**Secure Configuration**

This document will guide you in the setup of storing your application configuration information securely in Azure KeyVault. Although you can perform all the necessary steps using the Azure Portal [UI](https://portal.azure.com/#home) , I find it faster and simpler to use powershell scripts. So, ensure you have the latest version of [powershell](https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-windows?view=powershell-7.1) installed.

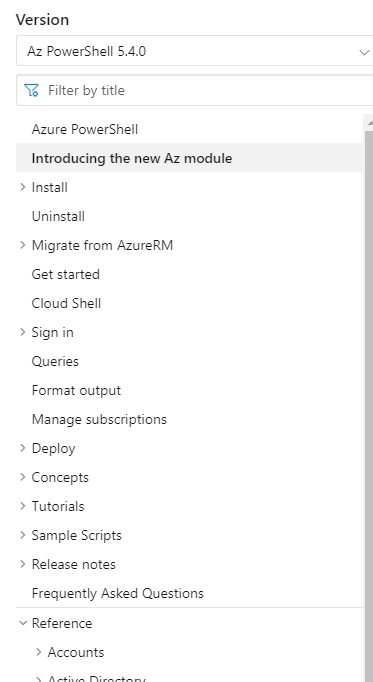
Once powershell is installed, run the following command to ensure you have the latest az module installed. This will allow you to execute scripts that communicate with Azure.

Update-module az

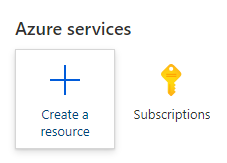
Sign into Azure using the [following](https://docs.microsoft.com/en-us/powershell/azure/authenticate-azureps?view=azps-5.4.0) command:

Connect-AzAccount

Powershell commands to help administer Azure KeyVault are listed below. For a comprehensive list of commands, please refer to [Key Vault](https://docs.microsoft.com/en-us/powershell/module/az.keyvault/?view=azps-5.4.0#key-vault) . Azure powershell commands can be found [here](https://docs.microsoft.com/en-us/powershell/azure/new-azureps-module-az?view=azps-5.4.0) . It’s not obvious where to find the list of all powershell commands. You have to select the “Reference” menu option near the bottom of the entire menu and then select the category of command you’re looking for.



To list your Azure subscriptions you can find this in the Azure Portal.



To do this from powershell:

Get-AzSubscription

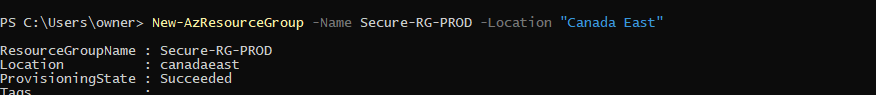
Before you can create a resource, you need to create a resource group to put it in.

The **Get-AzResourceGroup** cmdlet gets Azure resource groups in the current subscription. You can get all resource groups, or specify a resource group by name or by other properties. By default, this cmdlet gets all resource groups in the current subscription.

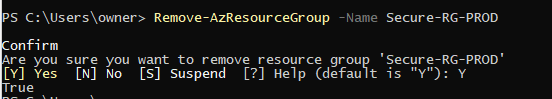
If you do not yet have any resource groups, then you can create one.

The **New-AzResourceGroup** cmdlet creates an Azure resource group. You can create a resource group by using just a name and location, and then use the New-AzResource cmdlet to create resources to add to the resource group. To add a deployment to an existing resource group, use the New-AzResourceGroupDeployment cmdlet. To add a resource to an existing resource group, use the **New-AzResource** cmdlet. An Azure resource is a user-managed Azure entity, such as a database server, database, or website. An Azure resource group is a collection of Azure resources that are deployed as a unit.

New-AzResourceGroup -Name Secure-RG-PROD -Location "Canada East"



The **Remove-AzResourceGroup** cmdlet removes an Azure resource group and its resources from the current subscription. To delete a resource, but leave the resource group, use the Remove-AzResource cmdlet. When you delete a resource group, ALL the resources in that group are automatically deleted too. This is a handy way to creating a bunch of resources for testing and then delete them all with one command.



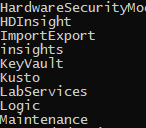
If you know the location where you want to create your vault, then go ahead and enter it. If you need a list of available locations, you can use the following command:

Get-AzLocation

Not every resource is available in every location. To find out what is available in ‘Canada East’ (the closest Azure data center), use the command:

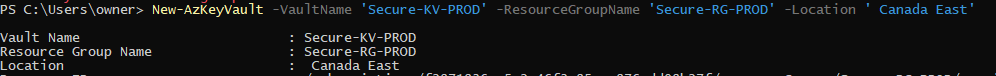
(Get-AzLocation | Where-Object {$\_.DisplayName -eq "Canada East"} | Select Providers).Providers| ForEach-Object {$\_.Substring($\_.IndexOf('.')+1)} | Sort-Object

You’ll see from this list that the resource that you want (KeyVault) is available in the selected location. If the resource you want is not in the list, then you have to find a different location where it is available.



The **Get-AzKeyVault** cmdlet gets information about the key vaults in a subscription.

The **New-AzKeyVault** cmdlet creates a key vault in the specified resource group. This cmdlet also grants permissions to the currently logged on user to add, remove, or list keys and secrets in the key vault.

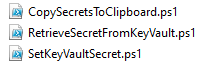


Now that you have a new key vault, you need to create one or more keys to put your secrets into.

The **Set-AzKeyVaultSecret** cmdlet creates or updates a secret in a key vault in Azure Key Vault. If the secret does not exist, this cmdlet creates it. If the secret already exists, this cmdlet creates a new version of that secret.

Although multi-line key values are supported by KeyVault, the Azure portal UI does not provide 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, I have provided 3 powershell scripts to create and retrieve KeyVault secrets.

:



* **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 creates/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.

SetKeyVaultSecret.ps1 will create a key in your keyvault if it does not exist yet, and will update the secret value if it does exist.

In this example we will create a new secret called ‘Secure-KV-PROD-SECRETS' . It will encode the contents of the file X:\...\...\YourSecrets.json into Base64 and then place that encoded value into the key ‘Secure-KV-PROD-SECRETS’ in the vault named ‘Secure-KV-PROD'.

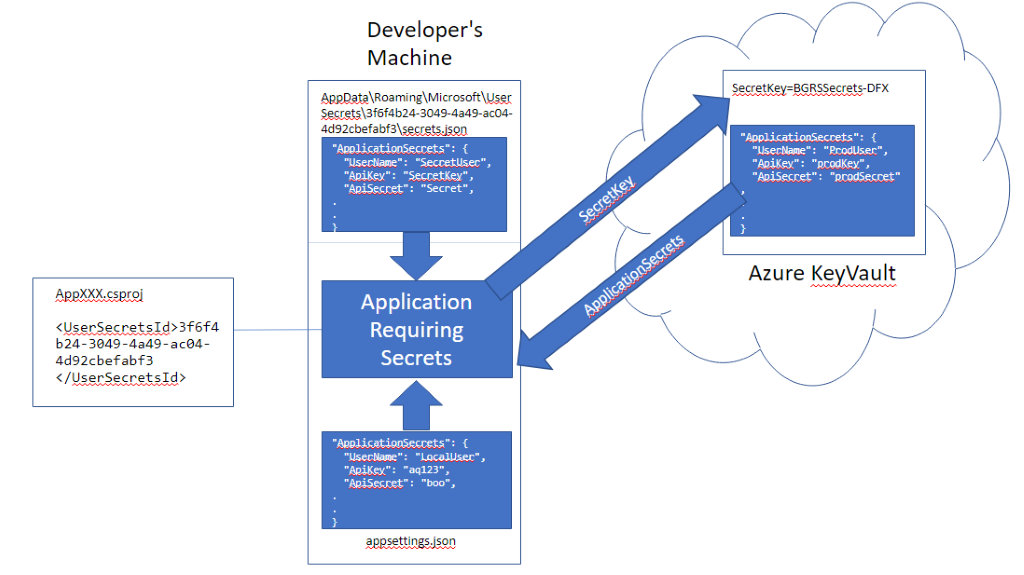


To retrieve the contents of the key, use the command:

.\RetrieveSecretFromKeyVault.ps1 'Secure-KV-PROD' 'Secure-KV-PROD-SECRETS'

While the contents of ‘YourSecrets.json’ file can contain ANYTHING you want, in any format you want, I recommend that you use the code provided that has been designed to make storing and retrieving secret data easy.

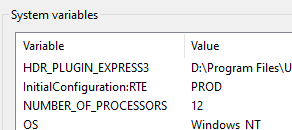
A reference application [SecureConfigOnly](https://github.com/stauent/SecureConfigOnly.git) can be downloaded from GitHub. It contains all the code that will show you how to use “User Secrets” as well as “Azure KeyVault”.



The appsettings.json file contains the KeyVault name and secret key name as described above. The key vault values contain substitution markers “{RTE}” which stands for “Run Time Environment”. This value is set in your “InitialConfiguration” section so that you can test out getting key vault information in different runtime environments. If the property 'InitialConfiguration:RTE’ is not set, then the value will be retrieved from an environment variable with that name. Note, that the environment variable name MUST BE ‘InitialConfiguration:RTE’ because it must match the path name that would be used to retrieve the property from the configuration file. The colon is how you specify a nested/child property name. This allows you to set this environment variable differently on each server. You don’t have to worry about changing this value in your configuration file and deploying a different one to each server. If you don’t have the property set in either location, then the value “DEFAULT” will be used. You can always hard code the key vault properties as well. You don’t have to use the {RTE} substitution of you don’t want to. You might have a need to test out some specific property hard coded values for some reason.

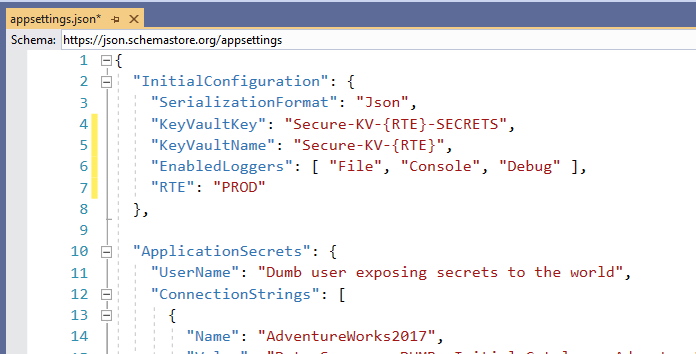
The following powershell command will allow you to set the environment variable:

[System.Environment]::SetEnvironmentVariable('InitialConfiguration:RTE','PROD',[System.EnvironmentVariableTarget]::Machine)



To delete the environment variable, simply set the value to $null

[System.Environment]::SetEnvironmentVariable('InitialConfiguration:RTE',$null,[System.EnvironmentVariableTarget]::Machine)



With the environment variable set to ‘PROD’, the keyvault values become:

Secure-KV-PROD-SECRETS and Secure-KV-PROD

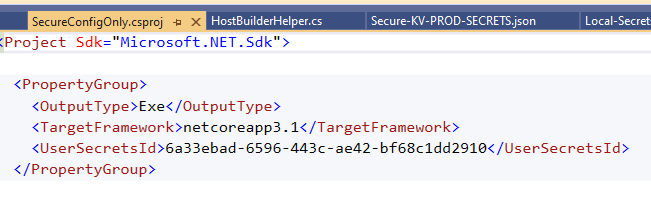
NOTE: ***Visual Studio caches all environment variables when it is launched. If you set an environment variable WHILE Visual Studio is still running, your code WILL NOT see that change. You must shut down visual studio and restart for your code to see any change in environment variables.***

A file called “Local-Secrets.json” contains some data that you must copy and place into your secrets.json file. Right click on the SecureConfigOnly.csproj file and select “Manage User Secrets”. You’ll be presented with an empty secrets.json file. Copy the contents of Local-Secrets.json into secrets.json and save it. Once this is done you can destroy the Local-Secrets.json file. It’s only there to save you some typing when you run your demo.

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”.



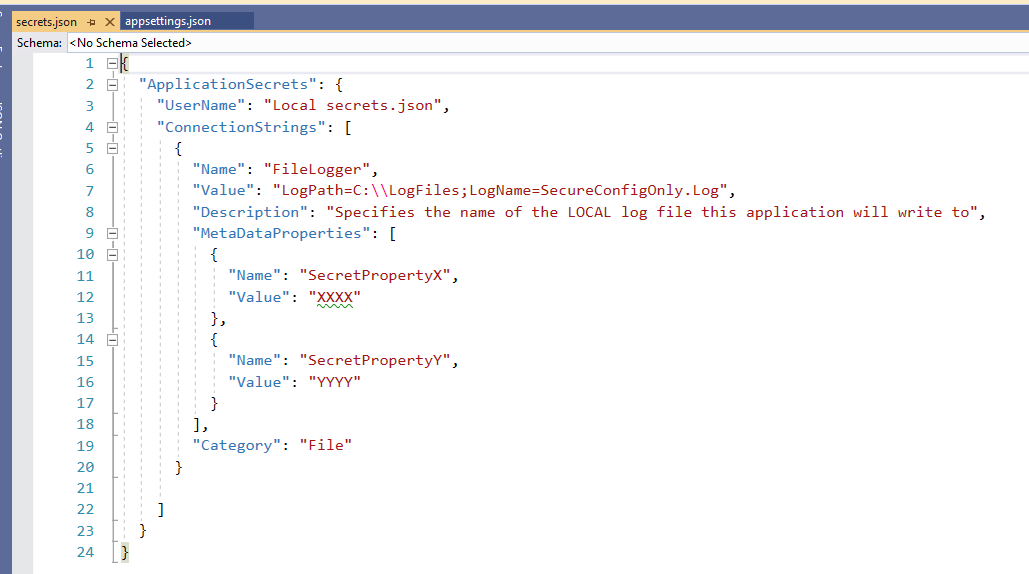
If your application was not set up to support user secrets yet, Visual Studio will inject a “UserSecretsId” section into the .csproj file:



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\6a33ebad-6596-443c-ae42-bf68c1dd2910

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.



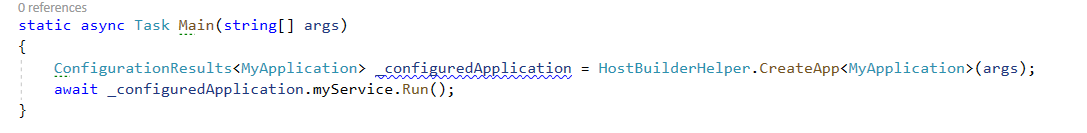
A file called “Secure-KV-PROD-SECRETS.json” contains data that will be placed into your KeyVault (Secure-KV-PROD) in the key “Secure-KV-PROD-SECRETS”. You will store the secret using the powershell command:



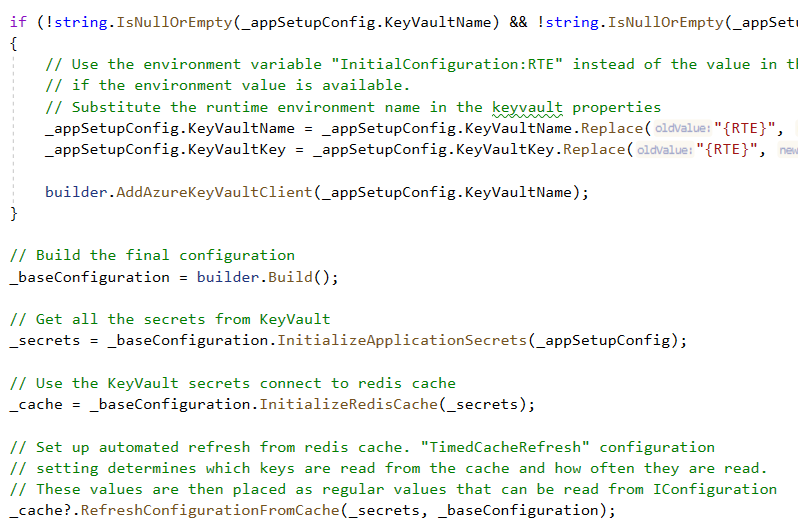
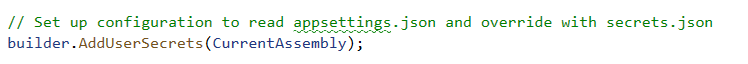
You must change the first parameter to specify the location where you installed the demo code from GitHub. Check to make sure that the secret got stored by retrieving it:

I suggest that you create a separate .json file for each environment and then call the SetKeyVaultSecret.ps1 powershell script to store your secrets for that environment.

If that works, then simply run the application. The following code configures the host to run the application. Although it’s set up for a console app, most of the code can easily be copied into a Startup.cs class in your web applications. It sets up the IOC container and initializes the configuration so that appsettings.json is loaded, then secrets.json and finally the configuration from the KeyVault. The result will be a merged configuration of the three files. Each configuration overrides the previous one so that the values in KeyVault will always be the last one used, if it exists.



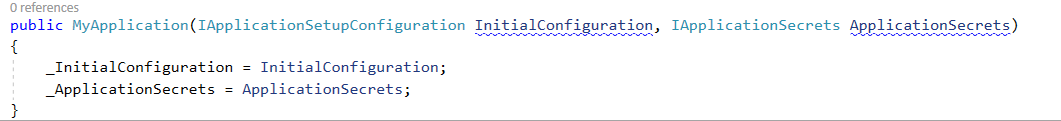
The code that sets this up is displayed here:



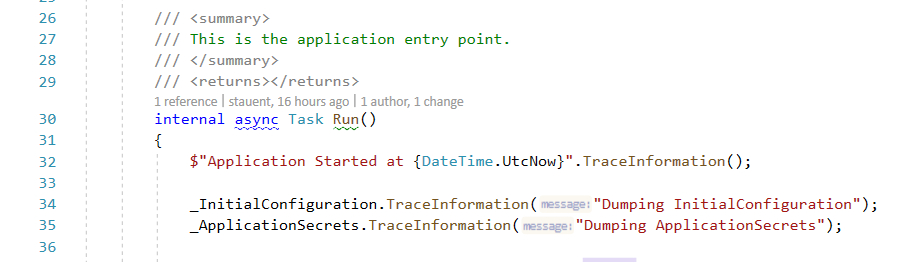
The “Run” command in main indicates that a method called “Run” on a class called “MyApplication” will execute to run the application. When the “Run” method completes, the application ends.

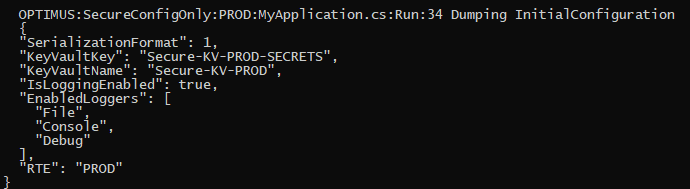
\_configuredApplication.myService.Run();

The IOC container initializes the MyApplication constructor passing in the interfaces that contain our configuration information:



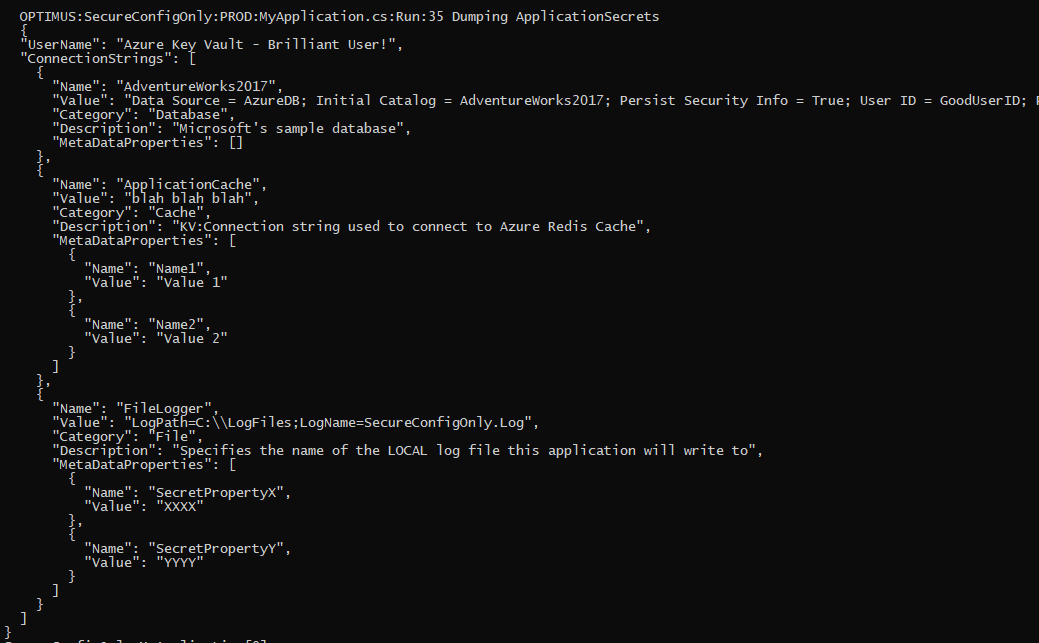
In the “Run” method, I simply show you how to access the configuration data.

An extension method that provides a very simple way to log data is provided. Simply use .TraceXXXX on ANY object to log that data.

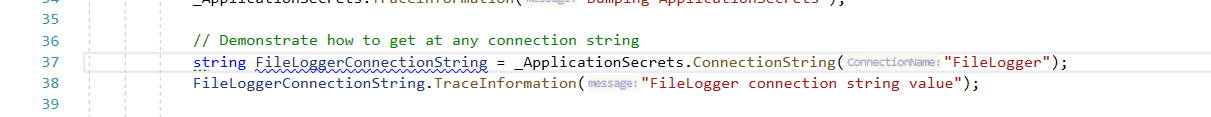


You’ll notice that a heading is displayed before the data being logged. That header contains the name of the server the code is running on, the name of the application, the environment, the name of the class, method and line number that was executed. This makes it easy to find the location where you logged the information from. If multiple co-operating applications or services are logging to a central location like “Application Insights”, then this makes it simple to determine the timeline of which service or application execute at what time and from which server.

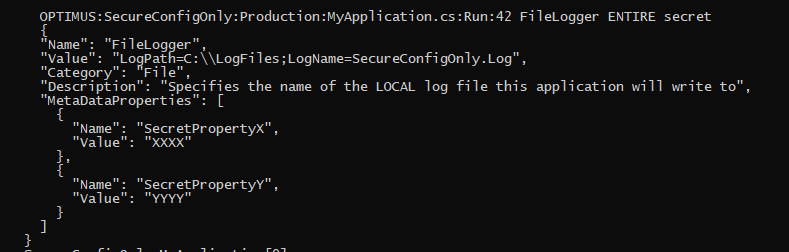
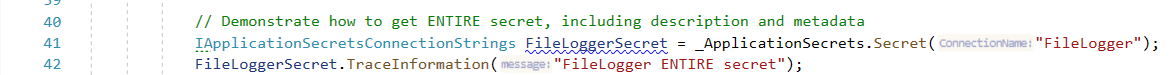
What you’ll notice when you log the application secrets is that the result is the combination of data from appsettings.json, secrets.json and the values you put into KeyVault:



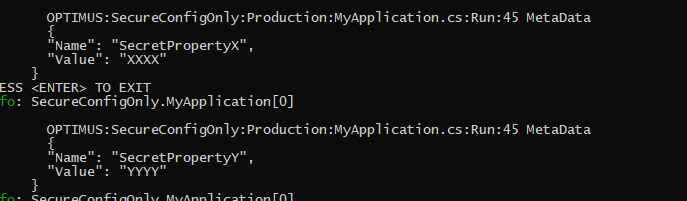
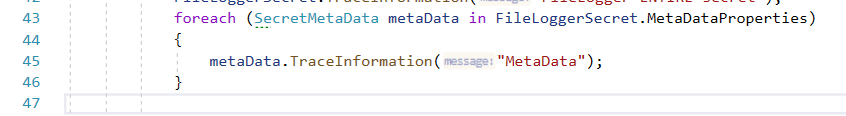
Next, I demonstrate how to get any specific secret connection string:



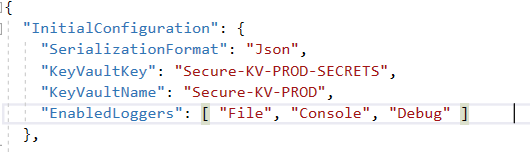
If you need to get at all the properties of the secret, that’s simple too:



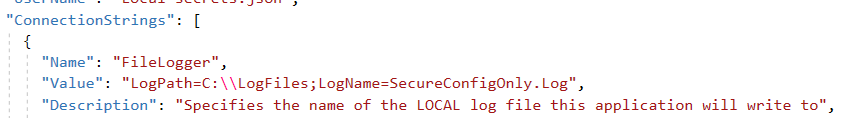
And finally, if you need to use the metadata associated with the secret, that’s simple too.

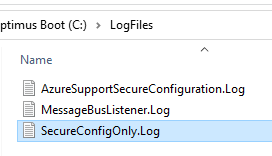


Last thing to mention about the logger is that you can configure where the log output goes. The “EnabledLoggers” property dictates this:

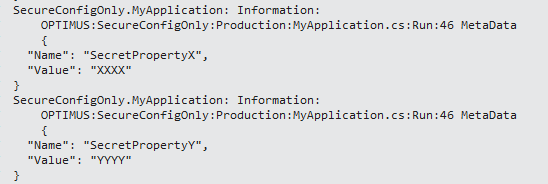


If “File” is specified then the “FileLogger” configuration setting is used to specify the location and name of the log file. In this case the value is:

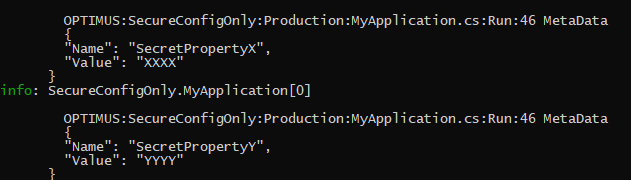




If “Debug” is specified then the log data is displayed in the visual studio output window:



If “Console” is specified then the data is displayed in the console window:

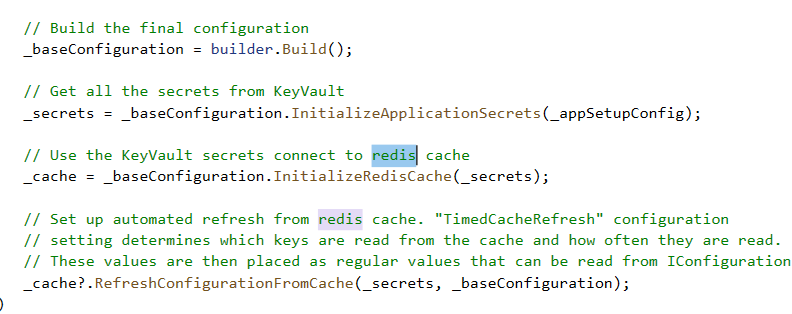
ef

If “None” is specified or the EnabledLoggers property is empty, then no logging takes place.

**Redis Cache**

One of the things that we’ll need from time to time is the ability to read values from Redis Cache. We currently have a Function app running in Azure that is constantly renewing a JWT value for LMC. This function reads the current JWT value from the cache key “ONIT\_JWT”, uses that value to renew the JWT, then writes the renewed JWT back into the cache. Our

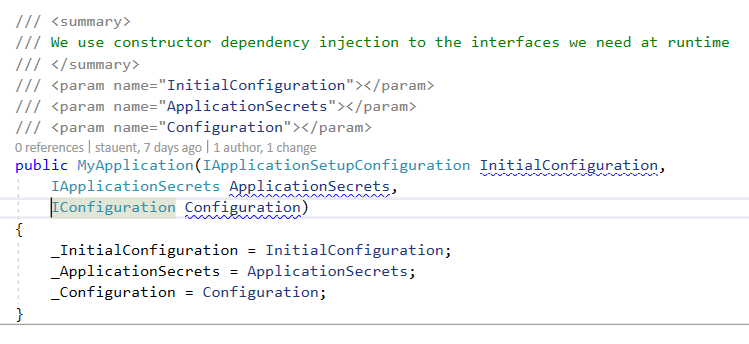
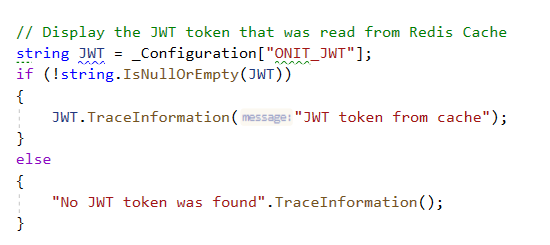
As part of application configuration, I instantiate a redis cache client that is used to read ANY key from the cache. Here’s the code that is required during configuration:



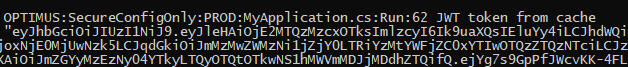
Here’s the part of the configuration that tells allows us to connect to the cache. The “ApplicationCache” value specifies the connection string required to connect to redis cache. The “TimedCacheRefresh” setting tells our code which values to pull from redis cache and how often to do the pull. The “Value” only specifies one cache key “ONIT\_JWT”. If we needed to pull multiple values from the cache, simply create a comma separated list. For example “ONIT\_JWT,MyAge,CertificateNumber” would read 3 different values from the cache. When these values are read from the cache, they are AUTOMATICALY added into the regular configuration properties.



Here’s how you read the redis cache values in your application. First ensure that you have access to IConfiguration. This is the interface you’ll read the cache values from.

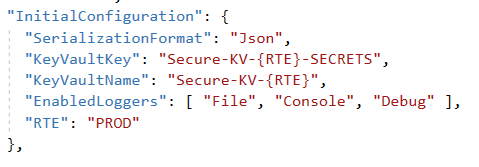
Here’s how simple it is to read the JWT that was in the cache:

Here’s the output I get on my machine:



**Authentication**

One of the things that you might be wondering is how this application gets authenticated so that it can access secrets in KeyVault. The only KeyVault settings of any kind are specified in appsettings.json. Here we indicate the name of the vault and the name of the key in the vault we want to access.

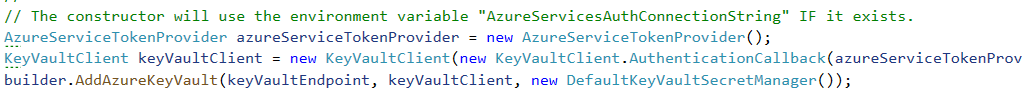


With this little information, how is it possible to securely authenticate and access KeyVault secrets?

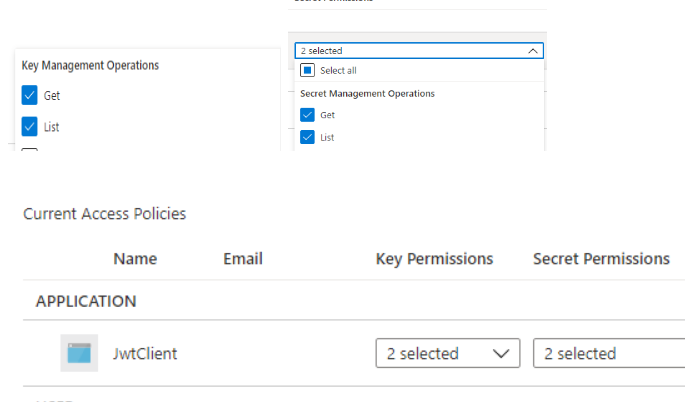
In a class called “KeVaultHelper”, the extension method AddAzureKeyVaultClient provides the “magic” that allows us to authenticate against Azure AD and get access to KeyVault. The key is the class “AzureServiceTokenProvider”. The constructor can take a “connection string” parameter which provides details about the “Service Principle” that will be used for authentication. The format of the connection string is:

**“RunAs=App;AppId=<Application ID>;TenantId=<Tenant Name>.onmicrosoft.com;AppKey=<App Secret>”**

This connection string provides details about a service principal that we want to use for authentication. But, as you can see in the code sample below, we don’t use this connection string as a constructor parameter. Read further to know why.



The idea here is that you register an application (service principal) in Azure AD, and create a secret for it. Then you go into KeyVault and add an access policy to allow this service principal to have get/list access to KeyVault keys and secrets. In the following example, I’ve registered an app called “JwtClient” and set KeyVault access policies to allow this application (Service Principal) access.



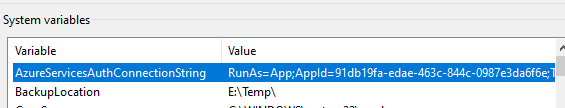
If you call the AzureServiceTokenProvider constructor with the connection string providing the details of this service principal, then you’ll get the access token you need to connect to KeyVault.

The following two sites provide a thorough explanation on how authentication works with KeyVault, but I’ll give you the short answer to save you some time.

<https://blogs.aaddevsup.xyz/2020/01/exploring-azureservicetokenprovider-class-with-azure-key-vault/>

<https://docs.microsoft.com/en-us/dotnet/api/overview/azure/service-to-service-authentication>

The correct way to use this code is NOT to pass in a connection string in the constructor, but rather use the no parameter constructor. That way, the AzureServiceTokenProvider will use the environment variable “**AzureServicesAuthConnectionString**” to authenticate with, but only if it exists. If not found, it will try Visual Studio developer credentials and Azure CLI to authenticate. So this is just fine for development environments.

I choose to create an environment variable for this purpose as demonstrated below. The configuration below works great! 

When your code is deployed to Azure, there will be no **AzureServicesAuthConnectionString** environment variable present. The AzureServiceTokenProvider code sees that this value is missing, and then uses the Managed Service Identity of the application to authenticate instead. Now there are no access keys, secrets or connection strings to manage at all. Everything is handled for you auto-magically through Azure AD.