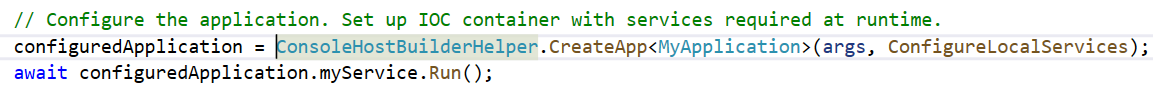
**Azure Support Secure Configuration (Reference Application)**

This document will provide details of how to set up this reference application to function correctly. The purpose of this reference application it to provide working code that demonstrates some of the concepts that are detailed in the application design document Intro-Intake-ReloAccess-Design-Document.docx. Specifically, we will provide code samples that demonstrate:

* How to configure any application without the need to hard-code any sensitive secret/password information directly in your code or configuration file that might end up in source control.
* How to construct classes to support the façade design pattern, allowing dependency injection to create the appropriate/correct class at runtime. This makes your code much more flexible and testable, and hence, reliable.
* How to set up a flexible logging pipeline so that you can send your log files to multiple destinations using only a little simple configuration.

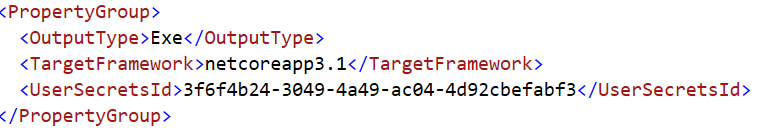
The reference application is a .Net Core 3.1 console application. As such, dependency injection is not configured by default. The ConsoleHostBuilderHelper class is used to set this up, configure the application and get it running:



The first step in securing the application configuration is to set up “User Secrets”. Right click on the main project file (AppSupportSecureConfiguration.csproj) and select “Manage User Secrets”.



If your application was not set up to support user secrets yet, Visual Studio will inject a “UserSecretsId” section into the .csproj file:

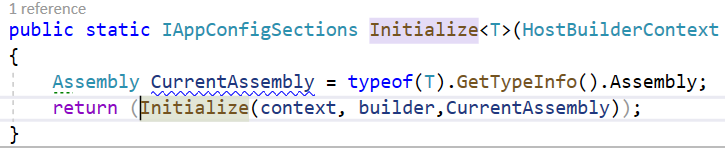


Because you’re downloading an existing application, the existing value in the .csproj file will be used. Visual Studio will open a file called “secrets.json” located in :

C:\Users\owner\AppData\Roaming\Microsoft\UserSecrets\3f6f4b24-3049-4a49-ac04-4d92cbefabf3

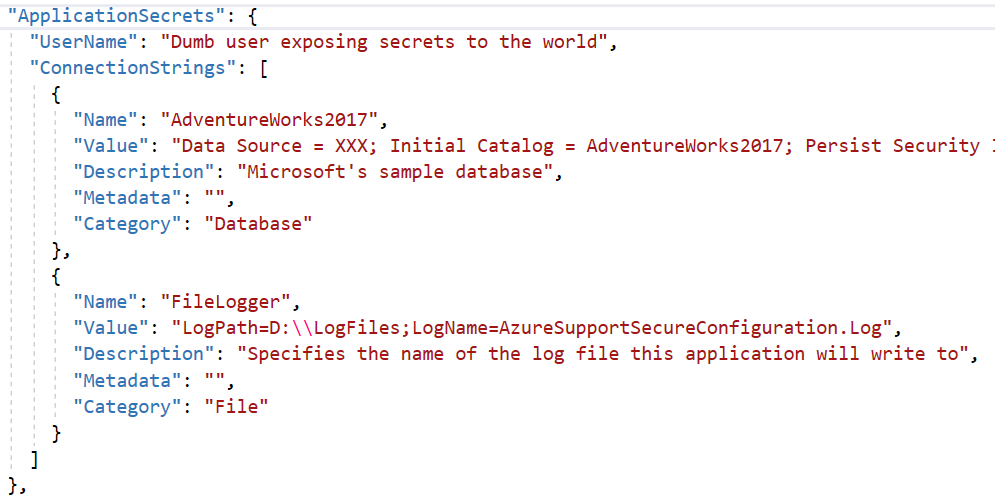
Each new user secret will create a new unique child folder from …\Microsoft\UserSecrets. As you can see the name of the folder is identical to the id used in the “UserSecretsId” section.

The “CreateApp<MyApplication>” method that is used to configure the app has a generic parameter <MyApplication>. The reason we need this type is to support “User Secrets”. The way user secrets works is that the code inspects the assembly that contains the <UserSecretsId> section in the .csproj file. The way we ensure that the code inspects the correct .csproj file is to pass in a type that exists inside that assembly. We pass the assembly that the “MyApplication” type lives in to the user secrets infrastructure in order for the code to know where to look:

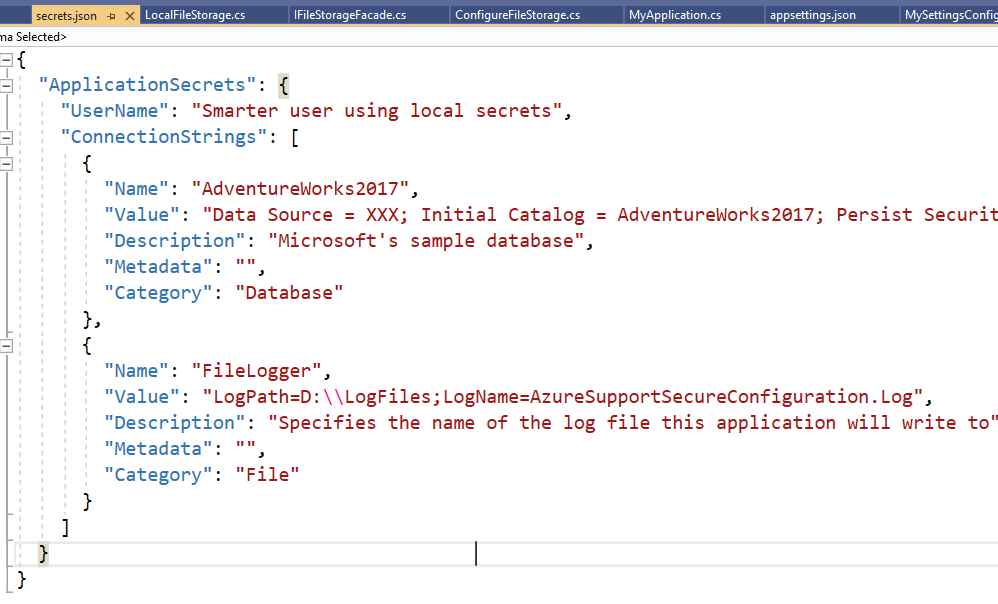


When the secrets.json file is opened, it will be empty. It is up to you to copy any sensitive information from your application.json file and put it into the secrets file. At runtime the secrets file is merged with appsettings. The values found in secrets will override those found in appsettings.

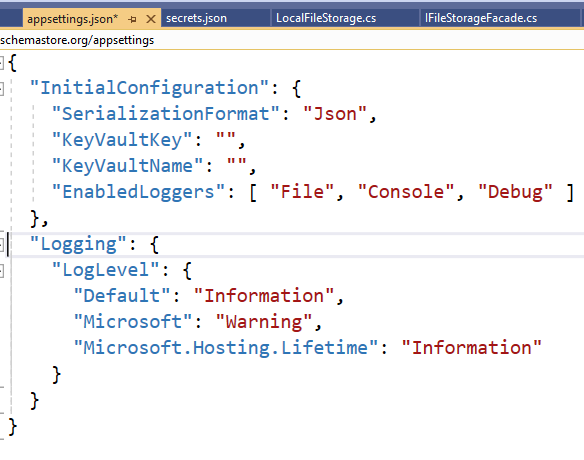
The following information in the appsetting.json file is considered sensitive. This is the information that we will eventually remove from our system. Everything in the “ApplicationSecrets” section is considered sensitive and should be removed from appsettings.json.



The first step is to copy this section into the secrets.json file so it looks like this:

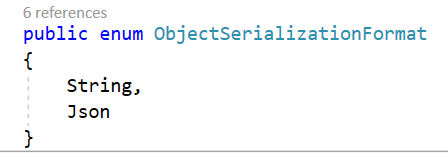


The next step is to remove the ApplicationSecrets section from appsettings.json. The file will now look like this:



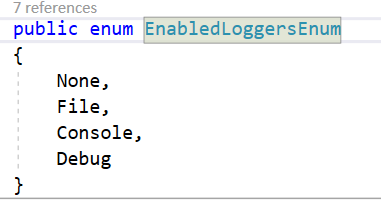
There are 2 main interfaces that are created from the configuration files. The first is IApplicationSetupConfiguration . This interface provides a property for each of the values you see in the “InitialConfiguration” section of the config file. These values dictate which logging mechanism(s) will be used at runtime, the serialization format used when serializing data to be logged as well as whether Azure KeyVault will be used to retrieve application “secrets”. A secret can be ANYTHING you don’t want to keep in your appsettings. It can be a password, an ApiKey, a database connection string, ANYTHING!

SerializationFormat maps to the following values:



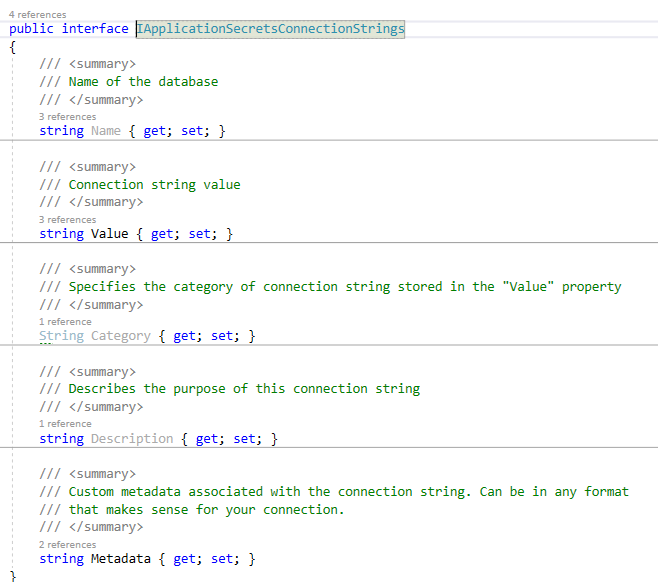
If “Json” is specified, then any data being serialized for logging will be serialized as formatted JSON. If “String” is specified then “ToString()” is called on the object being serialized. If the object does not override ToString, then whatever the default implementation of ToString for that object is used. Usually this means only the name of the object is produced.

EnableLoggers is mapped to the following values:

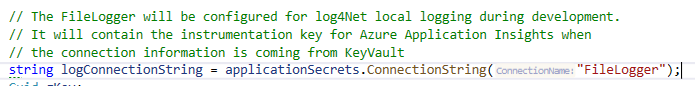


If the array is empty or the value “None” is specified, then no logging will take place. If File is specified, then either local file log4Net logging will be used or logging to Azure Application Insights will be used. This depends on the values specified “KeyVaultKey” and “KeyVaultName”. If these values are empty, then local file system log4Net logging will occur. Otherwise, these values specify the location of the logging workspace in Azure Application Insights.

The next interface is IApplicationSecrets . This interface allows you to access all of your secret values regardless of where they came from. This interface is tied to the “ApplicationSecrets” section of the configuration file. Each item in the “ConnectionStrings” array is considered a “secret” because it contains some kind of password, or other information you don’t want others to know. Each element in the array is mapped to the following interface:



If you simply want to get the value of the connection string, use the “ConnectionString” method. It will locate the correct secret by “Name” and then return the contents of the “Value” property. E.g.

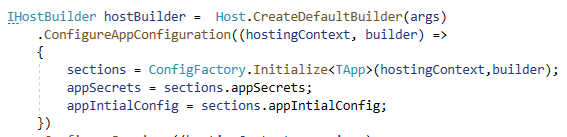


If you want access to all the properties in the interface, then use the “Secret” method e.g.:

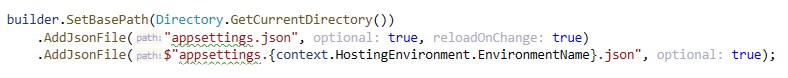
All of the interfaces are initialized and configured by “CreateHostBuilder”. This code initializes a console application to use dependency injection and read the configuration information. You can easily adapt this code to be used in any web application or API.



The code that sets up the order in which configuration information is loaded is located in ConfigFactory.Initialize:



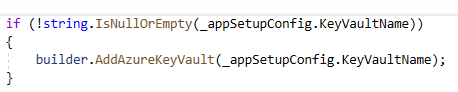
In there you can see that configuration starts by loading “appsettings.json”



We ensure that local appsettings values are overridden by “secrets.json” by adding user secrets to the pipline:



If the “KeyVaultName” provided in the configuration is not empty, then we add AzureKeyVault as a location to get configuration information from:



If KeyVault was enabled, then the following code will pull the “secret” from the KeyVault using the “KeyVaultName” and “KeyVaultKey” values:

If we provided the following initial configuration, then the code above would have accessed an Azure KeyVault named “myapp-keyvault-prod". It will request the secret contained in the key named “AllMySecrets”:



Although you can call your vault anything you like, it makes sense to follow some kind of convention to make it easy for you to pull the right secrets for the current environment the code is executing in. By simply appending the name of the current environment onto the end of the KeyVaultName, your code can be deployed into any environment and it will pick the right vault to retrieve secrets from.

In KeyVault, you’d create multiple vaults such as:

Myapp-keyvault-PROD

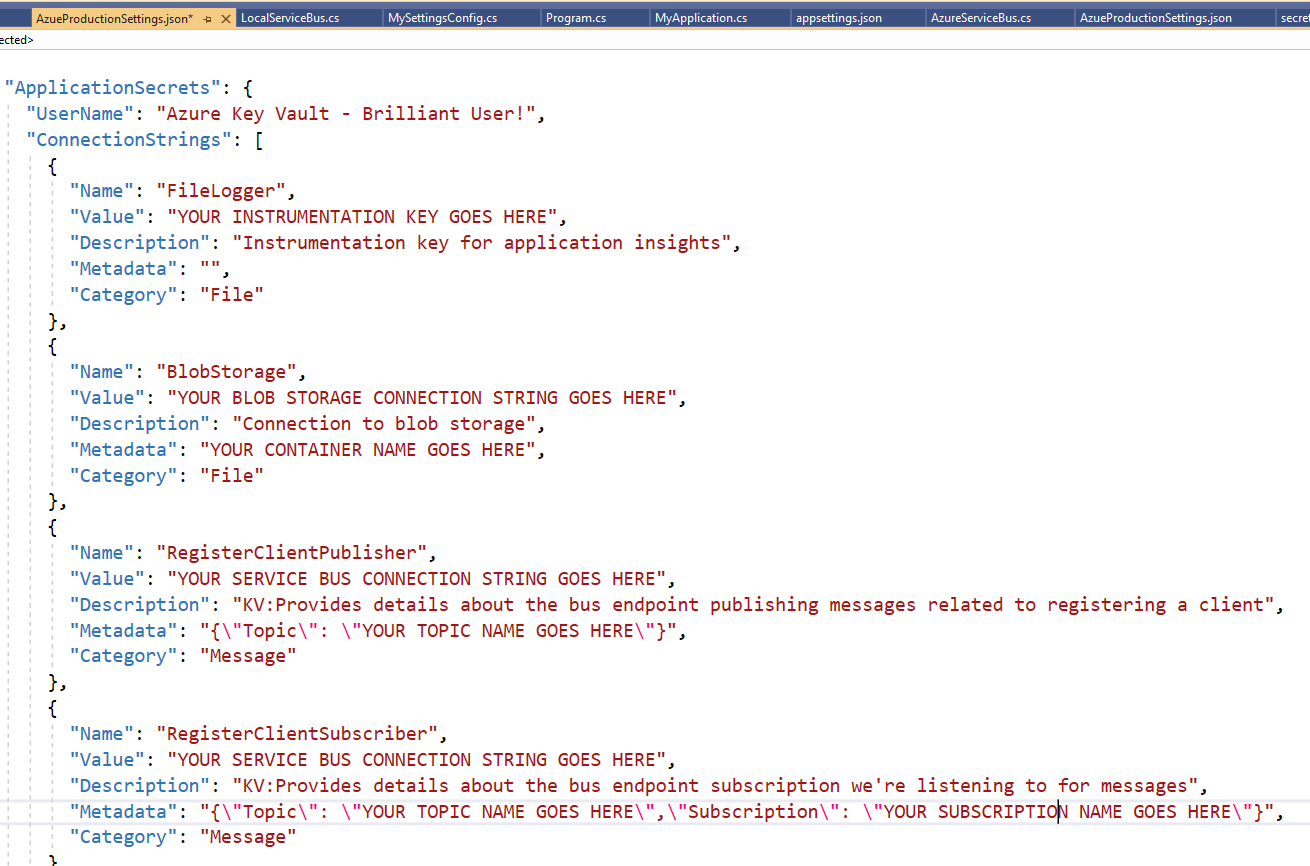
Myapp-keyvault-DFX

Myapp-keyvault-QFX

Myapp-keyvault-Q6

The value “PROD”, “DFX”, “QFX” etc would come from a machine local environment variable named something like “MYAPP\_KEYVAULT\_ENV”. You’d then take the “KeyVaultName” provided in the InitialConfiguration section and append the value you found in the environment variable “MYAPP\_KEYVAULT\_ENV” to produce the name of the specific vault you want to connect to.

Assume that the following is the data stored in the key “AllMySecrets”:



At this point the IApplicationSecrets points to all the secret information that was stored in Azure KeyVault. As you can see I have one secret named “FileLogger” which gives me the information I need in order to connect to Azure Application Insights, where I will send my log information.

I have another secret that contains the information I need in order to connect to my Azure Blob Storage. When you run the demo we will copy a file from local storage to blob storage based on the information provided in the “BlobStorage” secret. If the demo did not provide a value for “KeyVaultKey”, then none of these secrets would have been downloaded from KeyVault. As a result the only secret information I would have is what was stored in “secrets.json” on my local machine.

**Configuring Azure to support the demo**

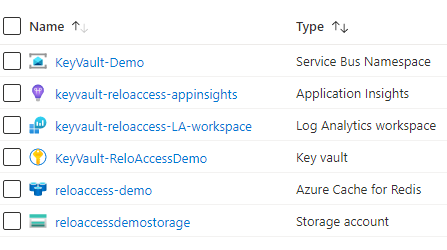
In order for you to run this reference application, you’ll need to set up a couple of things in Azure first.

I will assume you have an azure subscription, even a free one in order to run the reference app. Once you log into the Azure portal you’ll need to create a resource group where all your other resources will be stored. A resource group is like a named shopping bag. It provides a way to organize/group related resources. When you destroy a resource group, you automatically destroy all other resources placed in that group. The name you chose for your resource group is up to you. Many resources in Azure must have unique names across organizations, so I can’t give you specific names that you must name your resources. As you create your resources for this demo, write down their names and various resource-specific values that Azure generates for you to uniquely identify that resource.

You’ll need the following for this reference app to work.

1. Azure subscription
2. A resource group to contain the other resources
3. KeyVault resource
4. Blob storage resource
5. Application Insights resource
6. Service Bus resource
7. Redis Cache resource

Once you’ve completed these tasks, you should see something like this in your Azure portal. These are the resources that you need in order to the demo to work using Azure. The names of these resources will be different in your portal. The names you see here are the resources I created for my own demo.



One thing you’ll notice is that the location of the resources are not all the same. The reason for this is that some of the resources are NOT available in all regions. So, it’s up to you to pick the location closest to you when creating the resource.

A KeyVault resource is required to store all your secret/confidential application settings. These are the values that you used to keep in your appsettings.json file.

Once you’ve been authenticated to KeyVault, the rest of the application runs simply on the values in the configuration file(s).

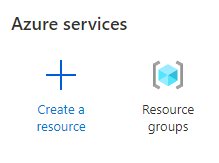
A Blob storage resource is required to demonstrate the ability for the app to store files either locally or in the cloud, depending on application settings.

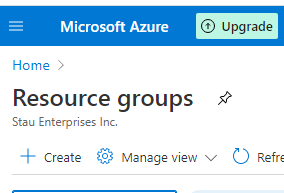
The Application Insights resource is required in order for the application to log information into the cloud workspace instead of the local file system. Again, the values you configure in appsettings.json will dictate how the overall application is configured and which code is executed at runtime.

**Create a resource group**

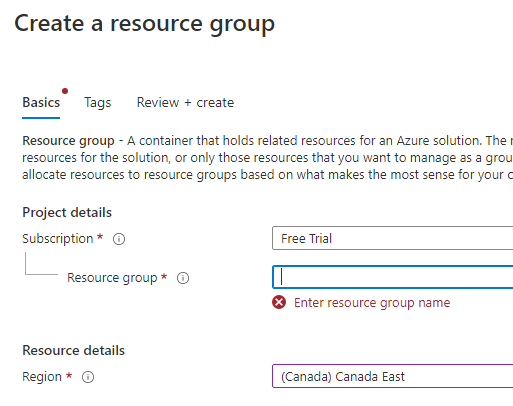
Open the Azure portal and log in. <https://portal.azure.com/#home>

You can either click on the “Resource Groups” or “Create a resource” icon.





Once you’re on the resource groups page, click the “+ Create” icon.



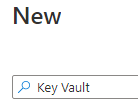
Select the name of the subscription you want this resource group created for. Also select Canada East as the region of the data center where you want this resource group created. Finally, give the resource group a meaningful name. The convention is to separate a multi-word group name with dashes. Also, you should append “-rg” on the end of the name to identify this as a resource group. You’ll find that as you create more and more resources in Azure it starts to become confusing as to what name belongs to which resource. By appending a suffix, it makes it easier for you to identify various resources. For example:

**bgrs-reference-app-rg**

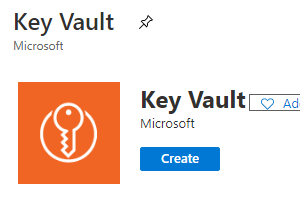
Click the  button to review your information. If you’re happy with what you see, you click . You’ll now see your new resource group name displayed in your list of resource groups. This is now your ”named shopping back” into which you’ll place the rest of the resources that you’ll create.

**Setting up KeyVault**

Click on the resource group you’ve just created and then click on the  icon in order to add a new resource into the resource group. You’re now taken to the marketplace where you can pick the type of resource you want to add. Type ”Key Vault” in the search window and press enter.

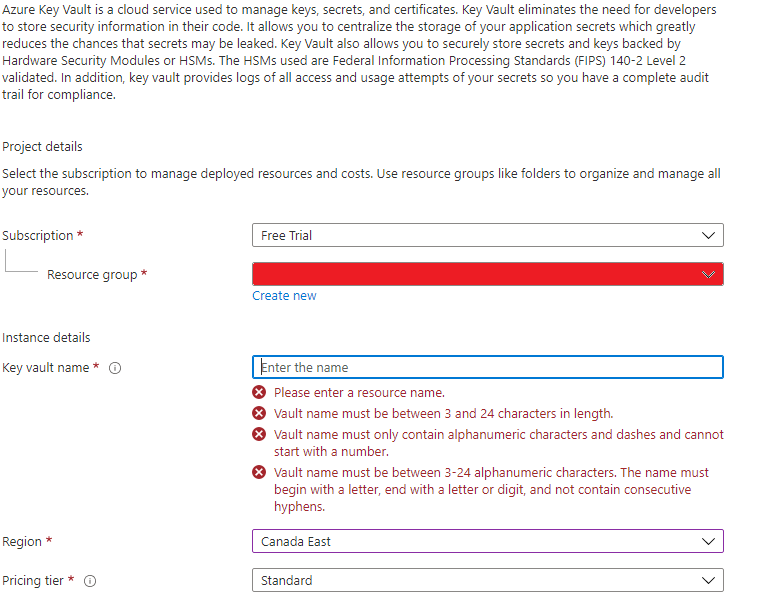


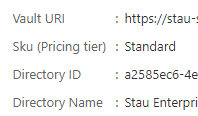
You’ll be taken to the Key Vault page where you can create a new vault:



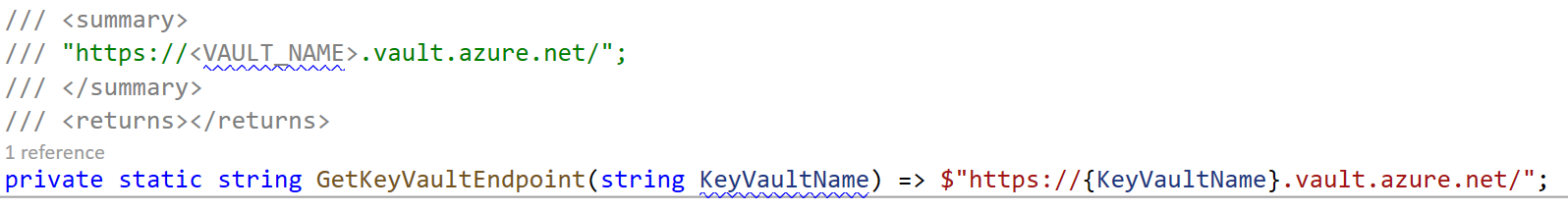
Click Create. This KeyVault must be placed into a resource group. Enter the name of the new resource group you created in the previous step. This is the field highlighted in red. The name of the KeyVault must adhere to certain rules to be considered a valid name. Remember to append “-kv” to the name to make it clear what this resource represents. To keep things consistent, I’d recommend a name something like this:

**bgrs-reference-app-kv**

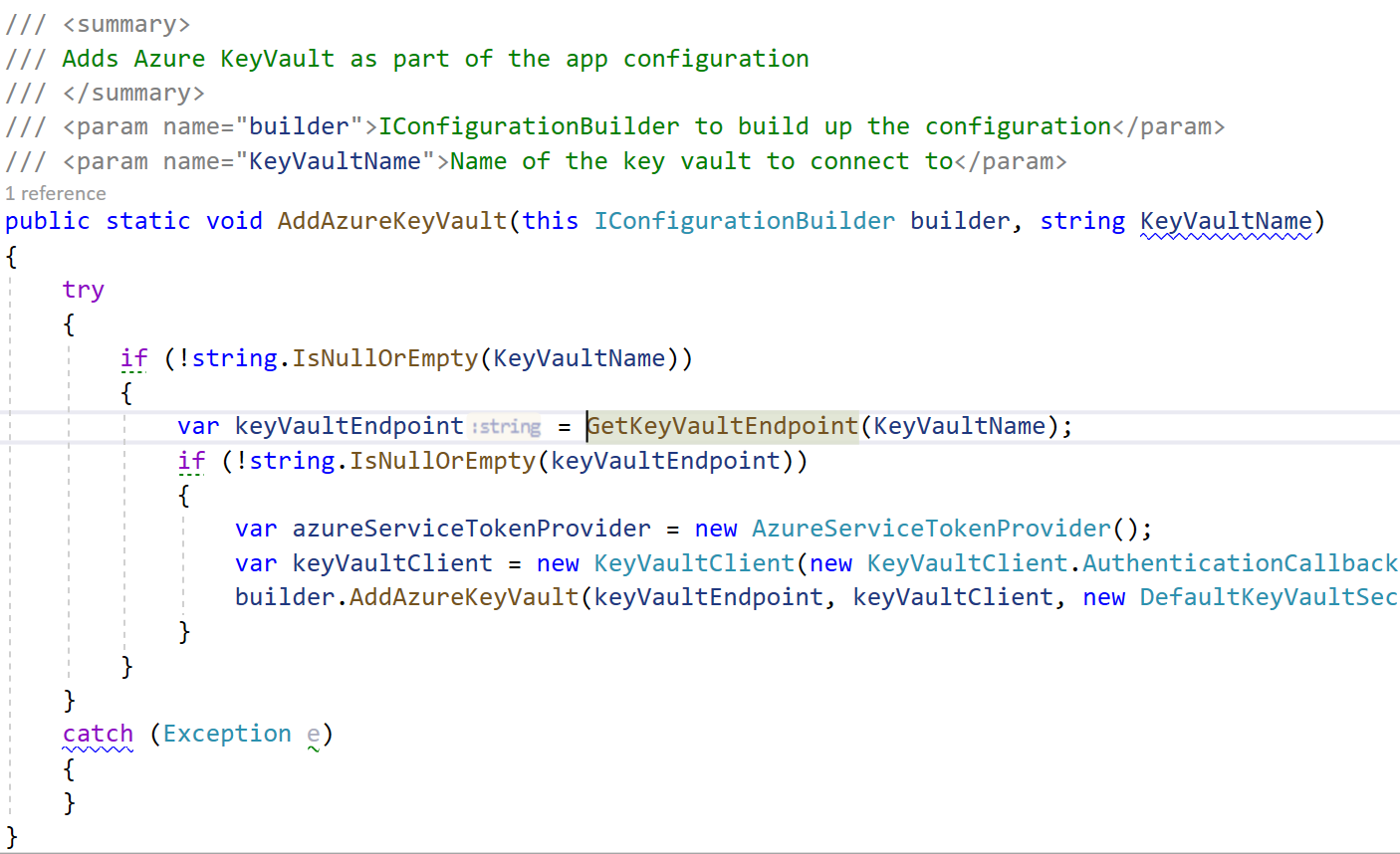
Click the  button to review your information. If you’re happy with what you see, you click . It takes about 30 seconds for the resource to be created and deployed. In order to use the KeyVault with your application, you’ll need to remember some information. Click on the ”download details” button to download all the details about your KeyVault. Click on the button  to get the URI that you’ll need to access your vault.

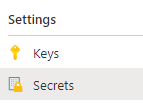


The Vault URI is always constructed with the following pattern:

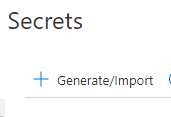


The following code shows how the KeyVaultName is used to integrate KeyVault into our configuration pipeline. When KeyVault is integrated, when we request any appsettings value (using standard IConfiguration interface), that value will be requested from the secret values that were retrieved from KeyVault.

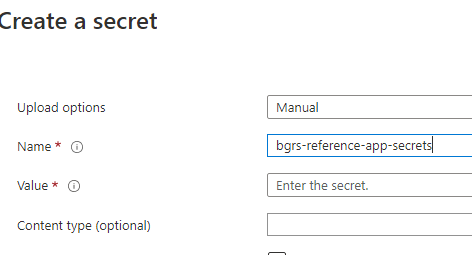
The reason we use KeyVault is that it provides a secure location to store the information that would otherwise be in plain text in your appsettings.json file. In order for us to store information in KeyVault, we have to create a named key. We then store our secret information in that key. Let’s create our key by selecting our new KeyVault and then click on “Secrets”:



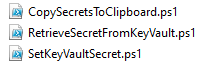
Next, click on “+ Generate/Import” to create a new secret.



Enter the name of the key you want to create. You’ll be storing your appsettings secret values in here.



Although multi-line key values are supported by KeyVault, the Azure portal UI does not a way to input them.The UI provides a single line to enter your secret value for any key. The Azure CLI on the other hand does not have this limitation. If you have multiple secrets that you need to store, using the portal, you’d have to create a key for each secret. You’d have to repeat this for each environment you need to support with different secrets. To solve this problem, we encode all key values as base64. This allows us to use complex json files to maintain different configurations for all our environments and then simply encode them to base64 before installing it in a key in KeyVault. A key value can be up to 25K in size so encoding configuration files to base64 and then manually inserting them into a key via the portal is not an issue. To simplify this task, the reference application includes 3 powershell scripts:

 dex3Open a powershell window and [Install Azure Powershell](https://docs.microsoft.com/en-us/powershell/azure/install-az-ps?view=azps-5.4.0) .You need it in order to run these scripts. Your session must be authenticated too. Use the “Connect-AzAccount” command.

* **CopySecretsToClipboard** contains parameters that allow you to specify which file you want to encode to base64. It then copies it to the clipboard so that you can paste the value in the Azure portal if you’re setting the key secrets manually.
* **SetKeyVaultSecret** uses the CopySecretsToClipboard script to generate the base64 encoding and then updates the KeyVault key you specify with the encoded value. You must login to azure in your powershell window in order for this script to work (Connect-AzAccount).
* **RetrieveSecretFromKeyVault** allows you to specify which secret you want to retrieve. It will automatically decode the base64 value so you can see the secrets in plain text.

The reference app contains a sample file for you to use to install your own secrets into the key you created. Its name is “AzueProductionSettings.json”. It is a sample that you should use to enter the secrets that you’ve created. Assuming the folder structure where you’ve downloaded the reference app is:

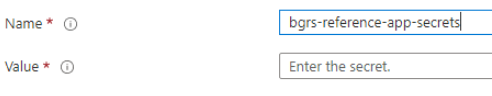
D:\BGRS\Code\ AzureSupportSecureConfiguration

If you open a powershell window and enter the following command, you’ll see the base64 representation of the data in the “AzueProductionSettings.json” file:

.\CopySecretsToClipboard.ps1 "D:\BGRS\Code\AzureSupportSecureConfiguration\AzureSupportSecureConfiguration\AzueProductionSettings.json" $true



This value has been automatically copied to the clipboard. You “COULD” go to your Azure KeyVault and “paste” int the value:



This will work just fine. But, it’s kind of a pain to go through so many manual steps. A better way is to use the powershell script “SetKeyVaultSecret.ps1”. This will create the base64 encoding and automatically set the value into the KeyVault key.

For example, to encode the contents of the ‘AzueProductionSettings.json’ file to base64 and place it into a key called ‘bgrs-reference-app-secrets’ in a valut called ‘bgrs-reference-app-kv’:

.\SetKeyVaultSecret.ps1 'AzueProductionSettings.json'

'bgrs-reference-app-kv'

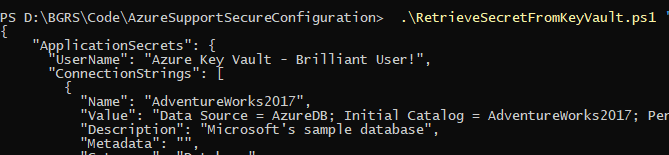
' bgrs-reference-app-secrets'

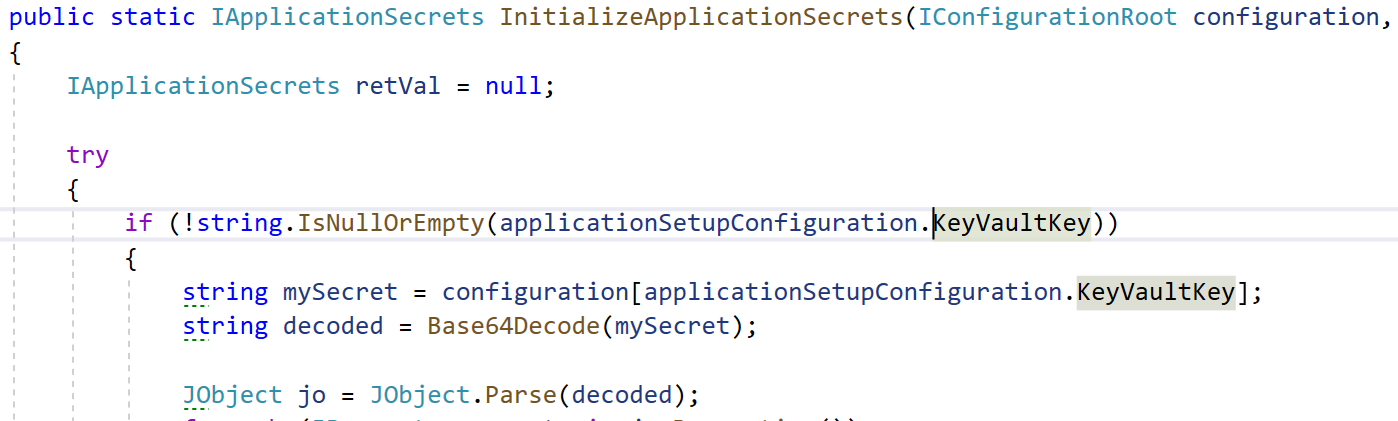
To retrieve your secrets from this KeyVault key, you can use:

.\RetrieveSecretFromKeyVault.ps1

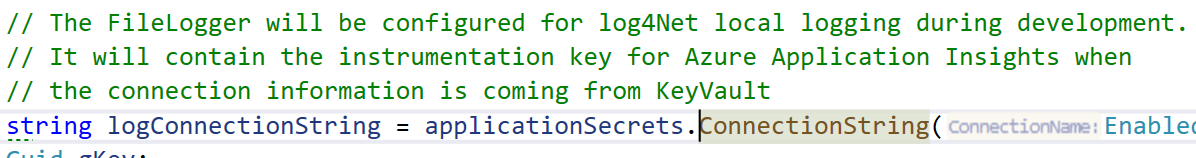
'bgrs-reference-app-kv'

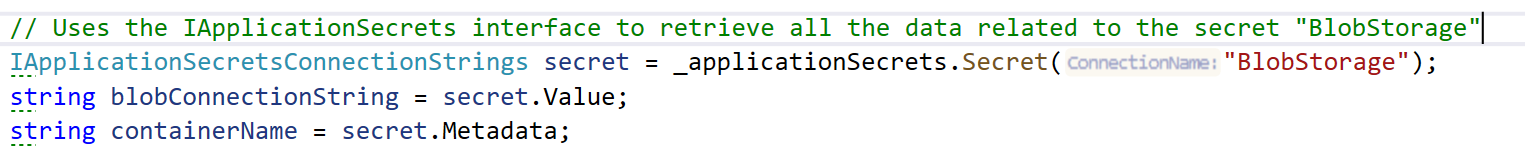
' bgrs-reference-app-secrets'



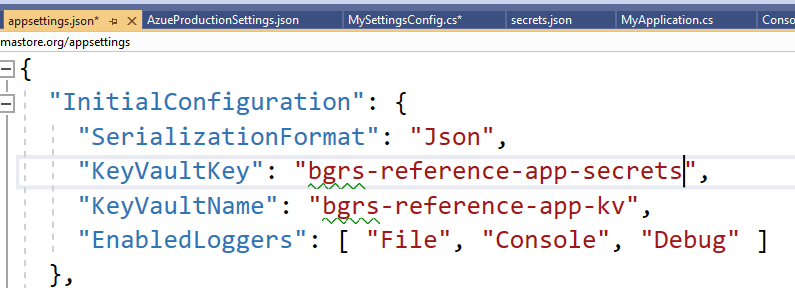
The following code uses the key vault key to pull the secret information from KeyVault, decode the base64 secrets value and update the appsettings to contain these secrets.

You access all of your secrets using the IApplicationSecrets interface that is provided to you via dependency injection. For example:



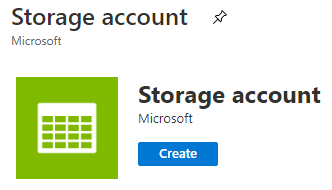


Now that you’ve created a KeyVault, created a named key and installed a secret into the key, you need to tell the application the information it needs to connect to your vault and extract the secrets from the key. You put those values into the “InitialConfiguration” section of appsettings.json. The values for KeyVaultKey and KeyVaultName are only for reference. You must enter the values that you created for your vault.



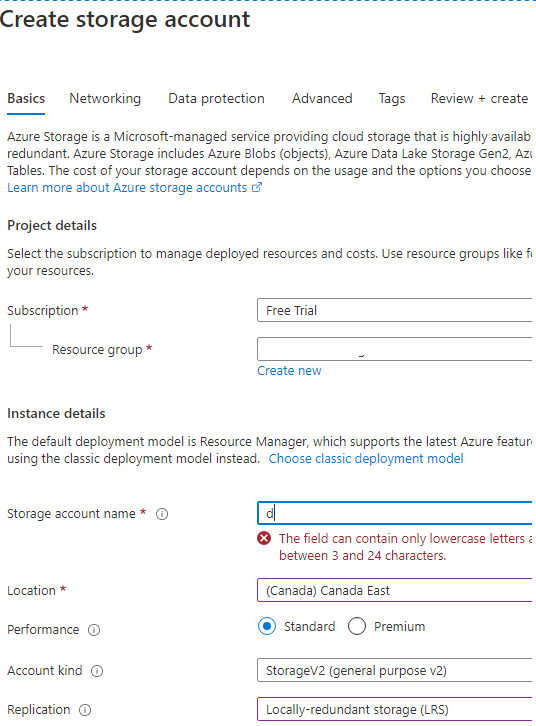
**Setting up your storage account**

Click on the resource group you’ve just created and then click on the  icon in order to add a new resource into the resource group. You’re now taken to the marketplace where you can pick the type of resource you want to add. Type ”Storage Account” in the search window and press enter.

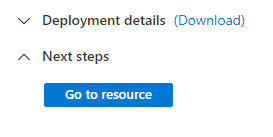


Click the create button. Now select the same resource group name as before and enter a unique name for the storage account. To keep things consistent, I’d recommend something like this:

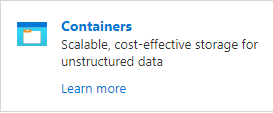
**bgrsreferenceappsa**

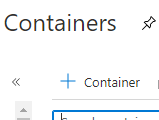


Click the  button to review your information. If you’re happy with what you see, you click . It takes about 30 seconds for the resource to be created and deployed. Just like when you created the key vault, you can download details of the deployment and also record information you’ll need to know about your storage account in order to use it in your application:



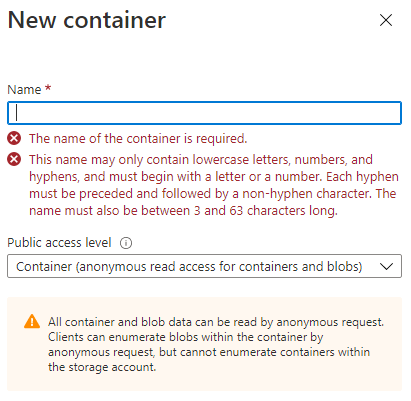
Click on “Go to resource” because now that you have a storage account, you’ll need a container in that account where your files will be stored. Select the “Containers” icon:





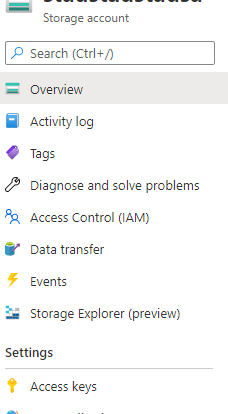
Now click on the “+ Container” to create a new container. Keeping the naming consistent, I’d recommend something like this. The –sc suffix indicates “storage container”.

**bgrs-reference-app-sc**



Click “Create” to create the container. You’ll be presented with a list of all your containers. Remember the name of your container as you’ll need it in order to store files there. Next, select the storage account (not the container, but the storage account **bgrsreferenceappsa).**

Select the “Access Keys” setting menu to get the keys you’ll need to access your storage account.



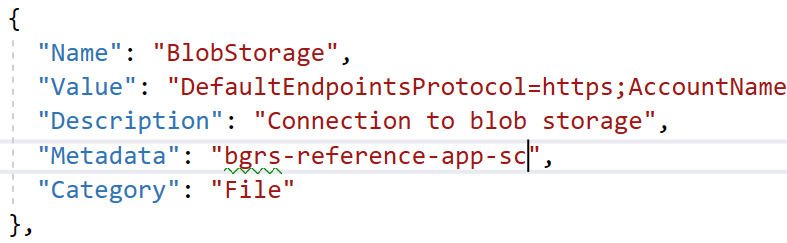
Click on show keys so you can see them in plain text:



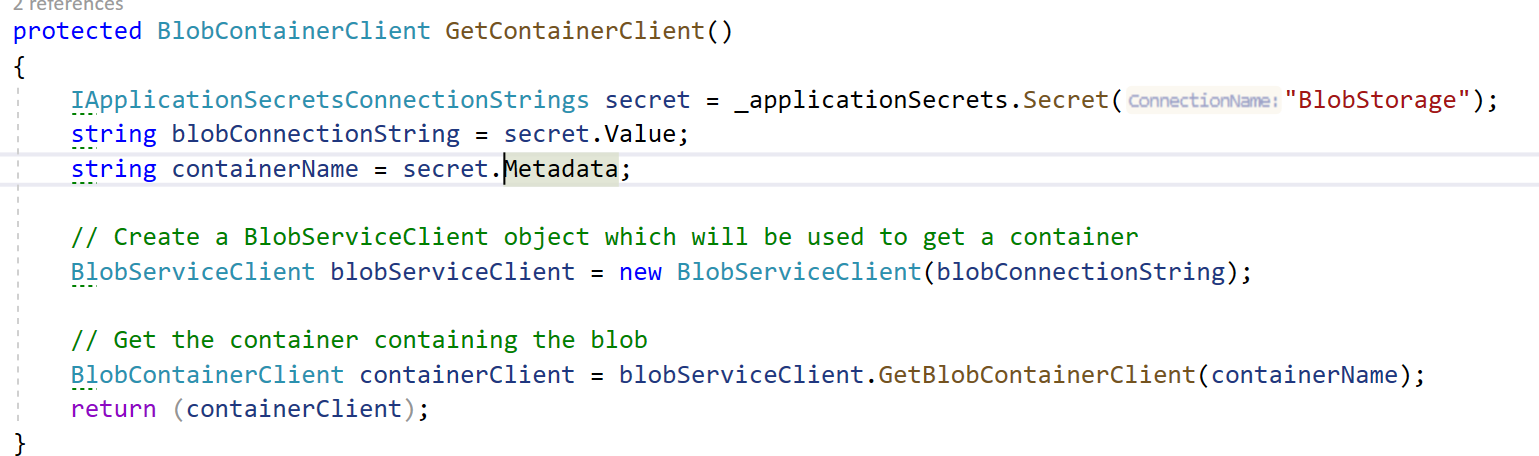
Copy the connection string because you’ll need it to connect to your storage account from the application:



This connection string will be placed into the “value” property of your “BlobStorage” app setting. The name of the storage container you’ll be storing files into is placed into the “Metadata” property.



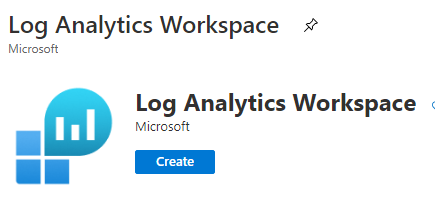
The code in the reference app that uses these secrets is presented here:



**Setting up application insights**

Azure Monitor Log Analytics you can easily store, retain and query data collected from your monitored resources in Azure and other environments for valuable insights.  
A Log Analytics workspace is the logical storage unit where your log data is collected and stored.  
A Log Analytics workspace is the basic management unit of Log Analytics.

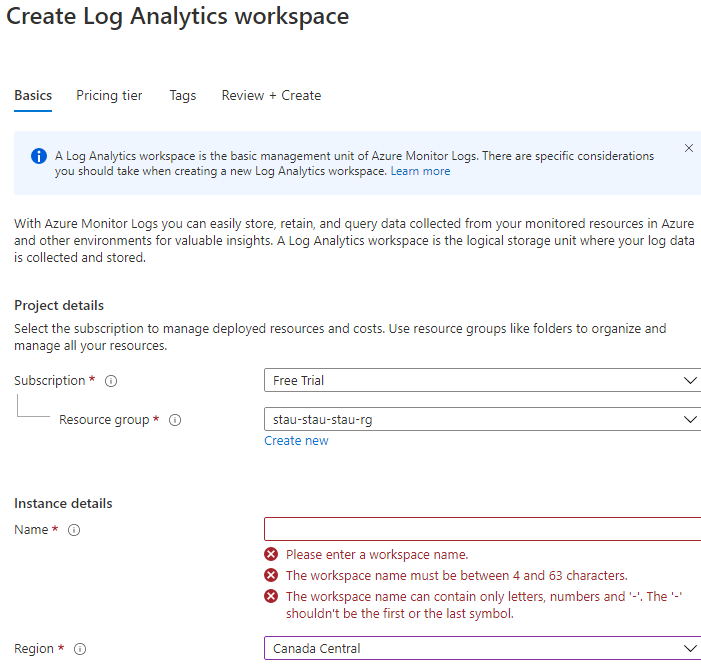
Click on the resource group you’ve just created and then click on the  icon in order to add a new resource into the resource group. You’re now taken to the marketplace where you can pick the type of resource you want to add. Type ”Log Analytics Workspace” in the search window and press enter.



Click the create button. Now select the same resource group name as before and enter a unique name for the workspace. To keep things consistent, I’d recommend something like this:

(law = Log Analytics Workspace)

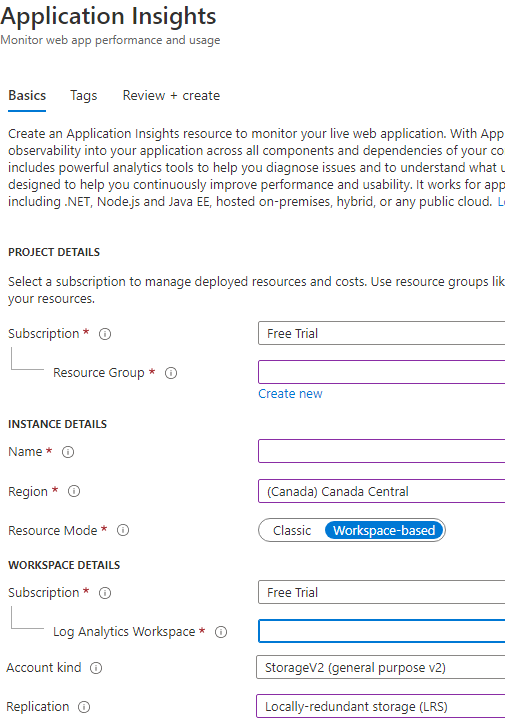
**bgrs-reference-app-law**



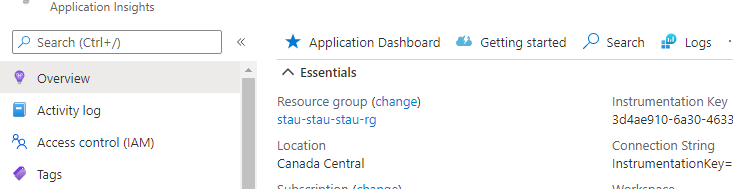
Now that you’ve created a workspace, you need to attach it to an application insights resource. Application Insights uses the workspace you previously created as an area where to store information it receives.

Click on the resource group you’ve just created and then click on the  icon in order to add a new resource into the resource group. You’re now taken to the marketplace where you can pick the type of resource you want to add. Type ”Application Insights” in the search window and press enter. Enter your resource group, the log analytics workspace name you just created, and provide a name for the application insights resource. Again, keeping the naming consisten, the name should be something like:

**bgrs-reference-app-ai**

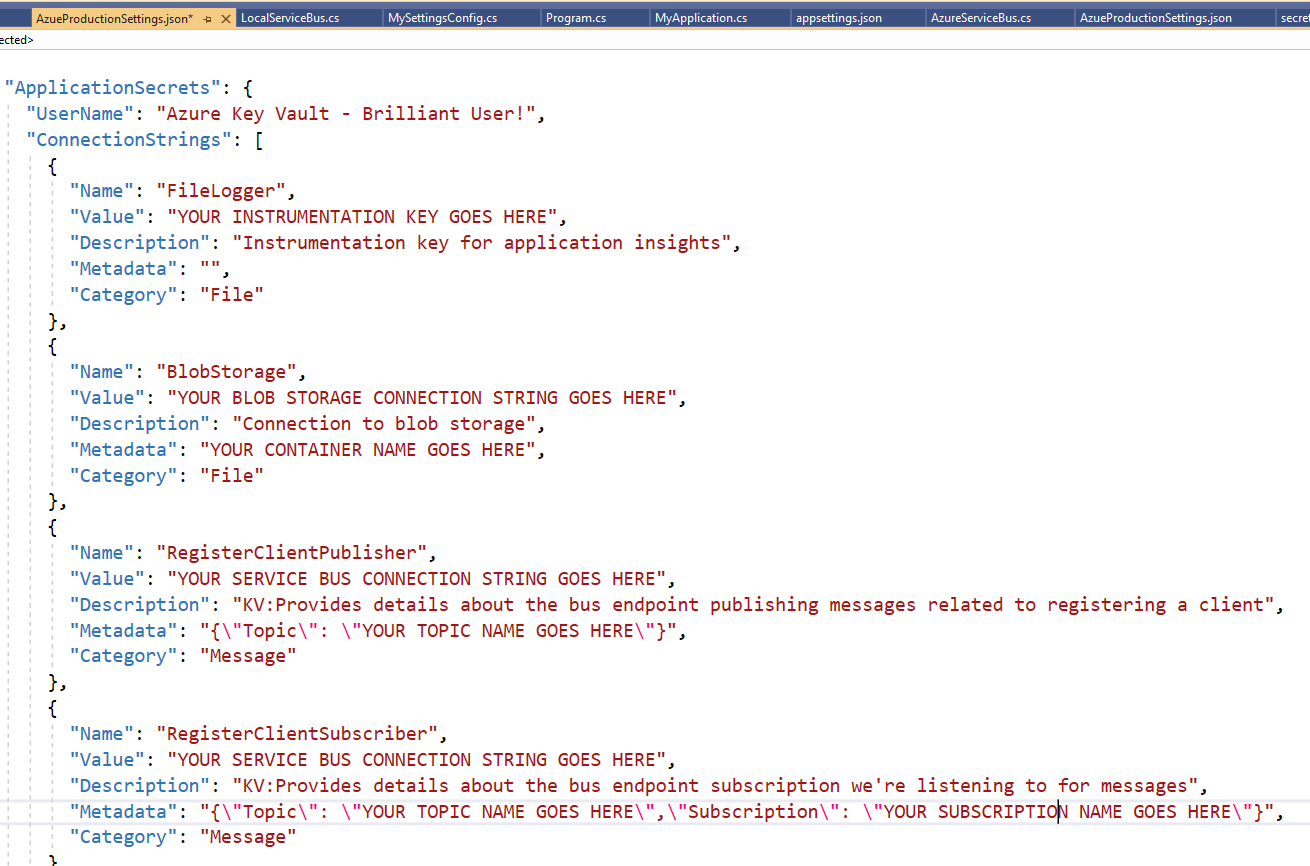


In order to use Application Insights from your application you need the “Instrumentation Key” assigned to this resource. Select the Application Insights resource you’ve just created and you’ll notice the “Instrumentation Key” appears in the top right corner.

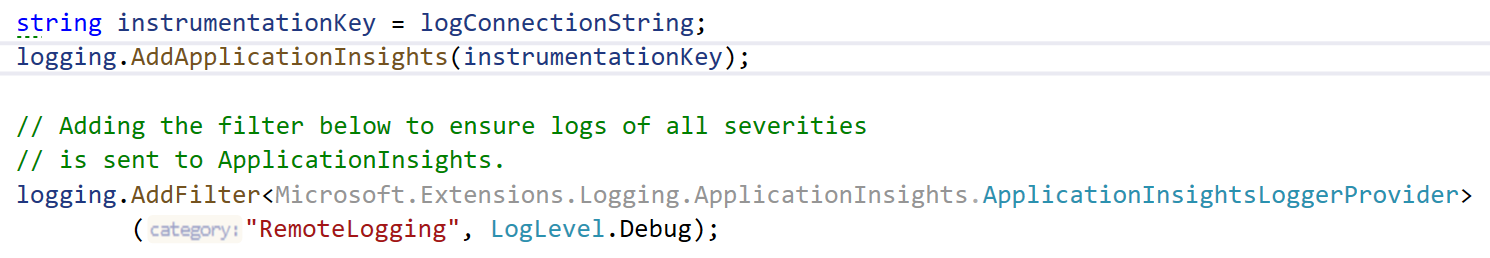
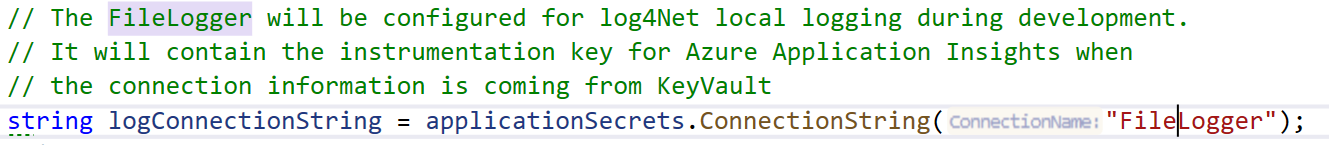


Copy the instrumentation key and insert it into the “Value” property of the “**FileLogger**” secret in the “AzueProductionSettings.json file”

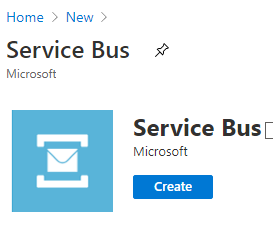
Your file should now look something like this. The values in the screenshot are dummy values to show an example. You MUST enter the values you saved when you were creating your Azure resources.



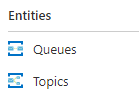
Here’s how the “FileLogger” information is used to connect to Application Insights:



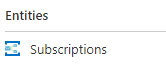
The next resource you need is a Service Bus



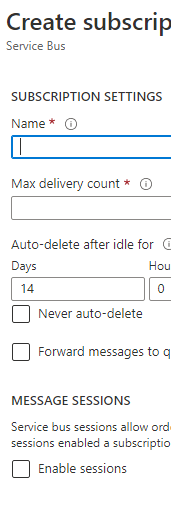
Once you’ve created the service bus namespace, you create a topic and a subscription for that topic. Click on Topic to start. Menu is on the left.



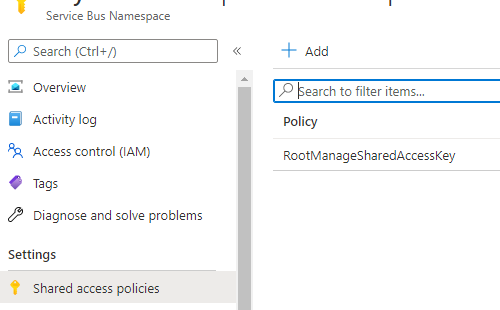
Once you’ve created the topic, create the subscript for the topic.



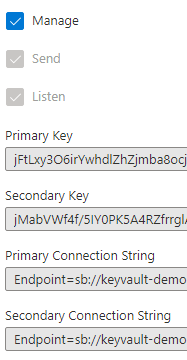
Enter the name of your subscription name, a max delivery count of 1 and DO NOT check “Enable Sessions”



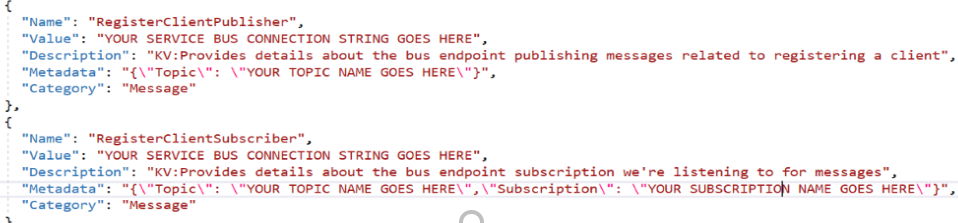
Finally, click on the Service Bus namespace and click “Shared access policies” and then select “RootManageSharedAccessKey”



A window will appear on the right. Copy the primary connection string value. You’ll need this to connect to the service bus:

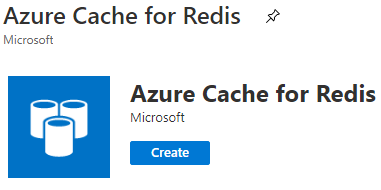


Insert this connection string in the “Value” property for both publisher and subscriber. Then insert your topic name and subscription name where indicated in the “Metadata” property. I have chosen to use JSON format for the metadata string. In order to do that, then double quotes must be escaped \”. I used JSON so that I could insert multiple values as metadata. If you prefer to use a comma separated list or semi-colons, it’s up to you. Just make sure you adjust the code that uses the metadata property accordingly.



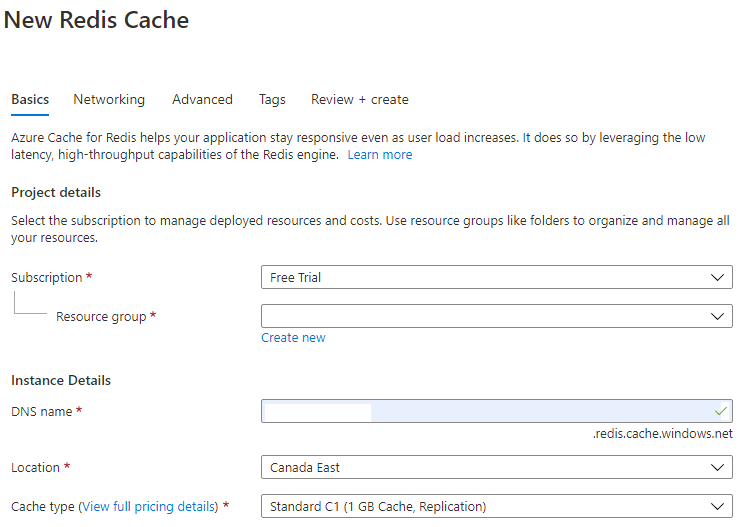
The final resource you’ll need is the Redis Cache

Go to your resource group and add a new resource of type “Azure Cache for Redis”

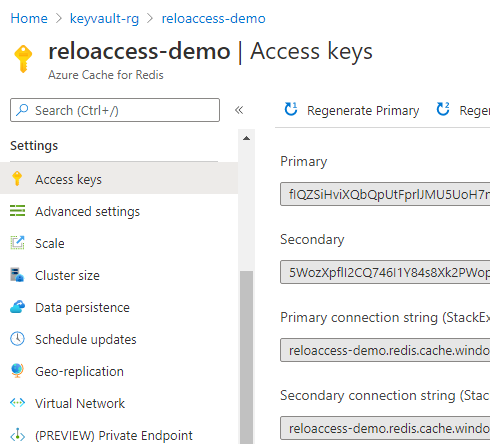


This one is pretty easy. Simply select your resource group and type in the DNS name you want to use to access the cache. To keep naming consistent, I suggest you use (rc = Redis Cache):

**bgrs-reference-app-rc**



After a minute or so the cache will be provisioned. You now have to get its connection string so that your application can connect to the cache. Click on the name of the cache you’ve just and select Settings/Access Keys to get the connection string:

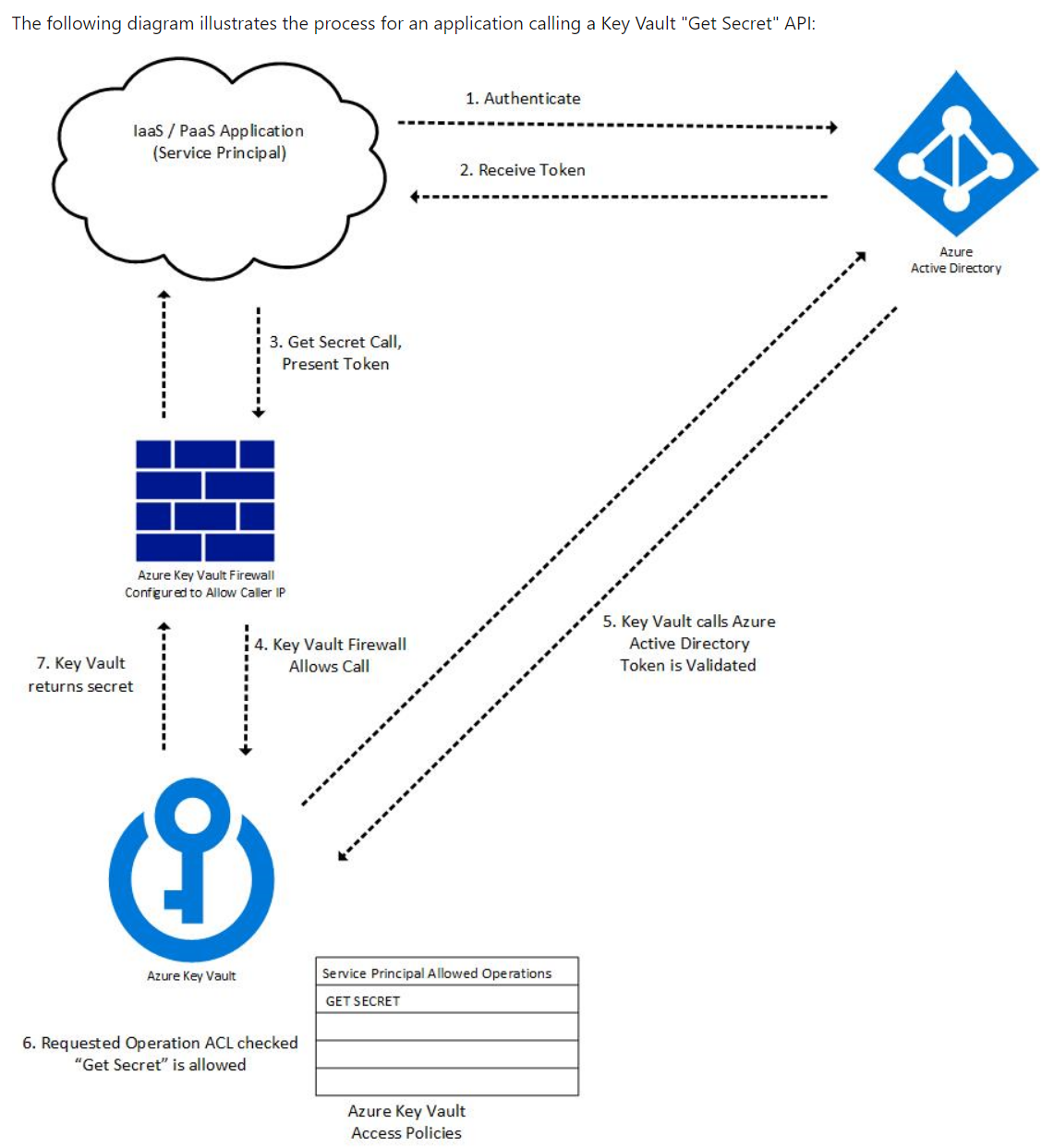


The first thing the application needs to do is authenticate with Azure KeyVault. Without that, we can’t access the secrets in our vault and demonstrate a secure way to configure the application.

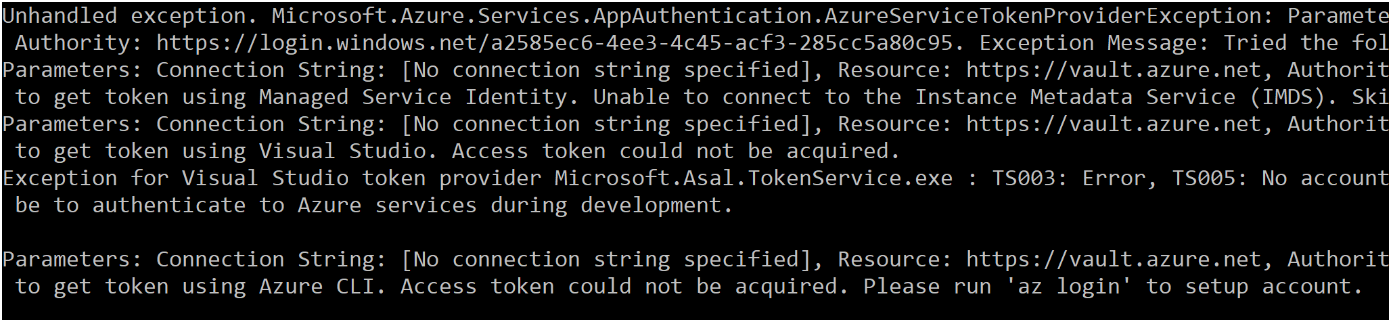
<https://docs.microsoft.com/en-us/azure/key-vault/general/authentication>

## **The Key Vault authentication flow**

1. A service principal requests to authenticate with Azure AD, for example:
   * A user logs into the Azure portal using a username and password.
   * An application invokes an Azure REST API, presenting a client ID and secret or a client certificate.
   * An Azure resource such as a virtual machine with a managed identity contacts the [Azure Instance Metadata Service (IMDS)](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/instance-metadata-service) REST endpoint to get an access token.
2. If authentication with Azure AD is successful, the service principal is granted an OAuth token.
3. The service principal makes a call to the Key Vault REST API through the Key Vault's endpoint (URI).
4. Key Vault Firewall checks the following criteria. If any criterion is met, the call is allowed. Otherwise the call is blocked and a forbidden response is returned.
   * The firewall is disabled and the public endpoint of Key Vault is reachable from the public internet.
   * The caller is a [Key Vault Trusted Service](https://docs.microsoft.com/en-us/azure/key-vault/general/overview-vnet-service-endpoints#trusted-services), allowing it to bypass the firewall.
   * The caller is listed in the firewall by IP address, virtual network, or service endpoint.
   * The caller can reach Key Vault over a configured private link connection.
5. If the firewall allows the call, Key Vault calls Azure AD to validate the service principal’s access token.
6. Key Vault checks if the service principal has the necessary access policy for the requested operation. If not, Key Vault returns a forbidden response.
7. Key Vault carries out the requested operation and returns the result.

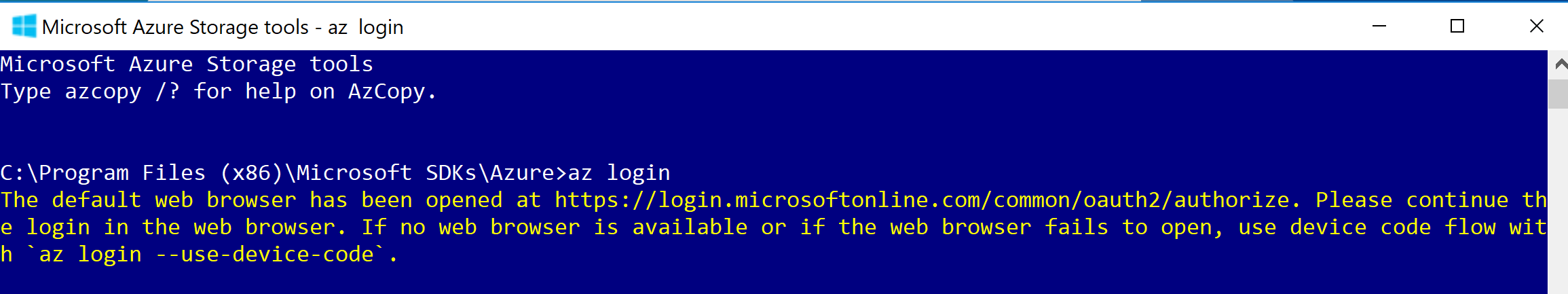


If you have not yet authenticated with Azure and try to run the application, you’ll get an error screen similar to this:

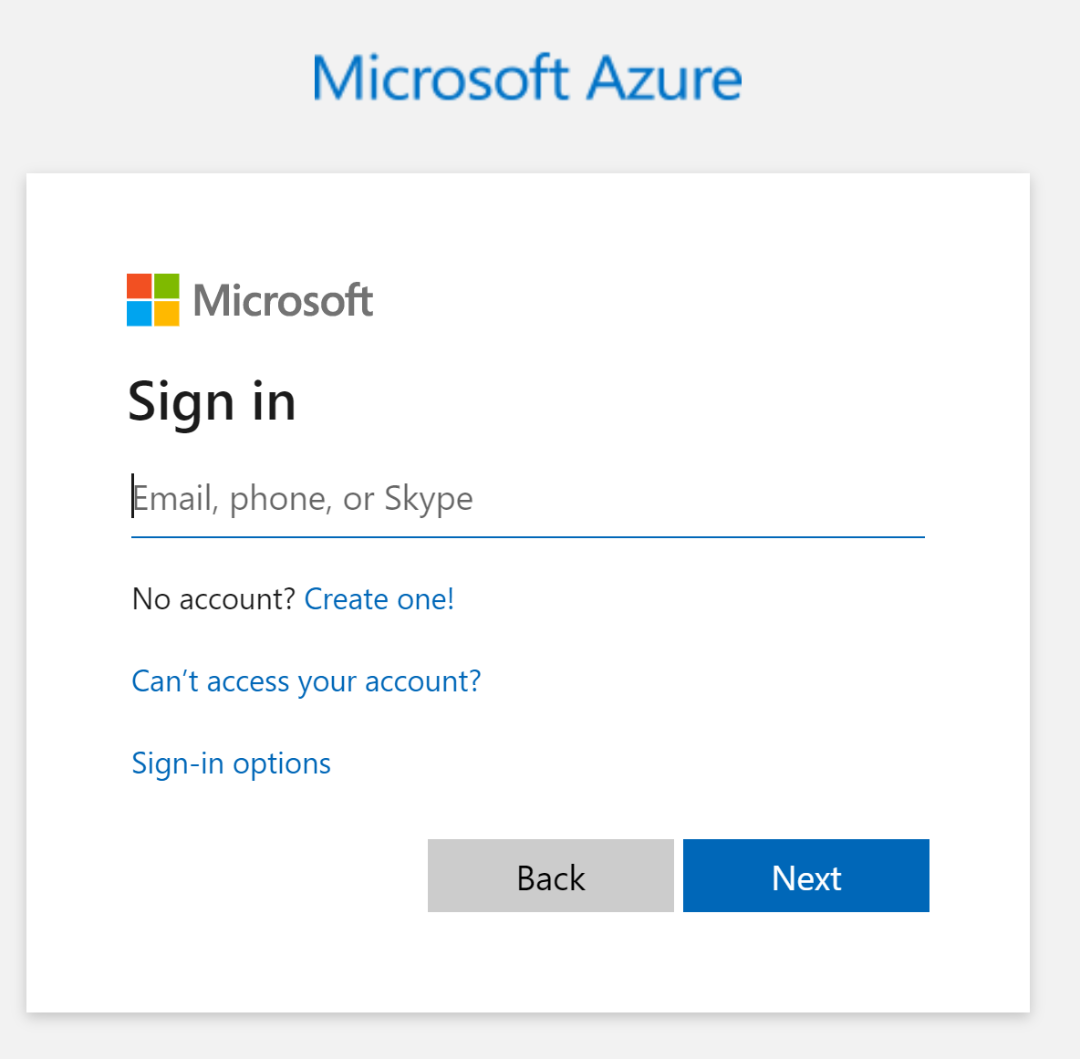


This saying that you were not authenticated using various different methods it tried, so you’re denied access. So, open up the Azure CLI and enter “az login’

If you have not installed the Azure CLI, you can get it here: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>



Your browser will open to allow you to authenticate on Azure. Once you’ve authenticated, you can run the demo.



You can also authenticate through Visual Studio tools/options:

