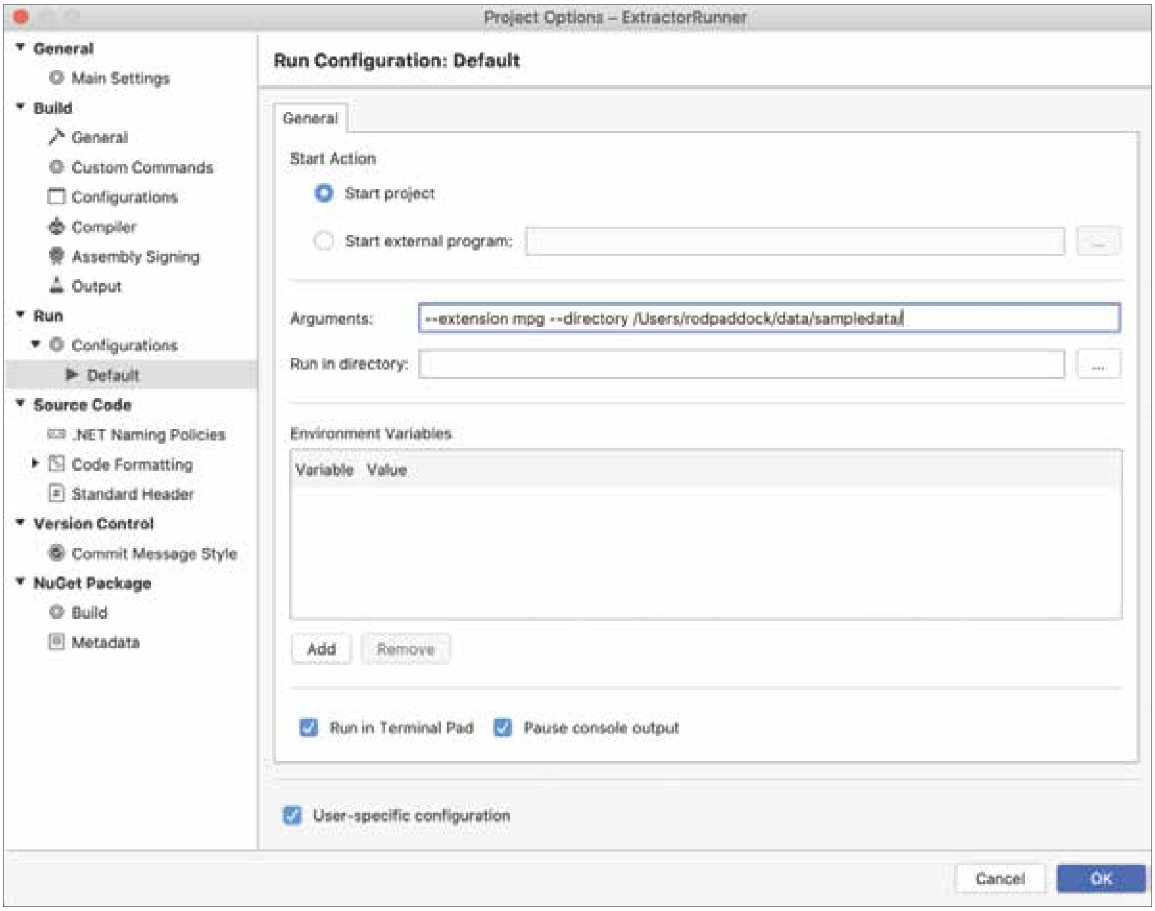
☐ iOS
Xamarin SDKs to create iOS, watchOS, and tvOS apps. Includes support for Xamarin Forms. 803 MB

☐ macOS (Cocoa)
Xamarin SDKs to create native macOS apps with AppKit. 116 MB

Some software downloaded with Visual Studio is licensed separately as set out in 3rd Party Notices or its accompanying license. By continuing you also agree to those licenses.

1 GB to download

Figure 4: Install the application with .NET Core option selected.



Start Action

● Start project

○ Start external program:

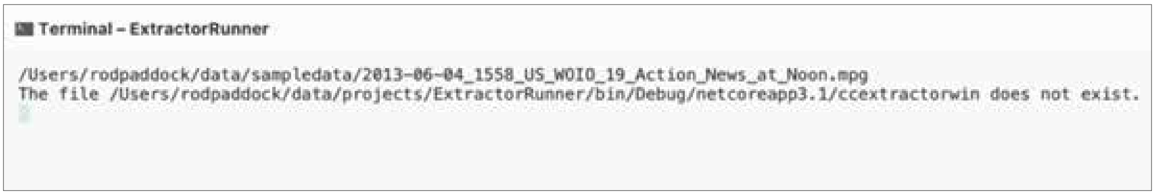
Arguments: --extension mpg --directory /Users/rodpaddock/data/sampledata/

Run in directory:

☑ Run in Terminal Pad ☑ Pause console output

☑ User-specific configuration

Figure 5: Set the command line parameters.


Terminal – ExtractorRunner

```
/Users/rodpaddock/data/sampledata/2013-06-04_1558_US_WOIO_19_Action_News_at_Noon.mpg
The file /Users/rodpaddock/data/projects/ExtractorRunner/bin/Debug/netcoreapp3.1/ccextractorwin does not exist.
```

Figure 6: The error shows in the console window.

```
  extractor_exe_path = “ccextractorwin”;
  }
else if (RuntimeInformation
  .IsOSPlatform(OSPlatform.OSX))
  {
  extractor_exe_path = “ccextractorwin”;
  }
else if (RuntimeInformation
  .IsOSPlatform(OSPlatform.Linux ))
  {
  extractor_exe_path = “ccextractor”;
  }
```

Run your code and you should see proper output in the runner window.

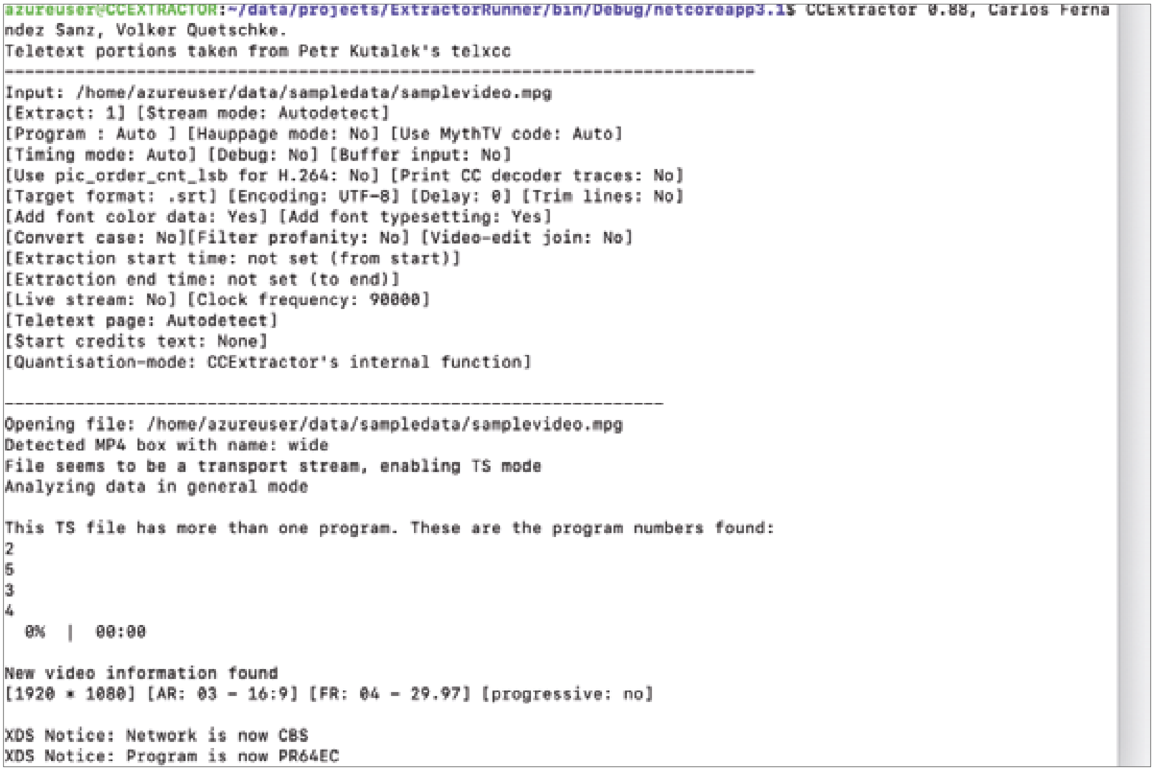
Now you have the same command line functionality for both the Python and C# versions of this program and can run the code on the Mac and Windows. Let’s take a look at the process of running thus under Ubuntu.

Running on Linux (Ubuntu)

Before modifying the runner programs, you need to install the CCExtractor application on your Linux server. Directions for installing the CCExtractor on Linux can be found here: https://github.com/CCExtractor/ccextractor/wiki/Installation.

Basically, you pull the code from GitHub, and run the typical process of building applications in the Linux world—i.e. MAKE the application. Luckily for me, the code “just compiled” using the instructions provided. Once built, I had to make one simple change to the script and was able to execute the runner application. The branch of code to determine the proper program to run looks like this:

```
if platform.system() == 'Windows' :
  extractor_name = 'ccextractorwin'
elif platform.system() == 'Darwin':
  extractor_name = 'ccextractor'
elif platform.system() == "Linux":
  extractor_name =
```



```
azureuser@CCEXTRACTOR:~/data/projects/ExtractorRunner/bin/Debug/netcoreapp3.1$ CCExtractor 0.88, Carlos Ferna
ndez Sanz, Volker Quetschke.
Teletext portions taken from Petr Kutalek's telxcc
--------------------------------------------------------------------------
Input: /home/azureuser/data/sampledata/samplevideo.mpg
[Extract: 1] [Stream mode: Autodetect]
[Program : Auto ] [Hauppage mode: No] [Use MythTV code: Auto]
[Timing mode: Auto] [Debug: No] [Buffer input: No]
[Use pic_order_cnt_lsb for H.264: No] [Print CC decoder traces: No]
[Target format: .srt] [Encoding: UTF-8] [Delay: 0] [Trim lines: No]
[Add font color data: Yes] [Add font typesetting: Yes]
[Convert case: No][Filter profanity: No] [Video-edit join: No]
[Extraction start time: not set (from start)]
[Extraction end time: not set (to end)]
[Live stream: No] [Clock frequency: 90000]
[Teletext page: Autodetect]
[Start credits text: None]
[Quantisation-mode: CCExtractor's internal function]

-----------------------------------------------------------------
Opening file: /home/azureuser/data/sampledata/samplevideo.mpg
Detected MP4 box with name: wide
File seems to be a transport stream, enabling TS mode
Analyzing data in general mode

This TS file has more than one program. These are the program numbers found:
2
5
3
4
  0%  |  00:00

New video information found
[1920 * 1080] [AR: 03 - 16:9] [FR: 04 - 29.97] [progressive: no]

XDS Notice: Network is now CBS
XDS Notice: Program is now PR64EC
```

Figure 7: Here's the code on Ubuntu.

```
'/home/azureuser
/data/projects/ccextractor/linux/ ccextractor'
```

Now I was able to run the code using the same command line options I used on the Mac.

```
python3 run_cc.py --extension mpg
  --directory /home/azureuser/data/sampledata/
```

Now that the Python code is up and running you can turn your sites onto running the C# code. To do this, you need to first install the .NET Core SDK on your Ubuntu instance. This is done by following the directions from this page: https://docs.microsoft.com/en-us/dotnet/core/install/linux-ubuntu.

If you're running a different flavor of Linux, you can find directions on this page: https://docs.microsoft.com/en-us/dotnet/core/install/linux.

Once you have the SDK installed, change into the folder where you cloned the GitHub repository and run the following command:

```
dotnet build
```

This builds an executable file and puts it in a sub-folder (off the root of your code) in this location **/bin/Debug/netcoreapp3.1**. There's one more step. Before you can run the code, you need to change your program.cs file to use the following executable selection code:

```
var extractor_exe_path = "";
if (RuntimeInformation
.IsOSPlatform(OSPlatform.Windo ws))
  {
  extractor_exe_path = "ccextractorwin";
  }
else if (RuntimeInformation
.IsOSPlatform(OSPlatform.OSX))
  {
  extractor_exe_path = "ccextractor";
  }
else if (RuntimeInformation
  .IsOSPlatform(OSPlatform.Linux ))
  {
    extractor_exe_path =
    "/home/azureuser/data
    /projects/ccextractor/linux/ ccextractor";
  }
```

Run the d**otnet build** command again, change into that folder, and run the following command:

```
./ExtractorRunner --extension mpg
--directory
 /home/azureuser/data/sampledata/
```

Figure 7 shows the runner application running on Ubuntu.

End Notes

This is how you create a totally cross-platform application in Python and C#. I was pleasantly surprised at how simple it was to build and run the C# code on Mac and Linux, which is a testament to the work that the Microsoft team has done over the last few years.

Rod Paddock

ONLINE QUICK ID 2107071

Building a VS Code Extension Using Vue.js

Visual Studio (VS) Code is one of the most preferred code editors that developers use in their everyday tasks. It's built with extendibility in mind. To a certain extent, most of the core functionalities of VS Code are built as extensions. You can check the VS Code extensions repository (https://github.com/microsoft/vscode/tree/main/extensions) to get an idea of what I'm talking about.

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

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

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

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

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

VS Code, under the hood, is an electron (https://www.electronjs.org/) cross-environment application that can run on UNIX, Mac OSX, and Windows operating systems. Because it's an electron application, you can extend it by writing JavaScript plug-ins. In fact, any language that can transpile to JavaScript can be used to build an extension. For instance, the VS Code docs website prompts the use of TypeScript (https://www.typescriptlang.org/) to write VS Code extensions. All the code examples (https://github.com/microsoft/vscode-extension-samples), provided by the VS Code team, are built using TypeScript.

VS Code supports a very extensive API that you can check and read on VS Code API (https://code.visualstudio.com/api).

VS Code allows you to extend almost any feature that it supports. You can build custom commands, create a new color theme, embed custom HTML inside a WebView, contribute to the activity bar by adding new views, make use of a Tree View to display hierarchical data on the sidebar, and many other extendibility options. The Extensions Capabilities Overview page (https://code.visualstudio.com/api/extension-capabilities/overview) details all the VS Code extension capabilities. In case you want to skip the overview and go directly to the details on how to build real-world extensions with capabilities, check the Extensions Guides page (https://code.visualstudio.com/api/extension-guides/overview).

Building extensions in VS Code is a huge topic that can be detailed into many books and countless articles. In this article, I will focus on:

- Creating VS Code Commands
- Using the Webview API to embed a Vue.js app inside Webview panels and views
- Adding a View Container to the Activity Bar

VS Code UI Architecture

Before I delve into building extensions, it's important to understand the parts and sections that make up the VS Code UI.

I'll borrow two diagrams from the VS Code website to help illustrate the concepts. **Figure 1** illustrates the major sections of the VS Code UI.

VS Code has the following main sections:

- **Activity Bar**: Every icon on the Activity Bar represents a View Container. In turn, this container hosts one or more views inside. In addition, you can extend the existing ones too. For example, you can add a new View into the Explorer View.
- **Sidebar**: A Sidebar is a container to host Views. For example, you can add a Tree View or Webview View to the Sidebar.
- **Editor**: The Editor hosts the different types of editors that VS Code uses. For instance, VS Code uses a text editor to allow you to read/write a file. Another kind of editor allows you to edit Workspace and User settings. You can also contribute your own editor using a Webview for instance.
- **Panel:** The Panel allows you to add View Containers with Views.
- **Status Bar**: The Status Bar hosts Status Bar Items that can use text and icons to display. You can also treat them as commands to trigger an action when clicking them.

Figure 2 illustrates what goes inside the major sections of the VS Code UI.

- The Activity Bar hosts View Containers, which, in turn, host Views.
- A View has a View Toolbar.
- The Sidebar has a Sidebar Toolbar.
- The Editor has an Editor Toolbar.
- The Panel hosts View Containers, which, in turn, host Views.
- A Panel has a Panel Toolbar.

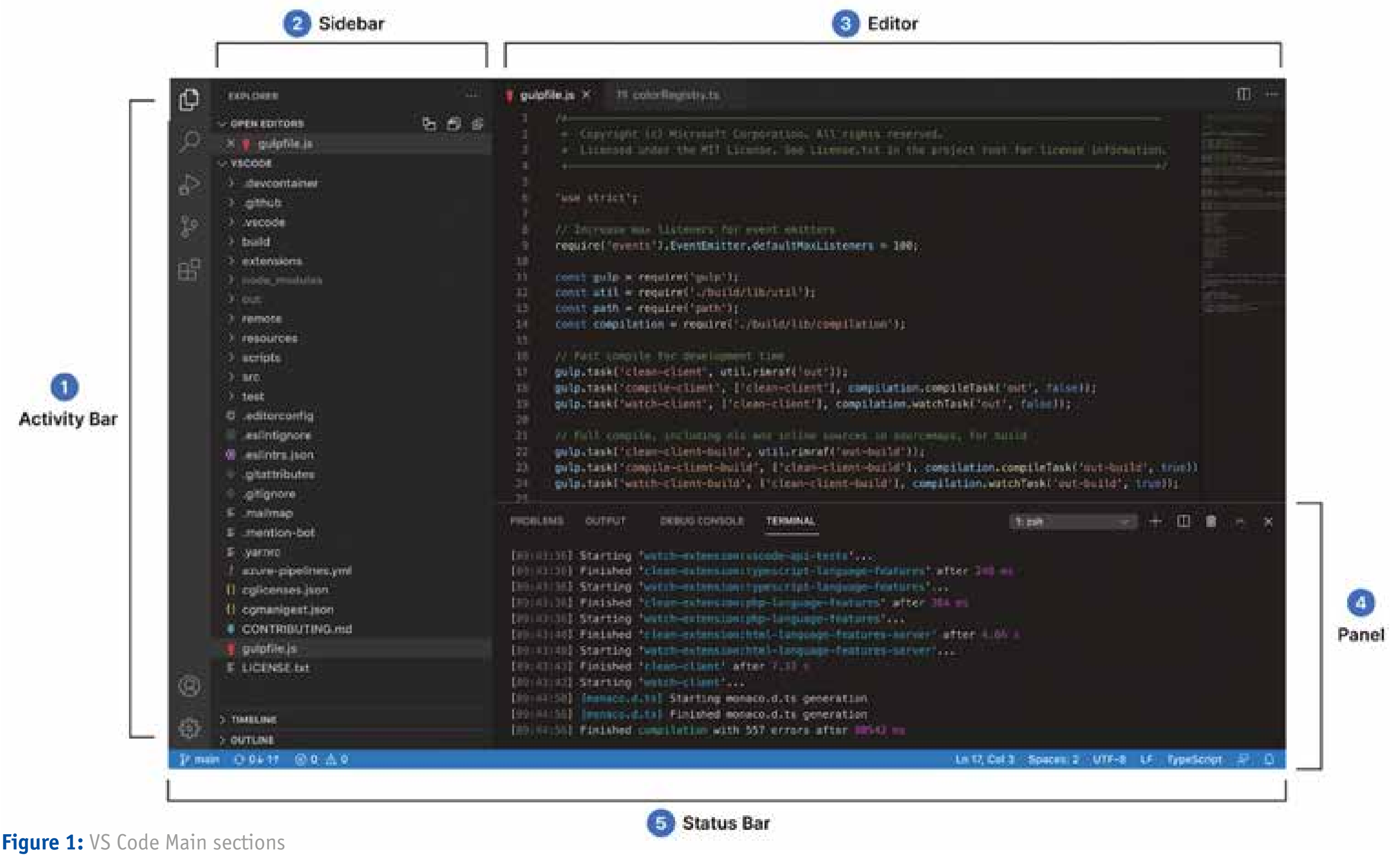
VS Code allows us to extend any of the major and minor sections using its API.

> VS Code API is rich enough to allow developers to extend almost every feature it offers.

Your First VS Code Extension

To start building your own custom VS Code extensions, make sure that you have Node.js (https://nodejs.org/en/) and Git (https://git-scm.com/) both installed on your computer. It goes without saying that you need to have VS Code (https://code.visualstudio.com/download) installed on your computer too.

I'll be using the Yeoman (https://yeoman.io/) CLI to generate a new VS Code extension project. Microsoft offers and supports the Yo Code (https://www.npmjs.com/package/generator-code) Yeoman generator to scaffold a complete VS Code extension in either TypeScript or JavaScript.



```
/*---------------------------------------------------------------------------------------------
 *  Copyright (c) Microsoft Corporation. All rights reserved.
 *  Licensed under the MIT License. See License.txt in the project root for license information.
 *--------------------------------------------------------------------------------------------*/

'use strict';

// Increase max listeners for event emitters
require('events').EventEmitter.defaultMaxListeners = 100;

const gulp = require('gulp');
const util = require('./build/lib/util');
const path = require('path');
const compilation = require('./build/lib/compilation');

// Fast compile for development time
gulp.task('clean-client', util.rimraf('out'));
gulp.task('compile-client', ['clean-client'], compilation.compileTask('out', false));
gulp.task('watch-client', ['clean-client'], compilation.watchTask('out', false));

// Full compile, including nls and inline sources in sourcemaps, for build
gulp.task('clean-client-build', util.rimraf('out-build'));
gulp.task('compile-client-build', ['clean-client-build'], compilation.compileTask('out-build', true))
gulp.task('watch-client-build', ['clean-client-build'], compilation.watchTask('out-build', true));
```

```
[09:43:36] Starting 'watch-extension:vscode-api-tests'...
[09:43:36] Finished 'clean-extension:typescript-language-features' after 248 ms
[09:43:36] Starting 'watch-extension:typescript-language-features'...
[09:43:36] Finished 'clean-extension:php-language-features' after 384 ms
[09:43:36] Starting 'watch-extension:php-language-features'...
[09:43:40] Finished 'clean-extension:html-language-features-server' after 4.06 s
[09:43:40] Starting 'watch-extension:html-language-features-server'...
[09:43:43] Finished 'clean-client' after 7.33 s
[09:43:43] Starting 'watch-client'...
[09:44:58] [monaco.d.ts] Starting monaco.d.ts generation
[09:44:58] [monaco.d.ts] Finished monaco.d.ts generation
[09:44:58] Finished compilation with 557 errors after 80542 ms
```

Figure 1: VS Code Main sections

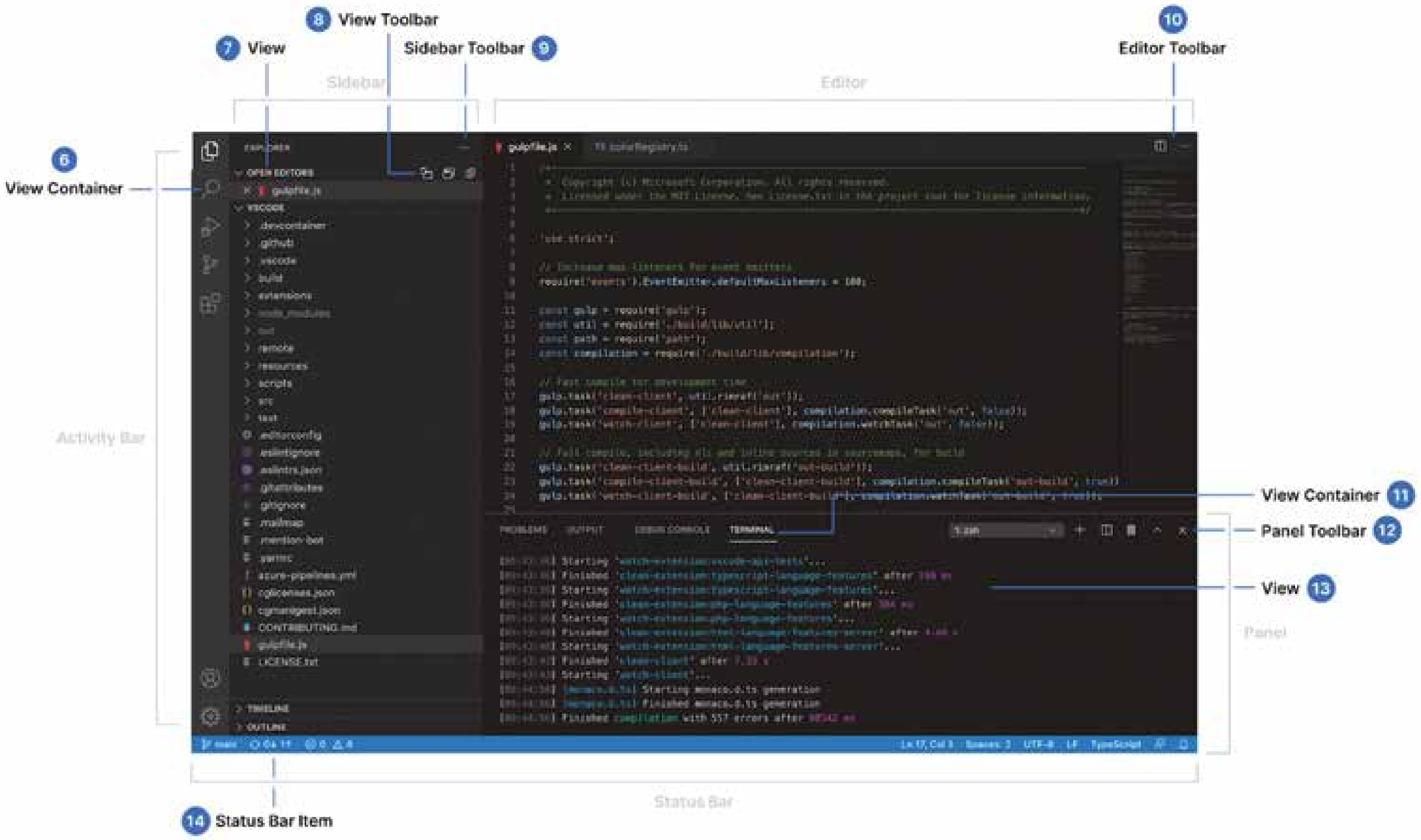


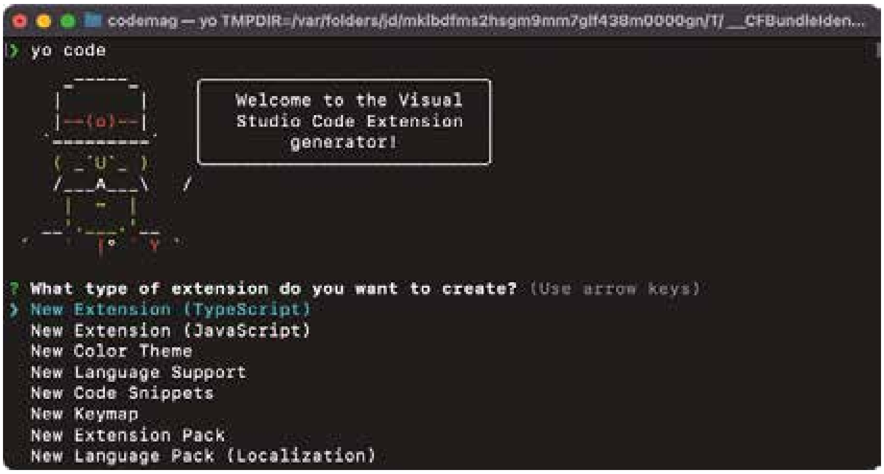
Figure 2: VS Code section details

Let's start!

Step 1

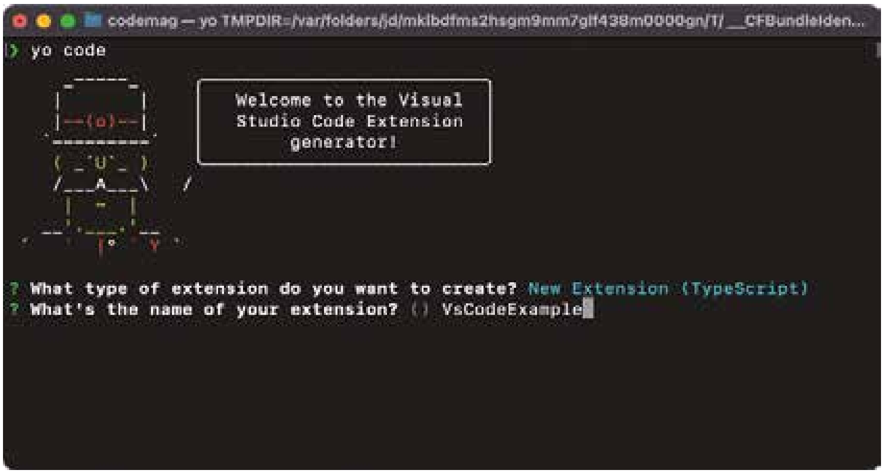
Install the Yeoman CLI and Yo Code generator by running the following command:

```
npm install -g yo generator-code
```


codemag — yo TMPDIR=/var/folders/jd/mklbdfms2hsgm9mm7glf438m0000gn/T/ __CFBundleIden...

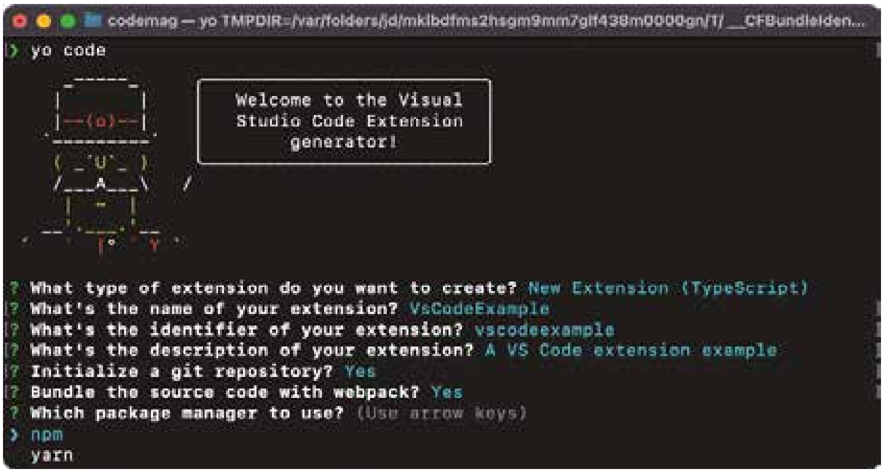
```
> yo code

     Welcome to the Visual
     Studio Code Extension
          generator!

? What type of extension do you want to create? (Use arrow keys)
> New Extension (TypeScript)
  New Extension (JavaScript)
  New Color Theme
  New Language Support
  New Code Snippets
  New Keymap
  New Extension Pack
  New Language Pack (Localization)
```

Figure 3: Start extension project scaffolding


codemag — yo TMPDIR=/var/folders/jd/mklbdfms2hsgm9mm7glf438m0000gn/T/ __CFBundleIden...

```
> yo code

     Welcome to the Visual
     Studio Code Extension
          generator!

? What type of extension do you want to create? New Extension (TypeScript)
? What's the name of your extension? () VsCodeExample
```

Figure 4: Naming the VS Code extension


codemag — yo TMPDIR=/var/folders/jd/mklbdfms2hsgm9mm7glf438m0000gn/T/ __CFBundleIden...

```
> yo code

     Welcome to the Visual
     Studio Code Extension
          generator!

? What type of extension do you want to create? New Extension (TypeScript)
? What's the name of your extension? VsCodeExample
? What's the identifier of your extension? vscodeexample
? What's the description of your extension? A VS Code extension example
? Initialize a git repository? Yes
? Bundle the source code with webpack? Yes
? Which package manager to use? (Use arrow keys)
> npm
  yarn
```

Figure 5: Finalize the code-generator scaffolding

Step 2

Run the following command to scaffold a TypeScript or JavaScript project ready for development.

```
yo code
```

During the process of creating the new VS Code extension project, the code-generator asks some questions. I'll go through them to create the app.

Figure 3 shows the starting point of the generator.

You can either pick TypeScript or JavaScript. Most of the examples you find online are written in TypeScript. It would be smart to go with TypeScript to make your life easier when writing and authoring your extension.

Next, you need to provide the name of your extension, as shown in **Figure 4**.

Now, you specify an identifier (ID) of your extension. You can leave the default or provide your own. I tend to use no spaces or dashes (-) to separate the identifier name.

Then, you can provide a description of your extension.

The next three questions are shown in **Figure 5**.

- Initialize a Git repository? **Yes**
- Use Webpack to bundle the extension? **Yes**
- Which package manager to use? **npm**

The generator takes all your answers and scaffolds your app. Once done, move inside the new extension folder and open VS Code by running this command:

```
cd vscodeexample && code .
```

Step 3

Let's quickly explore the extension project.

Figure 6 lists all the files that the Yo Code generated for you.

The /.vscode/ directory contains configuration files to help us test our extension easily.

The /dist/ directory contains the compiled version of the extension.

The /src/ directory contains the source code you write to build the extension.

> Microsoft offers the Yo Code Yeoman generator to help you scaffold a VS Code extension project quickly and easily.

The package.json file is the default NPM configuration file. You use this file to define your custom command, views, menus, and much more.

The vsc-extension-quickstart.md file contains an introduction to the extension project and documentation on how to get started building a VS Code extension.

Step 4

Open the package.json file and let's explore the important sections you need for building this extension.

```
“contributes”: {
    “commands”: [
        {
            “command”: “vscodeexample.helloWorld”,
            “title”: “Hello World”
        }
    ]
},
```

You define your custom commands inside the **contributes** section. You provide command and title, as a minimum, when you define a new command. The command should uniquely identify your command. By default, the command used is a concatenation of the extension identifier that you've specified at the time of scaffolding the extension together with an arbitrary string that represents the command you're providing. The new command automatically shows up now in the Command Palette.

VS Code defines a lot of built-in commands that you can even consume programmatically. For instance, you can execute the **workbench.action.newWindow** command to open a new VS Code instance.

Here's a complete list of Built-in Commands (https://code.visualstudio.com/api/references/commands) in VS Code.

The command does nothing for now. You still need to bind this command to a command handler that I'll define shortly. VS Code provides the registerCommand() function to do the association for you.

You should define an Activation Event that will activate the extension when the user triggers the command. It's the Activation Event that lets VS Code locate and bind a command to a command handler. Remember, extensions aren't always activated by default. For example, an extension might be activated when you open a file with a specific file extension. That's why it's needed to make sure the extension is activated before running any command.

The package.json file defines an activationEvents section:

```
“activationEvents”: [
     “onCommand:vscodeexample.helloWorld”
 ],
```

When the user invokes the command from the Command Palette or through a keybinding, the extension will be activated and registerCommand() function will bind the (vscodeexample.helloWorld) to the proper command handler.

Step 5

It's time to explore the extension source code and register the command together with a command handler. The extension source code lies inside the /src/extension.ts file. I've cleaned up this file as follows:

```
import * as vscode from ‘vscode’;

export function activate(
   context: vscode.ExtensionContext) {
      context.subscriptions.push(...);
}

export function deactivate() {}
```

VS Code calls the activate() function when it wants to activate the extension. Similarly, when it calls the deactivate() function, it wants to deactivate it. Remember, the extension is activated only when one of your declared Activation Events happens.

If you instantiate objects inside the command handler and want VS Code to release them for you later, push the new command registration into the context.subscriptions array. VS Code maintains this array and will do garbage collection on your behalf.

Let's register the Hello World command as follows:

```
context.subscriptions.push(
vscode.commands.registerCommand(
    ‘vscodeexample.helloWorld’,
    () => {
      vscode.window.showInformationMessage(‘…’);
    }
  )
);
```

The **vscode** object is the key to access the entire VS Code API. You register a command handler similarly to how you register DOM events in JavaScript. The code binds the same command identifier, the one that you previously declared inside the package.json file under the commands and activationEvents sections, to a command handler.

VS Code shows an information message when the user triggers the command.

Let's test the extension by clicking **F5**. VS Code opens a new instance loaded with the new extension. To trigger the command, open the Command Palette and start typing “hello”.

Figure 7 shows how VS Code filters the available commands to the one you are after.

Now click the command, and **Figure 8** shows how the information message appears on the bottom right side of the editor.

Congratulations! You've just completed your first VS Code extension!

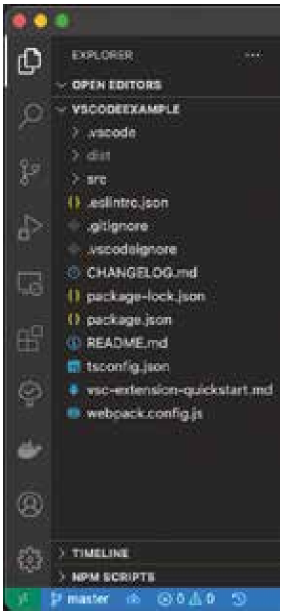


Figure 6: Project files

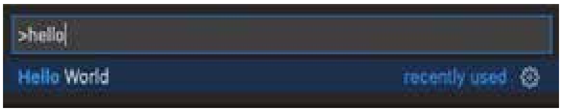
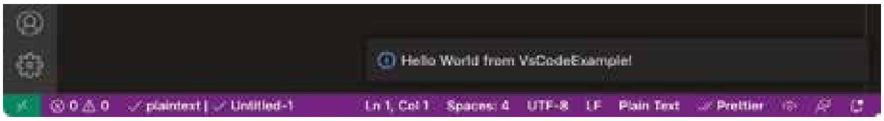


Figure 7: Command Palette filtered



Hello World from VsCodeExample!

Figure 8: Showing information message

Build a VS Code Extension with Vue.js Using Vue CLI

Let's use your newfound knowledge and build something more fun!

In this section, you'll use the Vue CLI to create a new Vue.js app and host it inside a Webview as a separate editor.

The Webview API allows you to create fully customizable views within the VS Code. I like to think of Webview as an iframe inside VS Code. It can render any HTML content inside this frame. It also supports two-way communication between the extension and the loaded HTML page. The view can post a message to the extension and vice-versa.

Webview API supports two types of views that I'm going to explore in this article:

- WebviewPanel is a wrapper around a Webview. It's used to display a Webview inside an editor in VS Code.
- WebviewView is a wrapper around a Webview. It's used to display a Webview inside the Sidebar.

In both types, the Webview hosts HTML content!

The Webview API documentation is rich and contains all the details you need to use it. Check it out here at Webview API (https://code.visualstudio.com/api/extension-guides/webview).

Let's start building our Vue.js sample application and hosting it inside an editor in VS Code.

> Webview allows you to enrich your VS Code extension by embedding HTML content together with JavaScript and CSS resource files.

Step 1

Generate a new VS Code extension project using Yeoman, as you did above.

Step 2

Add a new command to open the Vue.js app. Locate the package.json file and add the following:

```
“contributes”: {
   “commands”: [
      {
         “command”: “vscodevuecli:openVueApp”,
         “title”: “Open Vue App”
      }
    ]
},
```

The command has an identifier of **vscodevuecli:openVueApp**.

Then you declare an Activation Event as follows:

```
“activationEvents”: [
     “onCommand:vscodevuecli:openVueApp”
],
```

Step 3

Switch to the extension.js file and inside the activate() function register the command handler.

```
context.subscriptions.push(
   vscode.commands.registerCommand(
    ‘vscodevuecli:openVueApp’, () => {
      WebAppPanel.createOrShow(context.extensionUri);
   })
);
```

Inside the command handler, you're instantiating a new instance of **WebAppPanel** class. It's just a wrapper around a WebviewPanel.

Step 4

In this step, you'll generate a new Vue.js app using the Vue CLI. Follow this guide (https://cli.vuejs.org/guide/creating-a-project.html#vue-create) to scaffold a new Vue.js app inside the /web/ directory at the root of the extension project.

Make sure to place any image you use inside the /web/img/ directory. Later on, you'll copy this directory to the dist directory when you compile the app.

Usually, the HTML page, hosting the Vue.js app, requests images to render from the local file system on the server. However, when the Webview loads the app, it can't just request and access the local file system. For security reasons, the Webview should be limited to a few directories inside the project itself.

Also, VS Code uses special URIs to load any resource inside the Webview including JavaScript, CSS, and image files. Therefore, you need a way to base all the images, so you use the URI that VS Code uses to access the local resources. The extension, as you'll see in **Step 5**, injects the VS Code base URI, into the body of the HTML of the Webview, so that the Vue.js app can use it to base its images.

Therefore, to make use of the injected base URI, you'll add a Vue.js mixin that reads the value of the base URI from the HTML DOM and makes it available to the Vue.js app.

Note that if you want to run the Vue.js app outside the Webview, you need to place the following inside the /web/public/index.html file:

```
<body>
   <input hidden data-uri="">
   ...
</body>
```

Inside the /web/src/mixins/ExtractBaseUri.js file, define a new Vue.js mixin.

It makes available the baseUri data option to any Vue.js component:

```
data() {
   return {
      baseUri: ‘’,
   };
},
```

It then uses the Vue.js mounted() lifecycle hook to extract the value:

```
mounted() {
  const dataUri =
        document.querySelector(‘input[data-uri]’);
  if (!dataUri) return;

  this.baseUri = dataUri.getAttribute(‘data-uri’);
},
```

If it finds an input field with a data attribute named data-uri, it reads the value and assigns it to the baseUri property.

The next step is to provide the mixin inside the /web/src/main.js file:

```
Vue.mixin(ExtractBaseUri);
```

Switch to the App.vue component and replace the image element with the following:

```
<img alt=”Vue logo” :src=”`${baseUri}/img/logo.png`”>
```

Now that the app is ready to run both locally and inside the Webview, let's customize the compilation process via the Vue.js configuration file.

Create a new /web/vue.config.js file. **Listing 1** shows the entire source code for this file. Basically, you're doing the following:

- Removing the hashes from the compiled file names. The compiled JavaScript file will look like app.js only without any hashes in the file name.
- Sets the output directory to be /dist-web/. The Vue CLI uses this property to decide where to place the compiled app files.
- Copy to the destination directory the /web/img/ directory and all of its content.

Next, let's fix the NPM scripts so that you can compile both the extension files and the Vue.js app at the same time using a single script.

First, start by installing the **Concurrently** NPM package by running the following command:

```
npm i --save-dev concurrently
```

Then, locate the package.json file and replace the watch script with this:

```
“watch”: “concurrently \”npm --prefix web run dev\”
                       \”webpack --watch\””,
```

The watch script now compiles both the Vue.js app and the extension files every time you change any files in both folders.

Run the following command to compile both apps and generate the /dist-web/ directory:

```
npm run watch
```

That's it for now! The Vue.js app is ready for hosting inside a Webview.

Step 5

Add a new TypeScript file inside the /src/ directory and name it **WebAppPanel.ts**. **Listing 2** has the full source code for this file. Let's dissect it and explain the most relevant parts of it.

Listing 1: vue.config.js

```
const path = require('path');

module.exports = {
  filenameHashing: false,
  outputDir: path.resolve(__dirname, "../dist-web"),
  chainWebpack: config => {
    config.plugin('copy')
      .tap(([pathConfigs]) => {
        const to = pathConfigs[0].to
        // so the original `/public` folder keeps priority
        pathConfigs[0].force = true

        // add other locations.
        pathConfigs.unshift({
          from: 'img',
          to: `${to}/img`,
        })

        return [pathConfigs]
      })
    },
}
```

You define the WebAppPanel class as a singleton to make sure there's always a single instance of it. This is done by adding the following:

```
public static currentPanel: WebAppPanel | undefined;
```

It wraps an instance of the WebviewPanel and tracks it by defining the following:

```
private readonly _panel: vscode.WebviewPanel;
```

The createOrShow**()** function is the core of WebAppPanel class. It checks to see whether the currentPanel is already instantiated, and it shows the WebviewPanel right away.

```
if (WebAppPanel.currentPanel) {
   WebAppPanel.currentPanel._panel.reveal(column);
   return;
}
```

Otherwise, it instantiates a new WebviewPanel using the createWebviewPanel() function as follows:

```
const panel = vscode.window.createWebviewPanel(
    WebAppPanel.viewType,
    'Web App Panel',
    column || vscode.ViewColumn.One,
    getWebviewOptions(extensionUri),
);
```

This function accepts the following parameters:

- **viewType**: A unique identifier specifying the view type of the WebviewPanel
- **title:** The title of the WebviewPanel
- **showOptions:** Where to show the Webview in the editor
- **options**: Settings for the new Panel

The options are prepared inside the getWebviewOptions() function.

```
function getWebviewOptions(
   extensionUri: vscode.Uri
): vscode.WebviewOptions {
```

Listing 2: WebAppPanel.ts

```
import * as vscode from “vscode”;
import { getNonce } from “./getNonce”;

export class WebAppPanel {

  public static currentPanel: WebAppPanel | undefined;

  public static readonly viewType = “vscodevuecli:panel”;

  private readonly _panel: vscode.WebviewPanel;
  private readonly _extensionUri: vscode.Uri;
  private _disposables: vscode.Disposable[] = [];

  public static createOrShow(extensionUri: vscode.Uri) {
      const column = vscode.window.activeTextEditor
        ? vscode.window.activeTextEditor.viewColumn
        : undefined;

      // If we already have a panel, show it.
      if (WebAppPanel.currentPanel) {
        WebAppPanel.currentPanel._panel.reveal(column);
        return;
      }

      // Otherwise, create a new panel.
      const panel = vscode.window.createWebviewPanel(
        WebAppPanel.viewType,
        ‘Web App Panel’,
        column || vscode.ViewColumn.One,
        getWebviewOptions(extensionUri),
      );

      WebAppPanel.currentPanel =
         new WebAppPanel(panel, extensionUri);
    }

  public static kill() {
    WebAppPanel.currentPanel?.dispose();
    WebAppPanel.currentPanel = undefined;
  }

  public static revive(panel: vscode.WebviewPanel,
    extensionUri: vscode.Uri) {
    WebAppPanel.currentPanel = new WebAppPanel(panel, extensionUri);
  }

  private constructor(panel: vscode.WebviewPanel,
    extensionUri: vscode.Uri) {
    this._panel = panel;
    this._extensionUri = extensionUri;

    // Set the webview’s initial html content
    this._update();

    this._panel.onDidDispose(() => this.dispose(), null,
         this._disposables);

    // Update the content based on view changes
    this._panel.onDidChangeViewState(
      e => {
        if (this._panel.visible) {
          this._update();
        }
      },
      null,
      this._disposables
    );

    // Handle messages from the webview
    this._panel.webview.onDidReceiveMessage(
      message => {
        switch (message.command) {
          case ‘alert’:
            vscode.window.showErrorMessage(message.text);
            return;
        }
      },
      null,
      this._disposables
    );
  }

  public dispose() {
    WebAppPanel.currentPanel = undefined;

    // Clean up our resources
    this._panel.dispose();

    while (this._disposables.length) {
      const x = this._disposables.pop();
      if (x) {
        x.dispose();
      }
    }
  }

  private async _update() {
      const webview = this._panel.webview;
      this._panel.webview.html = this._getHtmlForWebview(webview);
  }

  private _getHtmlForWebview(webview: vscode.Webview) {
    const styleResetUri = webview.asWebviewUri(
      vscode.Uri.joinPath(this._extensionUri, “media”, “reset.css”)
    );

    const styleVSCodeUri = webview.asWebviewUri(
      vscode.Uri.joinPath(this._extensionUri, “media”, “vscode.css”)
    );

    const scriptUri = webview.asWebviewUri(
      vscode.Uri.joinPath(this._extensionUri, “dist-web”, “js/app.js”)
    );

    const scriptVendorUri = webview.asWebviewUri(
      vscode.Uri.joinPath(this._extensionUri, “dist-web”,
         “js/chunk-vendors.js”)
    );

    const nonce = getNonce();
    const baseUri =
       webview.asWebviewUri(vscode.Uri.joinPath(
         this._extensionUri,
         ‘dist-web’)
       ).toString().replace(‘%22’, ‘’);

    return `
      <!DOCTYPE html>
      <html lang=”en”>
      <head>
        <meta charset=”utf-8” />
        <meta name=”viewport”
              content=”width=device-width, initial-scale=1” />
        <link href=”${styleResetUri}” rel=”stylesheet”>
        <link href=”${styleVSCodeUri}” rel=”stylesheet”>
        <title>Web App Panel</title>
      </head>
      <body>
      <input hidden data-uri=”${baseUri}”>
          <div id=”app”></div>
          <script type=”text/javascript”
            src=”${scriptVendorUri}” nonce=”${nonce}”></script>
          <script type=”text/javascript”
            src=”${scriptUri}” nonce=”${nonce}”></script>
      </body>
      </html>
    `;
  }
}

function getWebviewOptions(extensionUri: vscode.Uri): vscode.WebviewOptions {
    return {
        // Enable javascript in the webview
        enableScripts: true,

        localResourceRoots: [
          vscode.Uri.joinPath(extensionUri, ‘media’),
          vscode.Uri.joinPath(extensionUri, ‘dist-web’),
        ]
    };
}
```

```
    return {
      enableScripts: true,
      localResourceRoots: [
        vscode.Uri.joinPath(extensionUri, 'media'),
        vscode.Uri.joinPath(extensionUri, 'dist-web'),
      ]
    };
}
```

It returns an object that has two properties:

- **enableScripts**: Controls whether scripts are enabled in the Webview content or not
- **localResourceRoots:** Specifies the root paths from which the Webview can load local resources using URIs (Universal Resource Identifier representing either a file on disk or any other resource). This guarantees that the extension cannot access files outside the paths you specify.

> WebviewPanel wraps a Webview to render inside a VS code editor.

The createOrShow() function ends by setting the value of the currentPanel to a new instance of the WebAppPanel by calling its private constructor.

The most important section of the constructor is setting the HTML content of the Webview as follows:

```
this._panel.webview.html =
     this._getHtmlForWebview(webview);
```

The _getHtmlForWebview() function prepares and returns the HTML content.

There are two CSS files that you'll embed in almost every Webview you create. The **reset.css** file resets some CSS properties inside the Webview. Although the **vscode.css** file contains the default theme colors and CSS properties of the VS Code. This is essential to give your Webview the same look and feel as any other editor in VS Code.

```
const styleResetUri = webview.asWebviewUri(
   vscode.Uri.joinPath(
     this._extensionUri, "media", "reset.css"
   )
);

const styleVSCodeUri = webview.asWebviewUri(
   vscode.Uri.joinPath(
     this._extensionUri, "media", "vscode.css"
   )
);
```

The _extensionUri property represents the URI of the directory containing the current extension. The Webview asWebviewUri() function converts a URI for the local file system to one that can be used inside Webviews. They cannot directly load resources from the Workspace or local file system using **file:** URIs. The asWebviewUri() function takes a local file: URI and converts it into a URI that can be used inside a Webview to load the same resource.

The function then prepares the URIs for the other resources including the js/app.js and js/chunk-vendors.js files that were compiled by the Vue CLI back in **Step 5**.

Remember from **Step 4**, the Vue CLI copies all images inside the /dist-web/img/ directory. All image paths inside the Vue.js app use a base URI that points to either a VS Code URI when running inside the Webview or a file: URI when running in a standalone mode.

At this stage, you need to generate a VS Code base URI and inject it into the hidden input field that the Vue.js loads and reads via the Vue.js mixin.

The WebAppPanel generates the VS Code base URI of the extension using the following code:

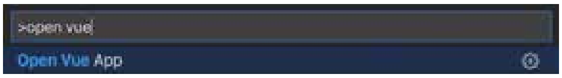
```
const baseUri =
   webview.asWebviewUri(
     vscode.Uri.joinPath(
       this._extensionUri, 'dist-web'
     )
   ).toString().replace('%22', '');
```

It communicates this URI to the Vue.js app by setting the data-uri data attribute value on a hidden input field inside the HTML page that's also loading the Vue.js app.

Finally, the function embeds all the CSS and JavaScript URIs inside the HTML page content and returns it.

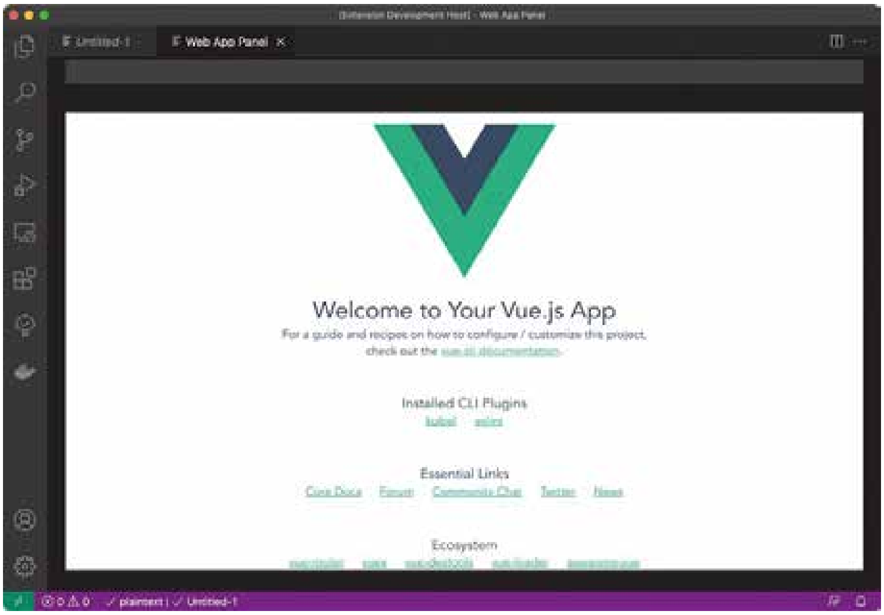
That's it!

Let's run the extension by clicking the **F5** key, and start typing "Open Vue App" inside the Command Palette of the VS Code instance that just opened, as shown in **Figure 9**.



```
>open vue
```

Figure 9: Open Vue App command



For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.

Figure 10: Vue app loading inside VS Code extension

Click the command to load the Vue.js app inside a Webview in a new editor window, as shown in **Figure 10**.

That's all you need to have a Vue.js app generated by Vue CLI load inside a VS Code extension.

Build a VS Code Extension with Vue.js using Rollup.js

In this section, I'll expand on what you've built so far and introduce a new scenario where the Vue CLI might not be the right tool for the job.

As you know, the Vue CLI compiles the entire Vue.js app into a single app.js file. Let's put aside the chunking feature offered by the CLI for now.

However, when building a VS Code extension, there are times when you need to load one HTML page inside a WebviewPanel in an editor. At the same time, you might need to load another HTML page inside a WebviewView in the Sidebar. Of course, you can use plain HTML and JavaScript to build your HTML, but because you want to use Vue.js to build your HTML pages, the Vue CLI is not an option in this case.

You need to create a Vue.js app that contains multiple small and independent Vue.js components that are compiled separately into separate JavaScript files and not just merged into a single app.js file.

I came up with a solution that involves creating micro Vue.js apps using a minimum of two files. A JavaScript file and one or more Vue.js components (a root component with many child components). The JavaScript file imports the Vue.js framework and mounts the corresponding Vue.js root component into the DOM inside the HTML page.

For this solution, I've decided to use Rollup.js (https://rollupjs.org/) to compile the files.

Let's explore this solution together by building a new VS Code extension that does two things:

- Uses a WebviewPanel to host a Vue.js app (or root component) into a new editor
- Uses a WebviewView to host a Vue.js app (or root component) into the Sidebar

Listing 3: Add a View Container

```
"viewsContainers": {
    "activitybar": [
        {
            "id": "vscodevuerollup-sidebar-view",
            "title": "Vue App",
            "icon": "$(remote-explorer)"
        }
    ]
  },
  "views": {
      "vscodevuerollup-sidebar-view": [
        {
            "type": "webview",
            "id": "vscodevuerollup:sidebar",
            "name": "vue with rollup",
            "icon": "$(remote-explorer)",
            "contextualTitle": "vue app"
        }
    ]
  },
```

Step 1

Generate a new VS Code extension project using Yeoman, as you did before.

Step 2

Add a new command to open the Vue.js app. Locate the package.json file and add the following:

```
"contributes": {
   "commands": [
      {
         "command": "vscodevuerollup:openVueApp",
         "title": "Open Vue App",
         "category": "Vue Rollup"
      }
   ]
},
```

The command has an identifier of **vscodevuerollup:openVueApp**.

Then you declare an Activation Event:

```
"activationEvents": [
     "onCommand:vscodevuerollup:openVueApp"
],
```

In addition, define a new View Container to load inside the Activity Bar. **Listing 3** shows the sections that you need to add inside the package.json file.

The Activity Bar entry has an ID of **vscodevuerollup-sidebar-view**. This ID matches the ID of the collection of Views that will be hosted inside this View Container and that's defined inside the views section.

```
"views": {
       "vscodevuerollup-sidebar-view": [...]
}
```

The (vscodevuerollup-sidebar-view) entry represents a collection of Views. Every View has an ID.

```
{
   "type": "webview",
   "id": "vscodevuerollup:sidebar",
   "name": "vue with rollup",
   "icon": "$(remote-explorer)",
   "contextualTitle": "vue app"
}
```

Make note of this ID **vscodevuerollup:sidebar**, scroll up to the activatinEvents section, and add the following entry:

```
onView:vscodevuerollup:sidebar
```

When using the onView declaration, VS Code activates the extension when the View, with the specified ID, is expanded on the Sidebar.

Step 3

Switch to the extension.js file and inside the activate() function register the command handlers.

First, register the **vscodevuerollup:openVueApp** command:

```
context.subscriptions.push(
   vscode.commands.registerCommand(
     ‘vscodevuerollup:openVueApp’, async (args) => {
       WebAppPanel.createOrShow(context.extensionUri);
     }
   )
);
```

Then register the **vscodevuerollup:sendMessage** command:

```
const sidebarProvider =
     new SidebarProvider(context.extensionUri);

context.subscriptions.push(
   vscode.window.registerWebviewViewProvider(
     SidebarProvider.viewType,
     sidebarProvider
   )
);
```

> WebviewViewProvider wraps a WebviewView, which, in turn, wraps a Webview. The WebviewView renders inside the Sidebar in VS Code.

You're instantiating a new instance of the SidebarProvider class and using the vscode.window.registerWebviewViewProvider() function to register this provider.

Here, you're dealing with the second type of Webviews that I've mentioned earlier, the WebviewView. To load a Webview into the Sidebar, you need to create a class that implements the **WebviewViewProvider** interface. It's just a wrapper around a WebviewView.

Step 4

In this step, you'll create a custom Vue.js app. Start by creating the /web/ directory at the root folder of the extension.

Inside this directory, create three different sub-directories:

- **pages:** This directory holds all the Vue.js pages.
- **components:** This holds all the Vue.js Single File Components (SFC).
- **img:** This holds all the images you use in your Vue.js components.

Let's add the first Vue.js page by creating the /web/pages/App.js file and pasting this code inside it:

```
import Vue from “vue”;
import App from “@/components/App.vue”;

new Vue({
  render: h => h(App)
}).$mount(“#app”);
```

There's no magic here! It's the same code that the Vue CLI uses inside the main.js file to load and mount the Vue.js app on the HTML DOM. However, in this case, I'm just mounting a single Vue.js component. Think of this component as a root Component that might use other Vue.js components in a tree hierarchy.

Listing 4: Sidebar.vue component

```
<template>
  <div>
    <p>Message received from extension</p>
    <span>{{ message }}</span>

    <p>Send message to extension</p>
    <input type=”text” v-model=”text”>
    <button @click=”sendMessage”>Send</button>

    <p>Open Vue App</p>
    <button @click=”openApp”>Open</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: ‘’,
      text: ‘’,
    };
  },
  mounted() {
    window.addEventListener(‘message’, this.receiveMessage);
  },
  beforeDestroy() {
    window.removeEventListener(‘message’, this.receiveMessage);
  },
  methods: {
     openApp() {
      vscode.postMessage({
        type: ‘openApp’,
      });
      this.text = ‘’;
    },
    sendMessage() {
      vscode.postMessage({
        type: ‘message’,
        value: this.text,
      });
      this.text = ‘’;
    },
    receiveMessage(event) {
      if (!event) return;

      const envelope = event.data;
      switch (envelope.command) {
        case ‘message’: {
          this.message = envelope.data;
          break;
        }
      };
    },
  },
}
</script>

<style scoped>
p {
  margin: 10px 0;
  padding: 5px 0;
  font-size: 1.2rem;
}
span {
  display: inline-block;
  margin-top: 5px;
  font-size: 1rem;
  color: orange;
}
hr {
  display: inline-block;
  width: 100%;
  margin: 10px 0;
}
</style>
```

Note that I’ve borrowed the same App.vue file from the Vue CLI files you created previously.

Let’s add another page by creating the /web/pages/Sidebar.js file and pasting this code inside it:

```
import Vue from “vue”;
import Sidebar from “@/components/Sidebar.vue”;

new Vue({
  render: h => h(Sidebar)
}).$mount(“#app”);
```

This page loads and mounts the Sidebar.vue component.

Listing 4 shows the complete content of the Sidebar.vue component. It defines the following UI sections:

- Display the messages received from the extension.
- Allow the user to send a message to the extension from within the Vue.js app.
- Execute a command on the extension to load the App.js page in a Webview inside an editor.

Navigate to the extension root directory and add a new rollup.config.js file.

Listing 5 shows the complete content of this file.

The most important section of this file:

```
export default fs
  .readdirSync(
      path.join(__dirname, “web”, “pages”)
   )
  .map((input) => {
    const name =
          input.split(“.”)[0].toLowerCase();

    return {
      input: `web/pages/${input}`,
      output: {
      file: `dist-web/${name}.js`,
      format: ‘iife’,
      name: ‘app’,
},
…
```

The code snippet iterates over all the *.js pages inside the /web/pages/ directory and compiles each page separately into a new JavaScript file inside the /dist-web/ directory.

Let’s install the **Concurrently** NPM package by running the following command:

```
npm i --save-dev concurrently
```

Then, locate the package.json file and replace the watch script with this:

```
“watch”: “concurrently \”rollup -c -w\”
                       \”webpack --watch\””,
```

The watch script now compiles both the Vue.js pages and the extension files every time you change any file in both folders.

Run this command to compile both apps and generate the /dist-web/ directory:

```
npm run watch
```

Listing 5: rollup.config.js

```
import path from "path";
import fs from "fs";

import alias from '@rollup/plugin-alias';
import commonjs from 'rollup-plugin-commonjs';
import esbuild from 'rollup-plugin-esbuild';
import filesize from 'rollup-plugin-filesize';
import image from '@rollup/plugin-image';
import json from '@rollup/plugin-json';
import postcss from 'rollup-plugin-postcss';
import postcssImport from 'postcss-import';
import replace from '@rollup/plugin-replace';
import resolve from '@rollup/plugin-node-resolve';
import requireContext from 'rollup-plugin-require-context';
import { terser } from 'rollup-plugin-terser';
import vue from 'rollup-plugin-vue';

const production = !process.env.ROLLUP_WATCH;

const postCssPlugins = [
  postcssImport(),
];

export default fs
  .readdirSync(path.join(__dirname, "web", "pages"))
  .map((input) => {
    const name = input.split(".")[0].toLowerCase();
    return {
      input: `web/pages/${input}`,
      output: {
        file: `dist-web/${name}.js`,
        format: 'iife',
        name: 'app',
        sourcemap: false,
      },
      plugins: [
        commonjs(),
        json(),
        alias({
          entries: [{ find: '@',
            replacement: __dirname + '/web/' }],
        }),
        image(),
        postcss({ extract: `${name}.css`,
            plugins: postCssPlugins }),
        requireContext(),
        resolve({
          jsnext: true,
          main: true,
          browser: true,
          dedupe: ["vue"],
        }),
        vue({
          css: false
        }),
        replace({
          'process.env.NODE_ENV': production ?
                 '"production"' : '"development"',
          preventAssignment: true,
        }),
        esbuild({
          minify: production,
          target: 'es2015',
        }),
        production && terser(),
        production && filesize(),
      ],
      watch: {
        clearScreen: false,
        exclude: ['node_modules/**'],
      },
    };
  });
```

Listing 6: WebAppPanel.ts loading single Vue.js Root Component

```
import * as vscode from “vscode”;
import { getNonce } from “./getNonce”;

export class WebAppPanel {
    public static currentPanel: WebAppPanel | undefined;

    public static readonly viewType = “vscodevuerollup:panel”;

    private readonly _panel: vscode.WebviewPanel;
    private readonly _extensionUri: vscode.Uri;
    private _disposables: vscode.Disposable[] = [];

    public static createOrShow(extensionUri: vscode.Uri) {
      const column = vscode.window.activeTextEditor
        ? vscode.window.activeTextEditor.viewColumn
        : undefined;

      // If we already have a panel, show it.
      if (WebAppPanel.currentPanel) {
        WebAppPanel.currentPanel._panel.reveal(column);
        return;
      }

      // Otherwise, create a new panel.
      const panel = vscode.window.createWebviewPanel(
        WebAppPanel.viewType,
        ‘Web App Panel’,
        column || vscode.ViewColumn.One,
        getWebviewOptions(extensionUri),
      );

      WebAppPanel.currentPanel =
           new WebAppPanel(panel, extensionUri);
    }

  public static kill() {
    WebAppPanel.currentPanel?.dispose();
    WebAppPanel.currentPanel = undefined;
  }

  public static revive(panel: vscode.WebviewPanel,
      extensionUri: vscode.Uri) {
    WebAppPanel.currentPanel =
      new WebAppPanel(panel, extensionUri);
  }

  private constructor(panel: vscode.WebviewPanel,
      extensionUri: vscode.Uri) {
    this._panel = panel;
    this._extensionUri = extensionUri;

    // Set the webview’s initial html content
    this._update();

    this._panel.onDidDispose(() => this.dispose(),
       null, this._disposables);

    // Update the content based on view changes
    this._panel.onDidChangeViewState(
      e => {
        if (this._panel.visible) {
          this._update();
        }
      },
      null,
      this._disposables
    );

    // Handle messages from the webview
    this._panel.webview.onDidReceiveMessage(
      message => {
        switch (message.command) {
          case ‘alert’:
            vscode.window.showErrorMessage(message.text);
            return;
        }
      },
      null,
      this._disposables
```

```
    );
  }

  public dispose() {
    WebAppPanel.currentPanel = undefined;

    // Clean up our resources
    this._panel.dispose();

    while (this._disposables.length) {
#      const x = this._disposables.pop();
      if (x) {
        x.dispose();
      }
    }
  }

  private async _update() {
      const webview = this._panel.webview;
      this._panel.webview.html = this._getHtmlForWebview(webview);
  }

  private _getHtmlForWebview(webview: vscode.Webview) {
    const styleResetUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
       this._extensionUri, “media”, “reset.css”)
    );

    const styleVSCodeUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
       this._extensionUri, “media”, “vscode.css”)
    );

    const scriptUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
       this._extensionUri, “dist-web”, “app.js”)
    );

    const styleMainUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
       this._extensionUri, “dist-web”, “app.css”)
    );

    const nonce = getNonce();

    return `
      <!DOCTYPE html>
      <html lang=”en”>
      <head>
        <meta charset=”utf-8” />
        <meta name=”viewport”
              content=”width=device-width, initial-scale=1” />
        <link href=”${styleResetUri}” rel=”stylesheet”>
        <link href=”${styleVSCodeUri}” rel=”stylesheet”>
        <link href=”${styleMainUri}” rel=”stylesheet”>
        <title>Web Pages Panel</title>
      </head>
      <body>
          <div id=”app”></div>
          <script src=”${scriptUri}” nonce=”${nonce}”>
      </body>
      </html>
    `;
  }
}

function getWebviewOptions(extensionUri: vscode.Uri): vscode.WebviewOptions {
    return {
        // Enable javascript in the webview
        enableScripts: true,

        localResourceRoots: [
         vscode.Uri.joinPath(extensionUri, ‘media’),
         vscode.Uri.joinPath(extensionUri, ‘dist-web’),
       ]
    };
}

}
```

You can now see four new files created inside the /dist-web/ directory:

- app.js
- app.css
- sidebar.js
- sidebar.css

Every page generates two files, specifically the JavaScript and CSS files.

That's it for now! The Vue.js pages are ready for hosting inside a Webview.

Step 5

Let's start first by copying the WebAppPanel.ts file from the extension project that uses the Vue CLI. Then you change the resource files to include both /dist-web/app.js and /dist-web/app.css.

Listing 6 shows the entire source code of this file after the changes.

Add a new /src/SidebarProvider.ts file and paste the contents of **Listing 7** inside it.

The SidebarProvider implements the **WebviewViewProvider** interface. It wraps an instance of the **WebviewView** that, in turn, wraps a Webview that holds the actual HTML content.

The resolveWebviewView() function sits at the core of this provider. It's used by VS Code to load the Webview into the Sidebar. It's in this function that you set the HTML content of the Webview for VS Code to display it inside the Sidebar. The provider loads both resource files /dist-web/sidebar.js and /dist-web/sidebar.css inside the HTML.

The HTML of this Webview now contains the following code:

```
<script>
   const vscode = acquireVsCodeApi();
</script>
```

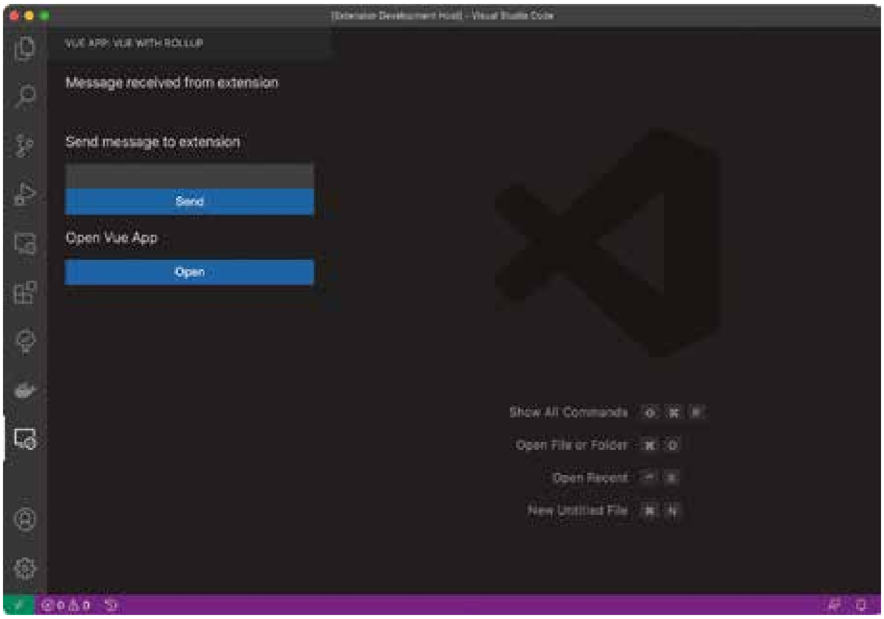


Figure 11: Sidebar.vue component inside the Sidebar

The **vscode** object will be the bridge that the Vue.js app can use to post messages to the extension.

That's it! Let's run the extension by pressing the **F5** key. A new instance of the VS Code opens.

Locate and click the last icon added on the Activity Bar. **Figure 11** shows how the Sidebar.vue Component is loaded inside the Sidebar section.

Step 6

Let's load the App.vue component inside an editor when the user clicks the Open button on the Sidebar.

Go to the /web/components/Sidebar.vue file and bind the button to an event handler:

```
<button @click="openApp">Open</button>
```

Then, define the openApp() function as follows:

```
openApp() {
   vscode.postMessage({
      type: ‘openApp’,
   });
},
```

The code uses the vscode.postMessage() function to post a message to the extension by passing a message payload. In this case, the payload specifies the type of the **message** only.

> Webview API allows two-way communication between the extension and the HTML content.

Switch to the SidebarProvider.ts file and inside the resolveWebviewView() function listen to the message type you've just defined. You listen to posted messages inside the onDidReceiveMessage() function as follows:

```
webviewView.webview.onDidReceiveMessage(
 async (data) => {
   switch (data.type) {
      case “openApp”: {
        await vscode.commands.executeCommand(
                ‘vscodevuerollup:openVueApp’,
                { ...data }
              );
        break;
      }
      // more
   }
});
```

When the user clicks the Open button on the Sidebar, the provider reacts by executing the command **vscodevuerollup:openVueApp** and passing over a payload (if needed).

That's it! Let's run the extension by pressing the **F5** key. A new instance of the VS Code opens.

Listing 7: SidebarProvider.ts

```
import * as vscode from "vscode";
import { getNonce } from "./getNonce";

export class SidebarProvider implements
  vscode.WebviewViewProvider {

public static readonly viewType = 'vscodevuerollup:sidebar';

  private _view?: vscode.WebviewView;

  constructor(
    private readonly _extensionUri: vscode.Uri
  ) {}

  public resolveWebviewView(
    webviewView: vscode.WebviewView,
    context: vscode.WebviewViewResolveContext,
        _token: vscode.CancellationToken
  ) {
    this._view = webviewView;

    webviewView.webview.options = {
      // Allow scripts in the webview
      enableScripts: true,

      localResourceRoots: [
        this._extensionUri
      ],
    };

    webviewView.webview.html = this._getHtmlForWebview(webviewView.webview);

    webviewView.webview.onDidReceiveMessage(async (data) => {
      switch (data.type) {
        case "message": {
          if (!data.value) {
            return;
          }
          vscode.window.showInformationMessage(data.value);
          break;
        }
        case "openApp": {
          await vscode.commands.executeCommand(
             'vscodevuerollup:openVueApp', { ...data }
          );
          break;
        }
        case "onInfo": {
          if (!data.value) {
            return;
          }
          vscode.window.showInformationMessage(data.value);
          break;
        }
        case "onError": {
          if (!data.value) {
            return;
          }
          vscode.window.showErrorMessage(data.value);
          break;
        }
      }
    });
  }

  public revive(panel: vscode.WebviewView) {
    this._view = panel;
  }

  public sendMessage() {
    return vscode.window.showInputBox({
             prompt: 'Enter your message',
                 placeHolder: 'Hey Sidebar!'
               }).then(value => {
                 if (value) {
                   this.postWebviewMessage({
                     command: 'message',
                     data: value,
                   });
                 }
               });
             }

  private postWebviewMessage(msg: {
     command: string, data?: any
  }) {
   vscode.commands.executeCommand(
      'workbench.view.extension.vscodevuerollup-sidebar-view');
   vscode.commands.executeCommand(
      'workbench.action.focusSideBar');

    this._view?.webview.postMessage(msg);
  }

  private _getHtmlForWebview(webview: vscode.Webview) {
    const styleResetUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
         this._extensionUri, "media", "reset.css")
    );

    const styleVSCodeUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
         this._extensionUri, "media", "vscode.css")
    );

    const scriptUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
         this._extensionUri, "dist-web", "sidebar.js")
    );

    const styleMainUri = webview.asWebviewUri(
      vscode.Uri.joinPath(
         this._extensionUri, "dist-web", "sidebar.css")
    );

    const nonce = getNonce();

    return `
      <!DOCTYPE html>
      <html lang="en">
      <head>
        <meta charset="utf-8" />
        <meta name="viewport"
              content="width=device-width, initial-scale=1" />
          <link href="${styleResetUri}" rel="stylesheet">
          <link href="${styleVSCodeUri}" rel="stylesheet">
          <link href="${styleMainUri}" rel="stylesheet">
          <title>Web Pages Panel</title>
          <script nonce="${nonce}">
            const vscode = acquireVsCodeApi();
          </script>
      </head>
      <body>
          <div id="app"></div>
          <script src="${scriptUri}" nonce="${nonce}">
      </body>
      </html>
    `;
  }
}
```

Click the last icon added on the Activity Bar. Then click the Open button. **Figure 12** shows the App.vue component loaded inside a Webview on the editor. The Sidebar.vue component is loaded inside a Webview on the Sidebar.

Step 7

Let's add a command to allow the extension to post a message to the Sidebar.vue component from within VS Code.

Start by defining the **vscodevuerollup:sendMessage** command inside the package.json file as follows:

```
{
   “command”: “vscodevuerollup:sendMessage”,
   “title”: “Send message to sidebar panel”,
   “category”: “Vue Rollup”
}
```

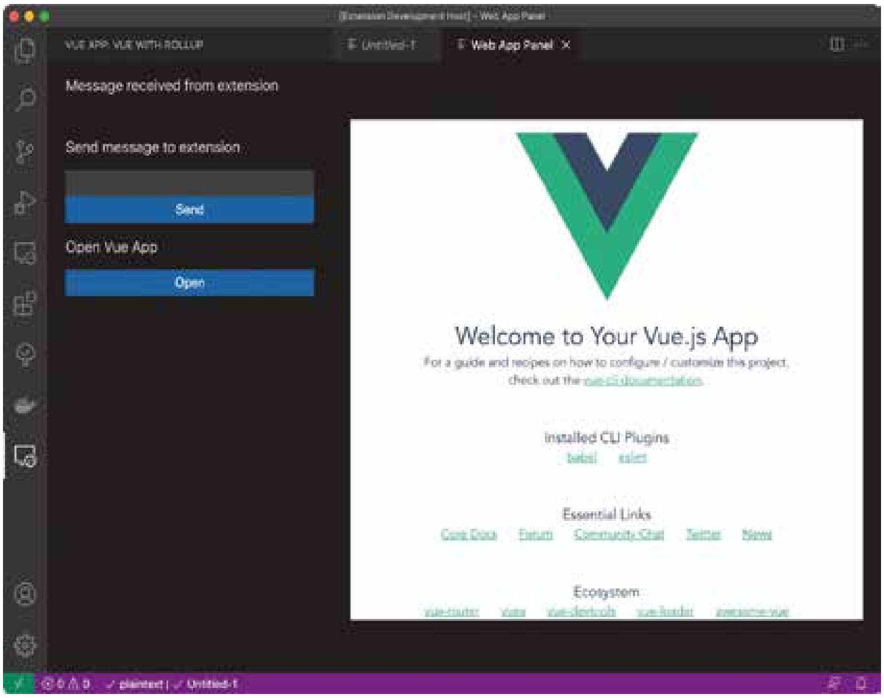

[Extension Development Host] - Web App Panel
VUE APP: VUE WITH ROLLUP
Untitled-1
Web App Panel
Message received from extension
Send message to extension
Send
Open Vue App
Open
Welcome to Your Vue.js App
For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.
Installed CLI Plugins
babel
eslint
Essential Links
Core Docs
Forum
Community Chat
Twitter
News
Ecosystem
vue-router
vuex
vue-devtools
vue-loader
awesome-vue
plaintext
Untitled-1

Figure 12: Sidebar.vue and App.vue components inside VS Code extension

Listing 8: sendMessage() function

```
public sendMessage() {
    return vscode.window.showInputBox({
             prompt: 'Enter your message',
             placeHolder: 'Hey Sidebar!'}
           ).then(value => {
      if (value) {
        this._view?.webview.postMessage({
          command: 'message',
          data: value,
        });
      }
    });
  }
```

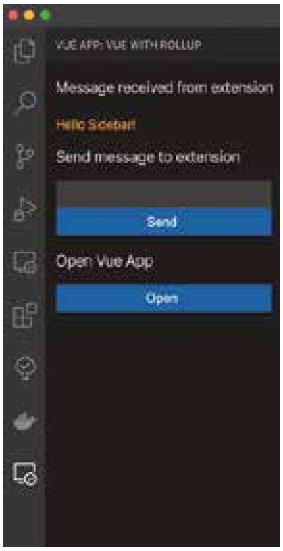


Figure 14: The Sidebar.vue component receives a message from the extension.

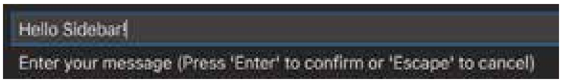
Then, register this command inside the extension.ts file:

```
context.subscriptions.push(
  vscode.commands.registerCommand(
    'vscodevuerollup:sendMessage', async () => {
      if (sidebarProvider) {
         await sidebarProvider.sendMessage();
      }
  })
);
```

The command handler calls the sendMessage() instance function on the SidebarProvider class when the user triggers the sendMessage command.

Listing 8 shows the sendMessage() function. It prompts the user for a message via the built-in vscode.window.showInputBox() function. The message the user enters is then posted to the Sidebar.vue component using the Webview.postMessage() built-in function.

Sidebar.vue component handles the message received from the extension by registering an event listener as follows:



Hello Sidebar!

Enter your message (Press 'Enter' to confirm or 'Escape' to cancel)

Figure 13: Promoting the user for input

```
mounted() {
   window.addEventListener(
     'message', this.receiveMessage
   );
},
```

The receiveMessage() function runs when the user triggers the command inside VS Code.

You define the receiveMessage() function as follows:

```
receiveMessage(event) {
   if (!event) return;

   const envelope = event.data;
   switch (envelope.command) {
       case 'message': {
          this.message = envelope.data;
          break;
       }
   };
},
```

It validates the command to be of type **message.** It then extracts the payload of the command and assigns it to a local variable that the component displays on the UI.

Let's run the extension!

Locate and navigate to the Sidebar.vue component hosted inside the Sidebar.

Open the Command Palette, start typing "Send message to sidebar panel". VS Code prompts you for a message, as shown in **Figure 13**. Enter any message of your choice and hit Enter.

The message will be displayed on the Sidebar, as shown in **Figure 14.**

Congratulations! You've completed your third VS Code extension so far.

Conclusion

You can see how dissecting the VS Code helps your greater understanding of how it works. Once you can grasp the basic behavior of each component, it becomes an easier task to manipulate the code to make it do what you need it to.

This is just the beginning of a new series of articles on extending VS Code by building more extensions. In the coming episodes, the plan is to expand the functionality of extensions and connect to remote REST APIs, databases and much more.

Stay tuned!

Bilal Haidar

ONLINE QUICK ID 2107081

Power BI and R: A Visual Power Punch

I first learned about the impact of data visualization several years ago from the late great Hans Rosling, the Swedish physician and public health professor. His colorful animated bubble charts tell you, despite what you may otherwise believe, that the world is indeed becoming a better place. I use Power BI a lot these days to create my own visualizations. Power BI lets you create scalable dashboards containing updated data analysis to share with a wide user group. It does, however, have limitations for customizing visuals. One way around this leverages the powerful graphic libraries in R to create visuals directly in Power BI. Combining the capabilities of Power BI and R together gives you a visual power punch.

Setting Up Power BI

For this example, you're going to use a volatile dataset that gets updated daily: the WTI spot price. WTI stands for West Texas Intermediate and it's a common measurement of energy futures found in pricing models. It's available from the EIA (Energy Information Administration) government website (https://www.eia.gov), which provides not only analysis for US energy trends but also access to a vast array of datasets you can easily query with its API query tool. You can initially analyze the overall trends for this spot price in a page on the EIA website (https://www.eia.gov/dnav/pet/hist/RWTCD.htm).

If you're following along on your own, you'll need your own API token for the EIA data to place in your own block of M code for the Power Query Editor, which you can find on the EIA website: https://www.eia.gov/opendata/register.php. If you're new to using the EIA API connection, you'll see a form space within this page where you can enter your email address and agree to the terms and conditions for using the API connection to register for your own API token. Once you fill out this form, you'll receive an email with details on how to access your own new API token. You don't need to register for a new API token through the EIA if you already have one, but if you forget yours you can get it from this page as well.

Get Data

Power BI enables you to easily connect to and refresh datasets from a variety of data sources. This project doesn't focus on how to implement the ETL framework in the Power Query Editor, but you can see how to set up a similar project using an API query in an earlier article I wrote on Power Query in CODE Magazine: https://www.codemag.com/Article/2008051/Power-Query-Excel%E2%80%99s-Hidden-Weapon. Note that while the earlier article uses the Power Query Editor in Excel, you can transfer its functionalities and M code directly into Power BI. You can get the data directly in Power BI Desktop by updating the attached starting Power BI Desktop file.

1. Select the Transform data button from the top Home ribbon.
2. Make sure to select the *WTI Prices* query from the query list on the left, then double click on the **Source** step in the **Applied Steps** list on the right.
3. In the open dialog box, where it says **<your_api_token_goes_here>** delete only this character string in the Web connection URL and replace it with your own API token.
4. Confirm this update and you'll see that your query now contains the updated WTI prices from the EIA API.

Choose the Color Palette

If you look at the top of the EIA website (www.eia.gov), you can see their logo. I'd like to bring in not only their logo but also incorporate the colors of the logo into the visuals. I matched the colors in the logo to their hex values using the Adobe color matching tool.

You can see the logo colors displayed in the palette of five colors (**Figure 1**). Keep this color palette on hand, as it will become helpful throughout this project to select consistent colors from the EIA logo.

Framing Time and Trends

The WTI price, as an oil commodity price, fluctuates frequently and you do want to ultimately illustrate these trends in tandem with the R visual you'll create. Placing several visuals in the same Power BI view creates an insightful analysis for the consumers of this data. For those of you starting in the *.PBIX file I included in this article's file, once you update the data to include your own API token, you'll see the initial framework for the analysis that you'll continue to build in this project including:

- EIA logo image
- A date range slicer visual to dynamically select the date range for this analysis
- A line chart visual showing the daily WTI price trends
- Analytics displaying the reference lines on the line chart for the maximum, average, and minimum WTI prices
- Summary cards in on the left displaying the KPIs matching to the reference lines on the line chart

You can see that the price fluctuates quite a bit by the neither smooth nor consistent line chart shape (**Figure 2**). Also, notice the color scheme of the line chart matches up to the blue in the EIA logo for the daily WTI price. The green and yellow colors of the references lines and summary cards match up to the EIA logo as well. The hex values for these colors come directly from the color hex values displayed in the EIA color palette (as you saw in **Figure 1**).

Below the line chart visual, you'll add the R visual to the current white space. My approach for creating Power BI visuals starts with first creating a table visual. This ensures that the dataset values make sense, and the DAX measures I calculate for the model directly in Power BI make sense as well. Let's create a table visual by selecting the standard

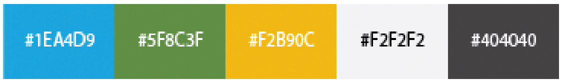


Figure 1: EIA color palette

Helen Wall
www.linkedin.com/in/helenrmwall/
www.helendatadesign.com

Helen Wall is a power user of Microsoft Power BI, Excel, and Tableau. The primary driver behind working in these tools is finding the point where data analytics meets design principles, thus making data visualization platforms both an art and a science. She considers herself both a lifelong teacher and learner. She is a LinkedIn Learning instructor for Power BI courses that focus on all aspects of using the application, including data methods, dashboard design, and programming in DAX and M formula language. Her work background includes an array of industries, and in numerous functional groups, including actuarial, financial reporting, forecasting, IT, and management consulting. She has a double bachelor's degree from the University of Washington where she studied math and economics, and also was a Division I varsity rower. On a note about brushing with history, the real-life characters from the book The Boys in the Boat were also Husky rowers that came before her. She also has a master's degree in financial management from Durham University (in the United Kingdom).

table visual from the Visualization pane. Next, add the WTI Date field to the Values field bucket, then add the WTI price field to the right, so it appears in the table in that order. Notice that it already aggregates the prices. If you compare this to the actual WTI prices, they're the same because this table uses a daily date dimension. But you can also see that adding the date field automatically adds four date dimensions to the table instead of one.

You want to ultimately create an R visual illustrating the averages and distributions by year, so let's remove all the date dimension fields from the table visual except **Year**. Notice that the aggregated values for the WTI price can change. This occurs because you just changed the pivot table coordinates of the table. Whereas before the table aggregated the WTI price by day, it now aggregates the price as a summation over the year. You can see its aggregation type by navigating to the WTI Price field in the Visualization pane, then selecting the down arrow to see the available aggregation options. For numeric values, Power BI defaults to the Sum aggregation type. For this summary though, you want to see the average price for the entire year because prices work like rates, which means that summing up the numbers doesn't make much sense. Once you have a summary table for the average WTI price by year, you can start to create the DAX measure calculations and check them by adding them to the table.

Installing R for Power BI Desktop

For R to successfully run in Power BI Desktop, you need to configure R in two places. First install a compatible version of R on your own computer that works with Power BI Desktop (this project uses version 3.6). Next, you'll need to configure R scripts to run directly in Power BI Desktop through the options menu. If you have several versions of R on your computer, you can choose from a drop-down list of multiple versions of R that will run in Power BI Desktop.

Calculate DAX Measures

Before you start to add DAX measures to the Power BI model, you can create a separate table solely to store these measures. This keeps the model clean and organized because not only can you easily identify the model's DAX measures, but more importantly, others can as well. I already created a Calculations table in the Power BI Desktop model. You can create your own Calculations table by following these steps:

1. Select **Enter data** from the top Home ribbon.
2. In the open dialog box, give this table a new name, like **Calculations**, then select **Load** to save the table.
3. Add a DAX measure to this new table.
4. You can then delete the current existing column and it will only contain this new measure.
5. If you collapse the Fields pane by selecting the arrow point right at the top of the pane and then expanding it again, you'll see that the table name appears with a calculator icon next to it. This indicates that it's a table exclusively for measures and remains that way as you add more measures to it.

You can see a measure already in the Calculations measures table in the existing Power BI Desktop file. This is the dynamic chart name for the line chart visual. If you want to see how it applies to the title name, you first select the line chart visual, then choose the formatting options for this visual. Open the Title submenu and look for the **fx** button, then select it. This opens a dialog box where you see the selected measure already applied for the title name. You can see in the visual title that this measure formula below gives a dynamic date range for the chart depending on the date range you select.

The VARs in this DAX measure formula let you set the variables for the first date selected and the last date selected, which pull from the date range slicer on the left of the view. Each of these formulas uses the DAX function FORMAT to display the date as a long date rather than the short date it displays by default. You then use these two variables to create a concatenated string that RETURN saves as the calculated output value.

```
Title Line Chart =

VAR first_date = FORMAT(FIRSTDATE(
ALLSELECTED(‘WTI Prices’[Date])),”Long Date”)

VAR last_date = FORMAT(LASTDATE(
ALLSELECTED(‘WTI Prices’[Date])),”Long Date”)

RETURN “Daily WTI price trends and fluctuations
between “ & first_date & “ and “ & last_date
```

First, you're going to create a DAX measure that calculates the average for each year, although this may seem odd because you already calculated this in the table visual as the aggregated average WTI price. You're creating this DAX measure because you'll later use it directly in other DAX measures. Select New Measure from either the top ribbon in Power BI or by selecting the ellipsis (three little dots) next to the Calculations table name. Next, you want to add the DAX measure expression into the formula space. You can calculate the average price by setting it equal to the CALCULATE function, which you'll wrap around the AVERAGE function to return the average WTI price.

```
Average Price = CALCULATE(AVERAGE(
‘WTI Prices’[WTI Price]))
```

The table pivot coordinates determine the results of this calculation. When you add it to the table visual, it calculates the average WTI price by year. However, if you add perhaps the month date dimension to the table, these aggregation values will change because you're no longer evaluating the calculation on a yearly basis, but a monthly basis.

Next, you want to calculate the standard deviation for the WTI price by creating a new measure for the standard deviation Like the average price DAX measure, you'll use the CALCULATE function, but inside the function you'll use the STDEV.P function. The P on the end of this function indicates that you're calculating the standard deviation on a population. Now you add it to your table visual, to make sure the calculation looks correct alongside the average prices for the year.

```
Standard Deviation = CALCULATE(STDEV.P(
‘WTI Prices’[WTI Price]))
```

But what exactly does the standard deviation mean? Let's take a step back to examine key concepts of statistics in the context of this analysis. The average WTI price comes from calculating this aggregation for each day in the work week for a year, which amounts to about 260 data observations per year. The prices can fluctuate quite a bit even within a single calendar year. The standard deviation measures the variance of these prices. You can take this standard deviation value and add or subtract it from the average yearly price to determine the lower and upper bounds of the range containing roughly 68% of that year's data, which you see in the two middle shaded sections of **Figure 2** (source is https://commons.wikimedia.org/wiki/File:Normal_Distribution_Sigma.svg). Adding or subtracting two standard deviations gives you the range for 95% of the data (**Figure 2**).

Let's calculate a price range using two standard deviations to get the Min Price and Max Price.

Next, you want to take a standard deviation DAX measure to calculate the minimum and maximum values representing a 95% confidence interval to display in the R visual. You'll create another new DAX measure for the minimum price lower bound. You will then take your average price DAX measure and subtract two times the standard deviation DAX measure from it. Notice that you're not using the CALCULATE function in this calculation because you're referencing two already calculated measures.

```
Min Price = [Average Price]
- 2 * [Standard Deviation]
```

Lastly, you'll create another DAX measure for the maximum price, but this time you'll add two times the standard deviation to the average price DAX measure instead of subtracting it from it.

```
Max Price = [Average Price]
+ 2 * [Standard Deviation]
```

Once you create these measures, add them both to the table visual alongside the average price measure (**Figure 3**). You can see how you now have a range of calculated values for each year, including a minimum calculated price, a maximum calculated price, and the average price.

To create the R visual, you want to leverage these three fields plus the Year date dimension from the model. This means that you can delete the other fields in the table to clean it up before moving to the next step of this project. Once you select the R visual for this set of data, you're going to transition to using R exclusively to build out the visualization.

Tapping into R Visualizations

Next, you want to enable R scripts to run directly in Power BI Desktop. To do so, you'll need to set up two processes.

Install R

First, you need to install R on your own computer if you don't already have it there. You can have multiple versions of R on your computer, but you'll specify the version for Power BI to connect to. Set up the R-CRAN for the area of the world that you live in. Once you install R, open the RGui to install the additional libraries you'll use in this project. In the interface, type in install.packages("ggplot2"). Then you'll want to install the packages for "scales" and "extrafont" in the same way.

Enable R Scripts

You also need to enable R scripts directly in Power BI Desktop. Navigate to the options menu within the Power BI Desktop home page and choose to enable R scripts. You'll receive a confirmation message for this set up. You also want to make sure that you select the R version 3.6 from the drop-down menu. The R version will certainly change in the future, but for now, make sure to select this version of R for the scripts to run properly.

Initiate Visual

To check that you enabled the R scripts to correctly set up a table in Power BI, convert it to an R visual by selecting the R icon in the visualization option list. Power BI automatically sets up initial code for the R visual script denoted with green font using the data fields you already selected (**Figure 4**).

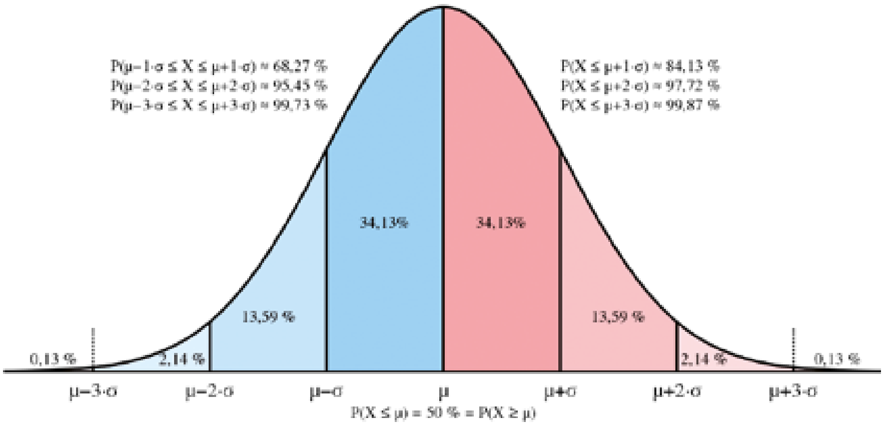


Figure 2: Normal distribution of data

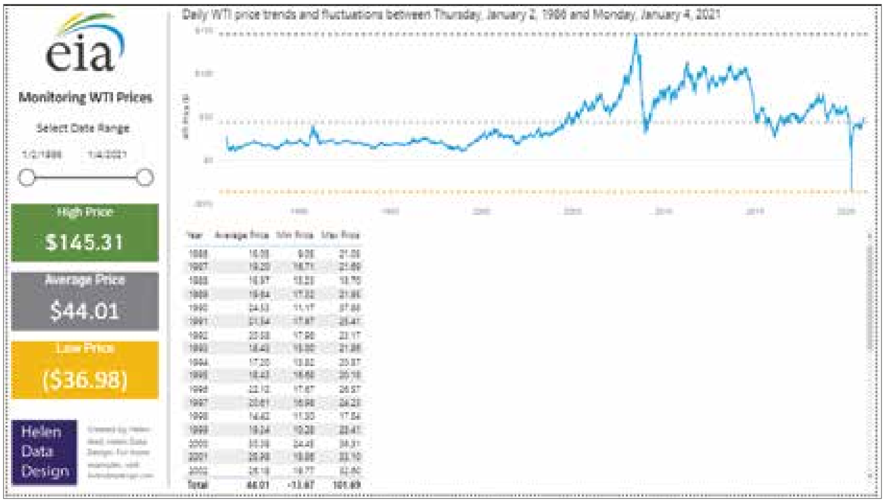
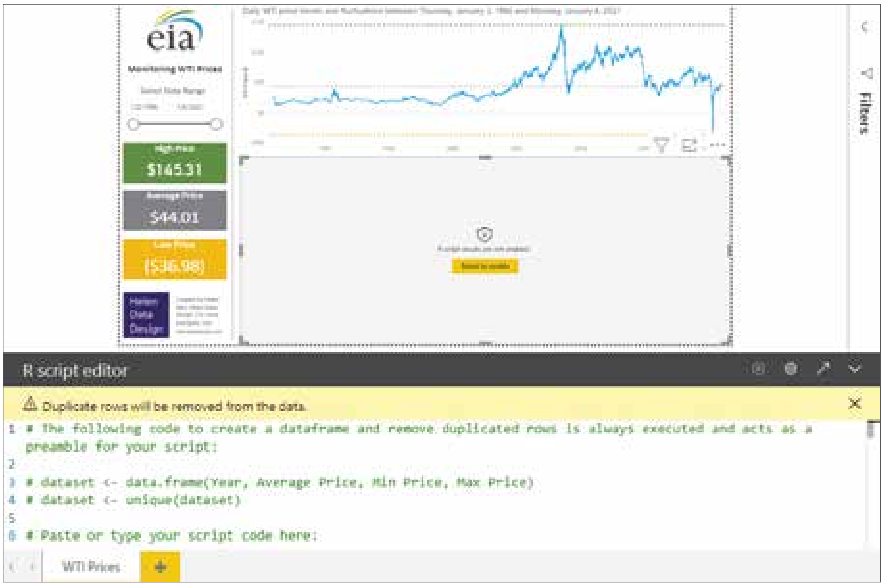


Figure 3: Table summary of DAX measure calculations



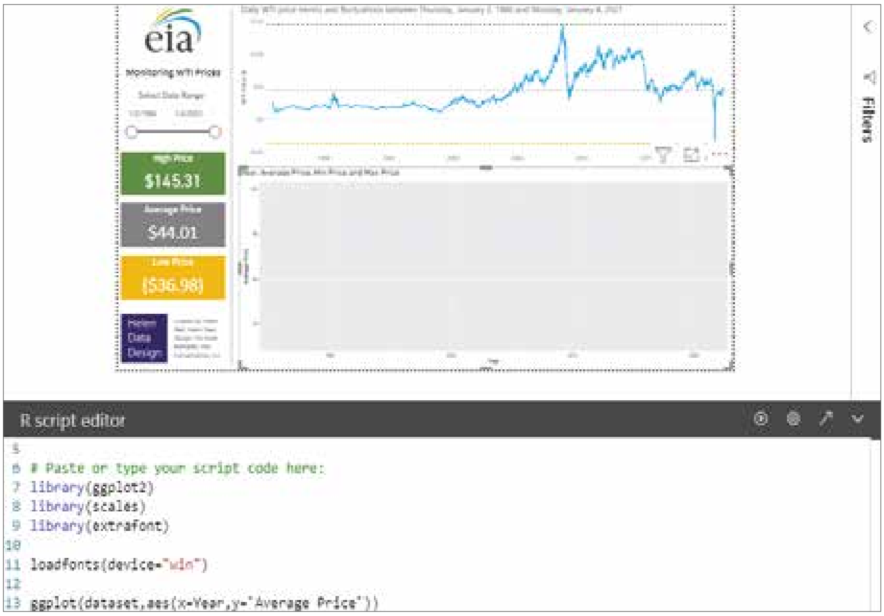
⚠ Duplicate rows will be removed from the data.

```
# 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(Year, Average Price, Min Price, Max Price)
# dataset <- unique(dataset)

# Paste or type your script code here:
```

Figure 4: Initial R script in Power BI standard R visual

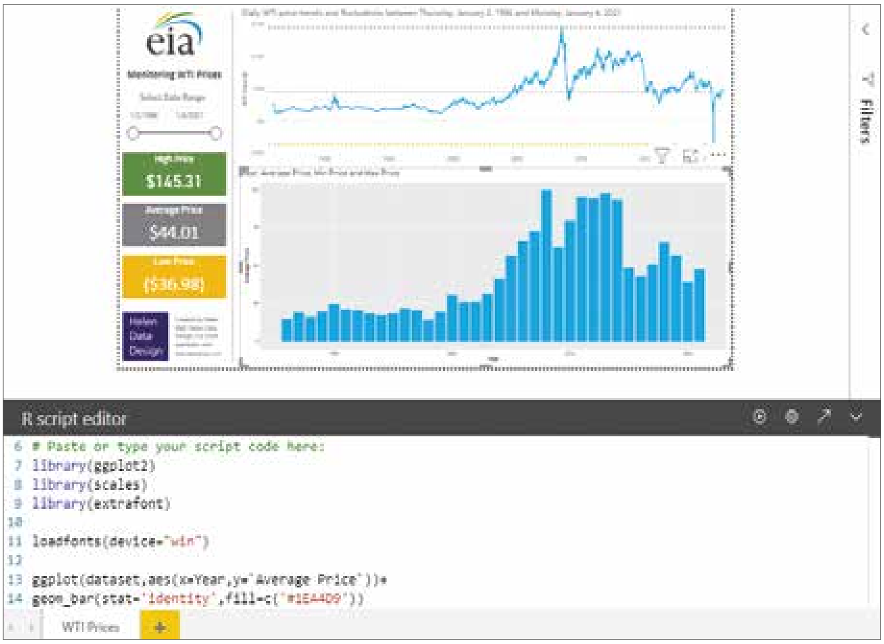
This lets Power BI do some of the planning process for running the R script for you. Power BI brings in the data into an R data.frame it calls a dataset, then runs another command directly after that to create a unique dataset for the visual. If you remove one of the data fields from this R visual and then added another field, the dataset wouldn't update, but rather you could leave the existing fields alone, or you can change the field names manually yourself. You can also see that the dataset has another line of code below it that returns the unique dataset. You see a final line that says paste or type your script here. Here's where you start to add your own code to create your custom R visual.

First, make sure to enable R script by selecting the yellow button directly on your R visual (see message for "R script visuals are not enabled" in **Figure 4**). Even though you imported the packages for this visual in the RGui, you also need to import them directly in this code to properly run the R script. To run the ggplot2 package in this visual, you add it to the first line of your own code by telling the R visual to load the ggplot2 library. You'll do the same for the scales and extrafont library (**Figure 5**). The next line of R code with the loadfonts command tells the R script that you're running this R script on Windows so it references the font options on the appropriate operating system.



```
# Paste or type your script code here:
library(ggplot2)
library(scales)
library(extrafont)

loadfonts(device="win")

ggplot(dataset,aes(x=Year,y=`Average Price`))
```

Figure 5: Initial R visual with ggplot library on canvas



```
# Paste or type your script code here:
library(ggplot2)
library(scales)
library(extrafont)

loadfonts(device="win")

ggplot(dataset,aes(x=Year,y=`Average Price`))+
geom_bar(stat='identity',fill=c('#1EA4D9'))
```

Figure 6: Change bar color.

In the next line, you call the ggplot function to initialize your R visual. You'll then reference the dataset that Power BI automatically created initially for the R visual as an input for this function, along with identifying the fields that go on the axes of this visual. Let's put the Year on the x-axis and the Average Price on the y-axis. Notice the back quotes around the '**Average Price'** field. Because the Average Price field contains spaces to make it easier to read, you need to add single-quotation characters around the field name for the R script to properly run the code. Otherwise, the script errors out because the field name isn't a single string, but two separate words separated by spaces. Next, you'll hit the play button icon in the top right of the R script window to run the R script within Power BI Desktop. This creates an R visual on the canvas with those fields on the axes. Notice that there's no chart yet. You'll create this chart in the next line of code.

Create the Bar Chart

You can choose from many chart options to run in R, but let's create a bar chart in this example. To create the bar chart, you want to first add a plus sign (+) to the existing code, and then use the geom_bar function to create a bar chart within this space. Set the stat for the geom_bar function to '**identity'**, which tells the R script to use the x-axis field you already set in the previous line of code. When you hit the play button to run the R script in Power BI again, it now creates a bar chart in the same space (**Figure 6**). If you want to explore other R visual options, check out this guide to R visuals from the R Studio website: https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf.

If you don't include the fill within the chart function, the visual defaults to a very dark grey chart color. There are several different ways you can add colors to your R script. You can add more code after **stat = 'identity'** to pass the fill color into the R script. You can set this fill parameter to a particular color like 'blue,' but you can also specify the exact color to match a hex color value like those in the EIA logo (**Figure 1**)., when you run the R script again, Power BI displays a bar chart visual with bars that exactly match the blue hue in the EIA logo.

Add Standard Deviation Bars

Next, let's add error bars to the existing bars in the visual to illustrate the range of values in the 95% confidence interval for each year of WTI price data. The error bars go directly on top of the blue bars representing the average WTI price for each year. To add them to the R script, first put the plus sign (+) at the end of the previous line of code. Then, in the next line, you'll use the geom_errorbar function to create these error bars. Within this function, you'll need to nest the **aes** function. The aes, or aesthetics function in ggplot2 creates visual characteristics within a chart including color and fill, point shapes, line type, size, or group. The aesthetics function lets you bring in groups or fills into your charts. For geom_errorbar, this includes the width of the error bar, which you set to 0.7, but you can set it to another value. If you play around with this value, you can see how changing to impacts the appearance of the R visual.

You can also set the color of these bars to the dark grey color in the EIA logo by using the **fill = c(#404040)** in a similar way to which you set the bar color to the blue in the EIA logo. To pass in the parameters for the bottom of the error bar, set **ymin** within the **aes** function to the Min Price field already added to the R visual. You'll do the same to add the top of the error bar by setting the **ymax** equal to the Max Price field. Remember to include back quotes around each field name because their names do contain spaces, otherwise, the R script will error out! When you run the visual, you'll see the error bars neatly added to the top of the blue bar chart (**Figure 7**).

Reformat Axes

Adding scale_x_continuous to your R script tells Power BI that you want the R visual to display the x-axis to using even breaks for the years. To do so, you want to use a function from the **scales** library that works with the ggplot2 library. You also want to calculate within the R script the number of years to use along this axis, which you do by calculating the length of the dataset Year field. Putting this within the scale_x_continuous function that you're adding to the line of code that creates the error bars lets you scale the x-axis tick marks to equal to the number of years in the date range of the current dataset**.** When you run this code, the R visual adds a label for each year with a tick mark to the x-axis for each bar in the bar chart (**Figure 8**).

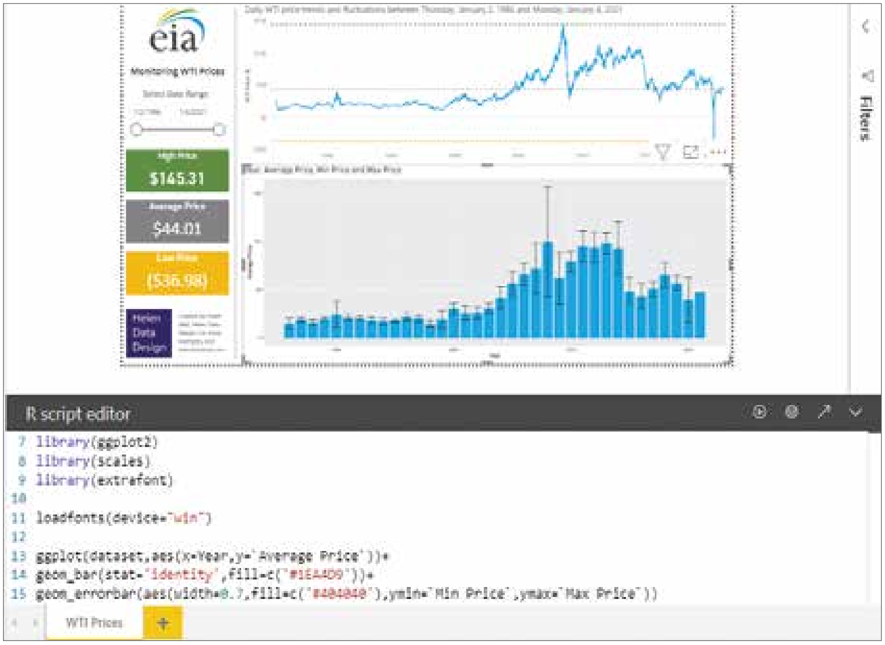
If you run the R script without adding the expand parameter at the end of this function, you'll notice that the chart displays a few leading and trailing years without data to the beginning and end of the date range before 1986 and after 2021. Eliminating these extra years in the visual improves readability, but more importantly, it also removes any confusion of potentially thinking the WTI prices are zeros in these years.

Change the Background View

Notice that even after updating the formatting for the x-axis in the previous step, the visual still displays a light grey background grid behind the bars and their error bars. Although this doesn't make the visual incorrect, it does clutter the canvas a bit. It would look a bit cleaner with a white background. To remove the default background from the R visual, you'll add another line of code after the previous line for scale_x_continuous with the plus sign. On the next line, adding the function theme_bw() removes the gray background (**Figure 9**).

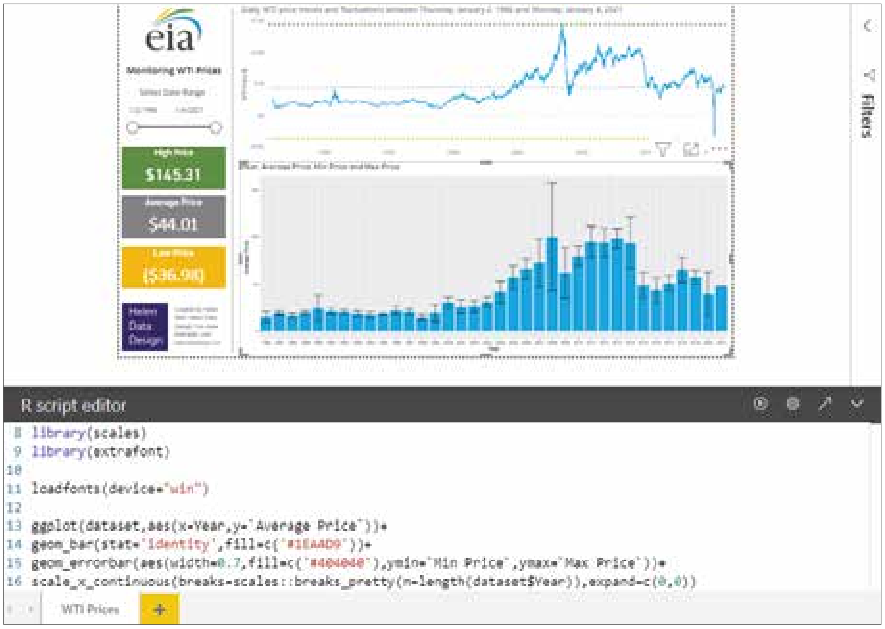
Add Labels

The ggplot2 library offers not only a plethora of chart options for the visual, but also a quite extensive array of formatting options. Let's say you want to add easy-to-read labels to the axes or the title. You can make these updates by adding a single line of code to the R script. Again, use the plus sign to add another function to the existing R script, then use the **labs** function to pass in the new titles for the axes and main title that you want to update.



```
library(ggplot2)
library(scales)
library(extrafont)

loadfonts(device="win")

ggplot(dataset,aes(x=Year,y=`Average Price`))+
geom_bar(stat='identity',fill=c("#1EA4D9"))+
geom_errorbar(aes(width=0.7,fill=c("#404040"),ymin=`Min Price`,ymax=`Max Price`))
```

Figure 7: Add error bars to bar chart.



```
library(scales)
library(extrafont)

loadfonts(device="win")

ggplot(dataset,aes(x=Year,y=`Average Price`))+
geom_bar(stat='identity',fill=c("#1EA4D9"))+
geom_errorbar(aes(width=0.7,fill=c("#404040"),ymin=`Min Price`,ymax=`Max Price`))+
scale_x_continuous(breaks=scales::breaks_pretty(n=length(dataset$Year)),expand=c(0,0))
```

Figure 8: Reformat x-axis.

Remove Gridlines

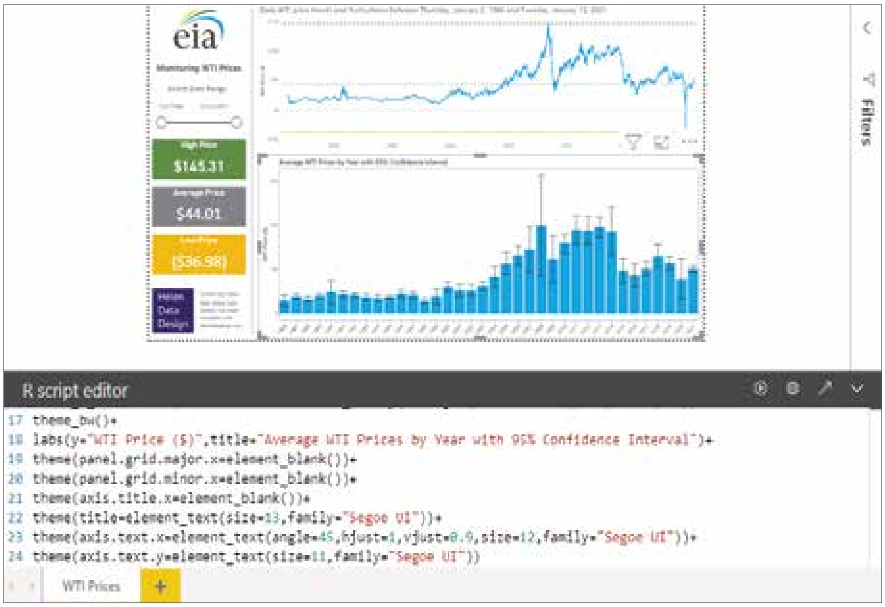
You can also use the theme() function within ggplot2 to change the formatting of your R visuals. You can set the grids to an empty view by making **panel.grid.major.x** and **panel.grid.minor.x** both equal to element_blank(). Running the script after adding these two new theme lines to the existing script results in the removal of the grid along the x-axis, but keeps the grid along the y-axis, which makes it easier to quantify the average, high, and low WTI price values for each year (**Figure 9**).

Add Labels

Because the year appears in both the title and the x-axis label, it seems reasonable to remove the x-axis label to avoid

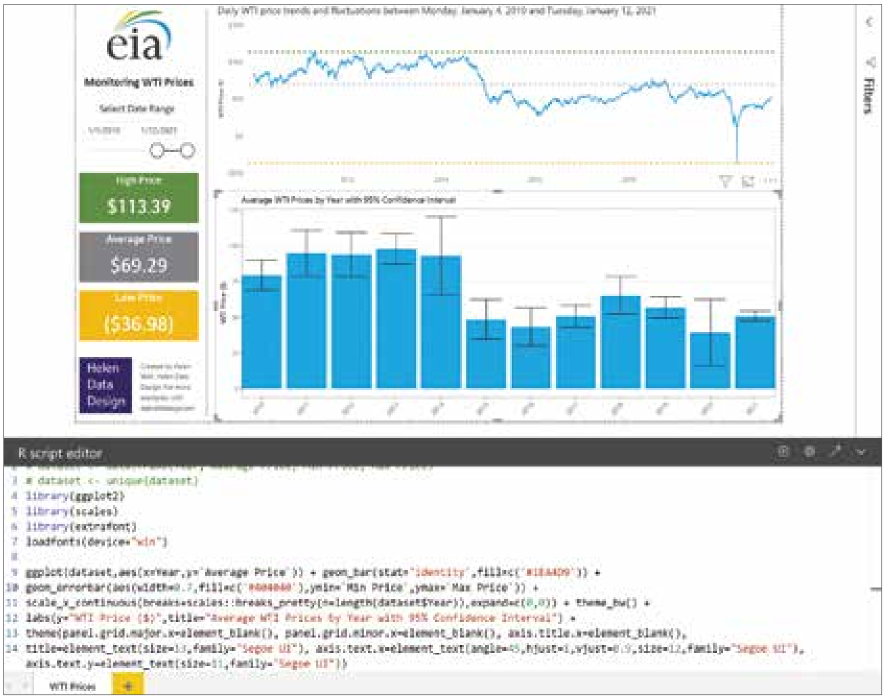
Remember Those Plus Signs!

Notice between all the functions in the R script, you can see plus signs joining them together The code to create the visual properly won't run properly if you don't remember to include these plus signs!



```
theme_bw()+
labs(y="WTI Price ($)",title="Average WTI Prices by Year with 95% Confidence Interval")+
theme(panel.grid.major.x=element_blank())+
theme(panel.grid.minor.x=element_blank())+
theme(axis.title.x=element_blank())+
theme(title=element_text(size=13,family="Segoe UI"))+
theme(axis.text.x=element_text(angle=45,hjust=1,vjust=0.9,size=12,family="Segoe UI"))+
theme(axis.text.y=element_text(size=11,family="Segoe UI"))
```

Figure 9: Changing the visual formatting.



```
# dataset <- unique(dataset)
library(ggplot2)
library(scales)
library(extrafont)
loadfonts(device="win")

ggplot(dataset,aes(x=Year,y=`Average Price`)) + geom_bar(stat='identity',fill=c('#1BA4D9')) +
geom_errorbar(aes(width=0.7,fill=c('#404040'),ymin=`Min Price`,ymax=`Max Price`)) +
scale_x_continuous(breaks=scales::breaks_pretty(n=length(dataset$Year)),expand=c(0,0)) + theme_bw() +
labs(y="WTI Price ($)",title="Average WTI Prices by Year with 95% Confidence Interval") +
theme(panel.grid.major.x=element_blank(), panel.grid.minor.x=element_blank(), axis.title.x=element_blank(),
title=element_text(size=13,family="Segoe UI"), axis.text.x=element_text(angle=45,hjust=1,vjust=0.9,size=12,family="Segoe UI"),
axis.text.y=element_text(size=11,family="Segoe UI"))
```

Figure 10: Final view with consolidated R script for the visual.

repeating labels and give the visual a clean appearance. To remove the x-axis label, you can add another line of code to the existing script again using the theme() function, but this time you want to set the axis.title.x equal to element_blank() before running the updated code (**Figure 9**).

Adjust Text Sizing and Font

Your R visual is starting to show some nice insights for the WTI price trends, including illustrating that higher average prices by years doesn't necessarily mean the variance is higher as well. You can continue to make formatting changes to this visual, including changes to the sizing and font for any text on the chart. The R library **"extrafont"** lets you leverage an extended array of font options with an R visual (**Figure 9**). First, you'll leverage the theme() function again by adding additional lines to the code. Nested within this function, you can set axis.text.x, axis.text.y, and title equal to the element_text function. You'll then pass the parameters for the actual element formatting into this element_text function. The size parameter refers to the font size of the axis mark labels or the title. You can set Family equal to the font family you want to use. In this case, you can see the script references for the font family Segoe UI, which the visual imports from the "extrafont" library. You can select from many font family types in the "extrafont" library, but Segoe UI matches the title text in the line chart above the R visual. If you run this code as is, you'll see subtle differences between the text appearances, even though they both have the same font size and type.

For the x-axis specifically, you can see several applied formatting options, including the angle (in this case, 45 degrees) to which the text labels sit in relationship to the direction of the x-axis. The parameters **hjust** and **vjust** lets you move each of the labels by an incremental adjustment in direction either horizontally or vertically from their original location to change the axis label markers and make them easier to read. You can play around with these values by changing them from the ones you already see to experiment with what the adjustments update (**Figure 9**).

Put It All Together

You just created your own custom visual in Power BI by leveraging the existing standard R visual within Power BI Desktop! You likely noticed that the R visual has two titles—one you added in the R script and that the existing default chart Power BI automatically includes with the visual. To turn off the visual title, select the R visual again, then go to the formatting options in the Visualizations pane and navigate to the Title submenu where you can simply turn the default Title radio button to the off mode.

Also, you may notice that, all told, the script for creating the R visual has 24 lines within Power BI. Granted, the initial R visual configuration Power BI automatically created several of those lines, but you can streamline the code by consolidating the transformations nested within the theme() function into a consolidated line of code within the R script at the end of the script (**Figure 10**). You can also consolidate these functions into a few lines of R code that you'll find in the final attached Power BI Desktop.pbix file for the project.

Notice how analyzing the visual output of this chart in tandem with the line chart and summarized WTI price trends gives valuable insight about the trends in WTI price on a daily basis, as well as how these prices fluctuate between years and even within individual years. You can see how adjusting the date range in the slicer on the left side of the page lets you analyze the WTI price trends over a narrower date range and changes some of the sizing proportions in the R visual for this narrower data range. This sample project also only represents a small part of the capabilities of R within Power BI! You can explore many more options for leveraging R visuals directly within Power BI, as well as Python and D3.

Helen Wall

R (and Python and D3) in Power BI

This project leverages R to create custom visuals within Power BI, but it isn't the only available language for visuals. You can also create your own custom Power BI visuals by utilizing scripts for Python and D3.

(Continued from 74)

how something as macro as organizational commitments directly relates to something as micro as source code control. What if the organization is a public entity? Add many, many more commitments around SOX, for instance. How do you verify SOX compliance? Via SOC/SOC-2. That's an example of strategy and tactics. You may think that it's like peeling the layers of an onion. It isn't, though, because in these cases, we start from the core and work our way out. That's where the bodies are buried. That's where the truth is. That's where all the small things are! It's there, in that sea of things at the core of the organization, where we find out if we're able to meet our commitments.

Honoring commitments, however, doesn't necessarily mean that there must be perfection 100% of the time. No project is 100% perfect. And yet, there are many, many successful projects. There's always failure, to at least some degree. The question is what does our commitment say about these situations where deliverables haven't been met, either in whole or part? What does the contract say about breach? In my opinion, it's always best to confront and adopt these procedures from the start. Otherwise, there's too much wheel-spinning and finger-pointing instead of doing the things necessary to remedy the failure! Job #1 is honoring commitments, not being perfect. Nobody and no thing is perfect.

It's necessary to make commitments. It's necessary to honor those commitments. It's necessary to enable the honoring of those commitments. Promises and commitments, like everything else, build on one another, like a sturdy structure. The key is to focus on how your organization goes about it, which includes what it prioritizes as necessary. And if you find there are issues, work from the inside out, starting with the small things. Often, the small things are easy to fix!

John V. Petersen

CODE COMPILERS

CODE

Jul/Aug 2021
Volume 22 Issue 4

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Editorial Contributors
Otto Dobretsberger
Jim Duffy
Jeff Etter
Mike Yeager

Writers In This Issue
Bilal Haidar
Julie Lerman
Rod Paddock
Paul D. Sheriff
Joydip Kanjilal
Sahil Malik
John V. Petersen
Helen Wall

Technical Reviewers
Markus Egger
Rod Paddock

Production
Franz Wimmer
King Laurin GmbH
39057 St. Michael/Eppan, Italy

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

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

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: The NEWS Group (TNG)
Media Solutions

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $50.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379
Phone: 832-717-4445

CODA: On Commitment…

To commit. To be committed. To make a commitment. Each statement can be met with a generic response: "To?" To what have I committed? To whom have I committed to do something by a certain date? "Commit" is an interesting word, a verb with two basic and quite different meanings:

- Carry out—perpetrate
- To pledge or bind

The first definition is about the act itself, whatever it may be. What was carried out, perpetrated? Unless we're talking about source code control, the word "commit" sounds downright nefarious!

The second definition is about those things that occur ***before*** what we commit to accomplish, and an earnest effort has begun. In another context, we may think of one being core development and the other being all those things that must occur before core development may begin; such things include pre-sales proofs of concept and terms of service negotiation. That's the commitment, the ***promise***. When the deal is "signed, sealed, and delivered," the popped champagne corks must turn to the team's code craft, for there is a software product to be delivered. That's the commitment, which as it turns out, is made up of several other commitments. In the legal world, it's a contract, a promise in exchange for a promise or some action. In the present context, the many commitments we all make to each other to deliver software is usually evidenced by a legal contact. The point here, before getting into the details, is that our commitments are the key, up and down an organization, to making things work. And they all build on one another, sometimes like a house of cards. Although it may be a bit of a mundane topic, it's often good to revisit such things because when we go back to reflect when on what went wrong, it's typically the little, mundane things that went awry.

What was promised? Does it depend on who you ask? If it's the lead sales team, the answer will be clear, whatever that answer may be. And if there is ambiguity, the statement of work (SOW) should provide clarity. If there's no clarity, there's a bigger problem, so whatever answer you get doesn't really matter! What if we asked the dev team what was promised? A response would be received to be sure. If not, then just like before, the answer you get doesn't matter. Assuming we get two actionable and clear responses, to what degree, in terms of organizational and project impact, would the two responses align? If they don't, that's yet another different problem. But if they're aligned, are they aligned legally with what was actually promised? This is why clear and consistent communications is so necessary in order for an organization to and its constituent people to meet their commitments. There are three basic groups/parties we need to be concerned with:

- Those who make the promises (e.g., sales)
- Those who fulfill the promises (e.g., development)
- Those who are the beneficiaries of such promises (e.g., the customer)

How we carry out our commitments is just as important, and perhaps **more** important than just meeting the commitment. It's my opinion that in any rational business (and I choose to punt on the irrational kind), there's sufficient amount of earnest ***desire*** to do the right thing, in the right way, for the right reasons. Sticking with the rule of three, another leg of the stool may be that despite the strong desire, the ***effort*** required to meet the commitment's letter isn't feasible. ***Why is that?*** Generically speaking, it's either ignorance, malice, or some combination of the two. At the core is information, who has it and who doesn't. In any successful organization, information moves as effectively as it needs to. It need not be perfect, just good enough.

Just good enough—for how long? In perpetuity? That can't work. When we make decisions to pin an effort or to just outright whack a feature, we're making an organizational commitment that we've assessed the risk. But the only way that can work is if ***people*** hold each other ***accountable***. Clear and effective communications are one important ingredient. Assuming we have that, what happens when commitments aren't being met. Why is that? That's where accountability comes into play. It's the device by which we see to it that we honor our commitments. And in the most successful organizations, people hold themselves accountable.

No tool or automated promise is going to force anybody to do anything. Only people, ***cloaked in appropriate authority,*** can do that. Assuming the happy path here, where there's enough shared vision of mutual respect, transparency, etc., the thing referred to by many is the Accountability Chain. The chain is bi-directional in that it goes up and down the organization. In other words, if everyone is in the same boat, everyone has made and has signed up to the promise, to the commitment, from the CEO to the intern. Commitments and the law are very similar. Often, there must be enforcement for them to be worth anything.

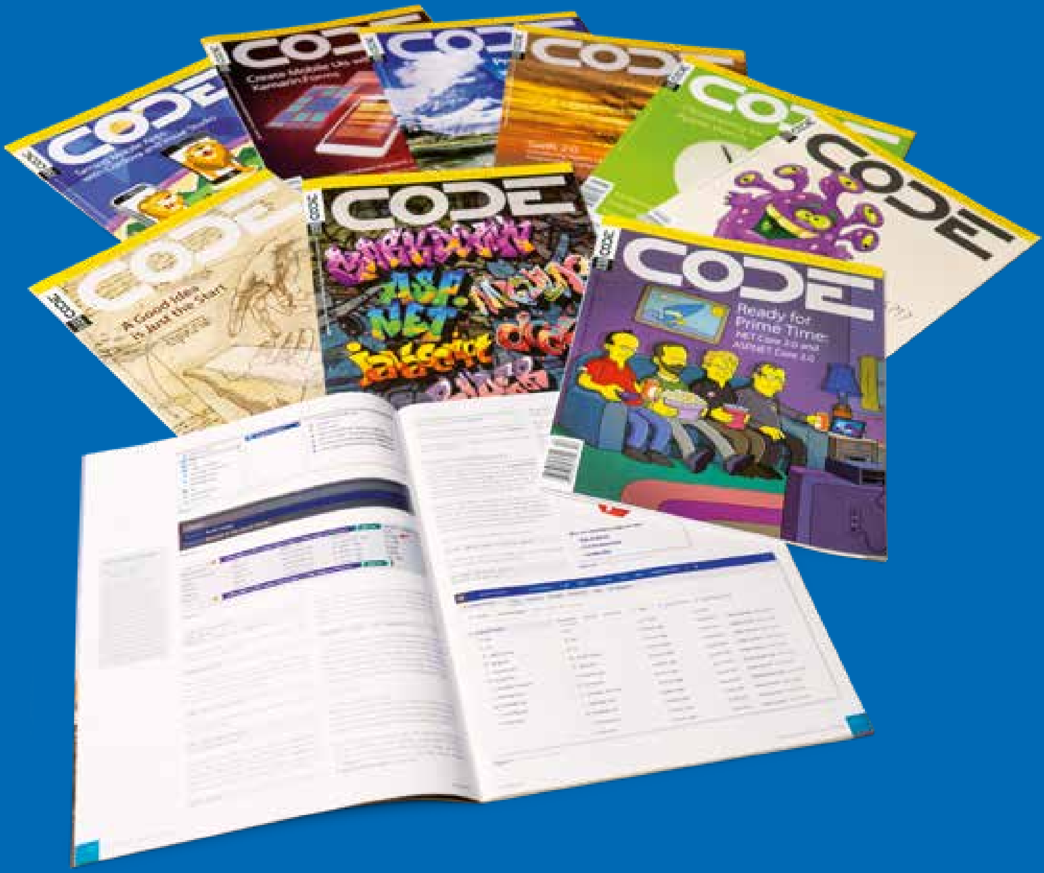
Instead of asking what is good enough, the more important question may be whether the team is ***enabled*** to meet "good enough." And for the record, I'm not suggesting that "good enough" is akin to chucking it over the wall! I suggest that we must define **what** must be good enough first and then second, see to it that we can actually meet the promises we're making.

As a practical matter, a good place to look for guidance on whether a deliverable is of sufficient quality (good enough) may be found in your ***Definition of Done***. If you have one, chock that up as another commitment that needs to be honored! As I stated earlier, promises and commitments build upon each other. If you haven't codified a definition of done or such other similar thing, if you take nothing else from this editorial, take that! And by that, I don't mean any aspirational talk or philosophies on how things should work in a utopian environment (business school). Take with you the commitment your team is going to codify and establish a definition of done!

For a moment, I'd like to revisit the mundane, the simple, the small things, not so much because they matter. It's because the small things tend to be authentic and, therefore, can't be faked. By small things, I mean those things that folks just ***know*** about an organization; how it works, what makes it tick, what motivates it, its people, etc. Some may call it tribal knowledge. This sort of information is vital because these things all go to how information moves through the organization, a thing our ability to honor commitments depends upon.

What I'm referring to are the "small things." Small things are units of work. For example, how does the shop implement Git, and perhaps more granularly, pull requests? You may be wondering

(Continued on page 73)


CODE
Ready for Prime Time: .NET Core 2.0 and ASP.NET Core 2.0
A Good Idea Is Just the Start
Create Mobile UIs with Xamarin Forms

Commercial UAV Expo – Americas