Azure Functions

- Function concept with PowerShell
- How to Function
- Centralize log collection with a custom REST API
- Credential handling with Azure KeyVault
- Access Azure Function App via OAuth 2.0 authentication
- Create reference to Azure Key Vault content from function code
- Run PowerShell Code from frontend on backend using Azure Function
- Extend KQL Data with Azure Function as Resource Exporter
- Secure and Centralized Secret Handling with mTLS in Azure-Certificate-Secret-Proxy

Function concept with PowerShell

Azure Function Apps is a "Function as a Service" solution and offers serverless execution of highly scalable code. This function can then be executed via HTTP requests and provide corresponding feedback in the HTTP response. However, this technology only makes sense if several requests are to be executed simultaneously (e.g. all devices are to fetch or save information). See chapter "Decision Support Technology (Comparison of Variants)".

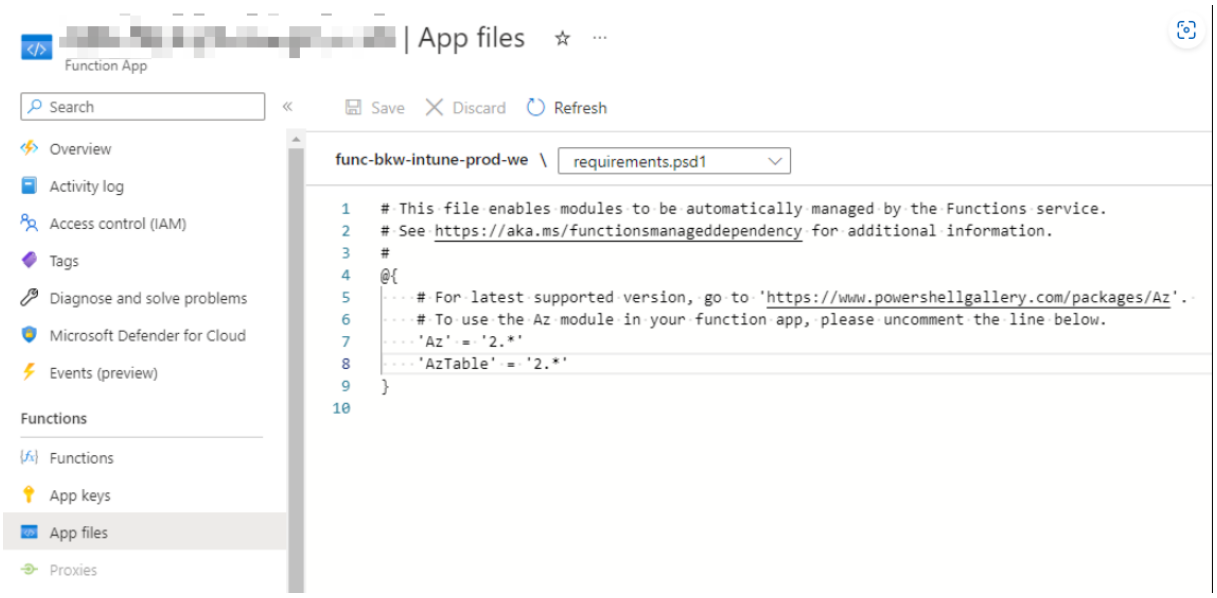
Function

The Functions are elements in the Azure Function Apps that contain the code and are then executed. Functions can have different triggers. Functions can be named according to the naming convention.

- FUNC-<Technology>-<KGTag>-<Object>-<Beschreibung>-<Status>-<Location>
  Example: FUNC-WIN-ALL-PS1-GetStorageTableContent-PROD-WE

PowerShell Modules

In order to use PowerShell modules, they must first be added to a config file in the Azure Function App. This can be achieved in the Function App via "App files" → "requirements.psd1" → Add module name and version to PowerShell list. However, adding the module to the Azure Function can take several 10 minutes.

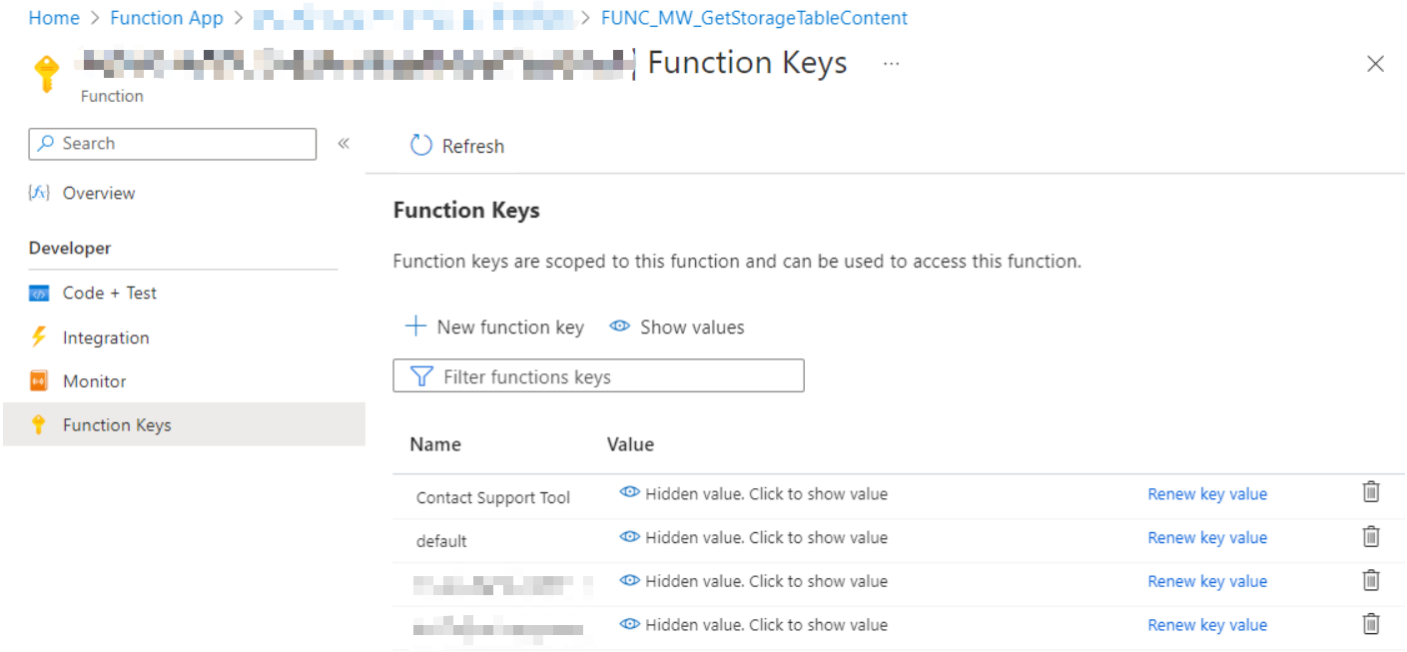


Import PowerShell Modules into scripts

```
Import-Module AzureAD
```

Authentication to Azure Function App

Authentication on the Azure function is done via the function keys, which are specified as a query. These function keys can be created on the function itself via → "Function Keys":



I don't use App Keys because I want to map multiple functions in an Azure Function App and then I wouldn't be able to use the security benefits of multiple keys.

The naming convention for Azure Function Keys is as follows:

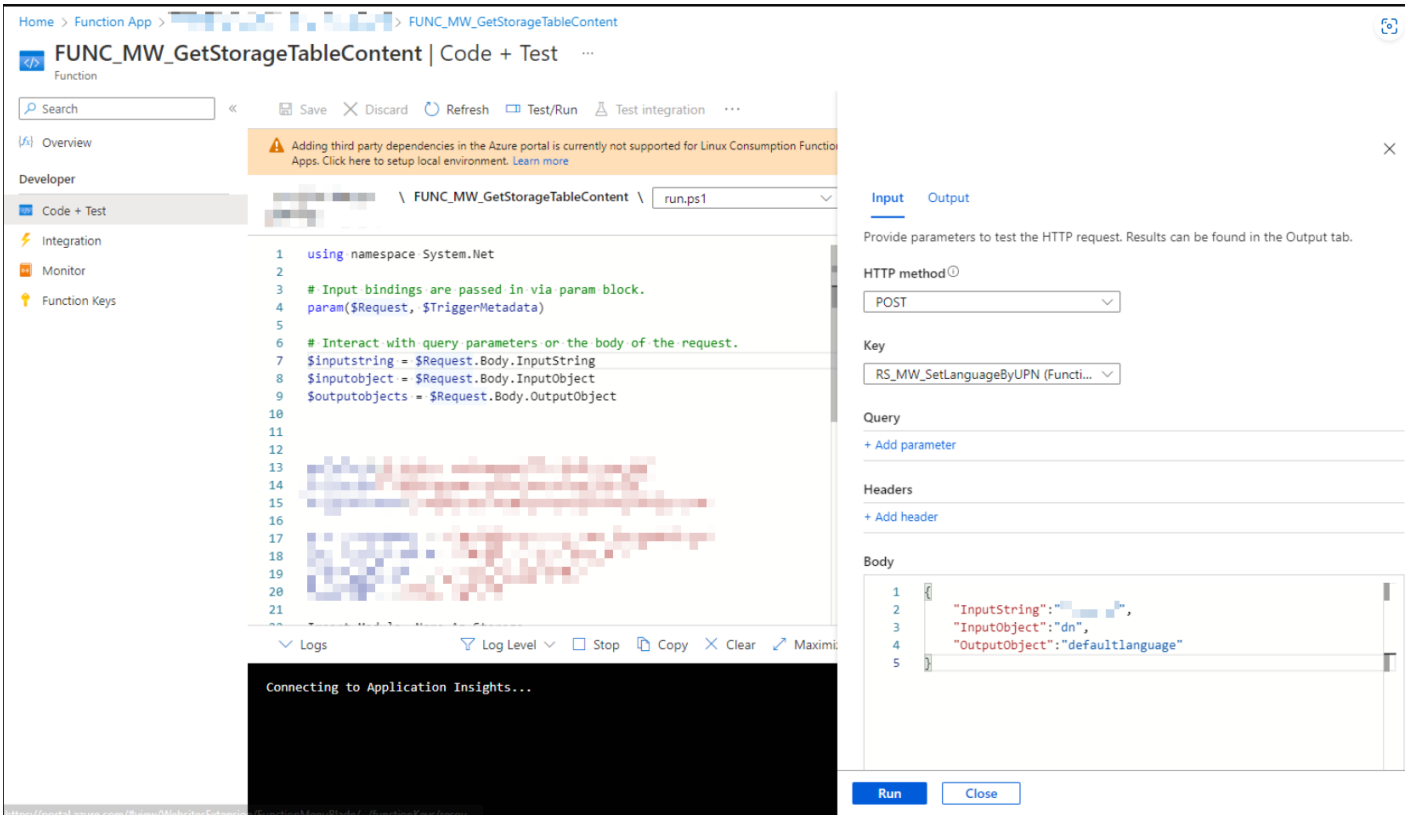
FKEY-<Technology>-<KGTag>-<Object>-<Description>-<Status>-<Location>.
Example: FKEY-RS-ALL-KEY-SetLanguageByUPN-PROD-WE

Triggers

Triggers mean types to call the Azure Function and execute the code.

Manual

The Azure Function can be started manually via the Azure Portal.

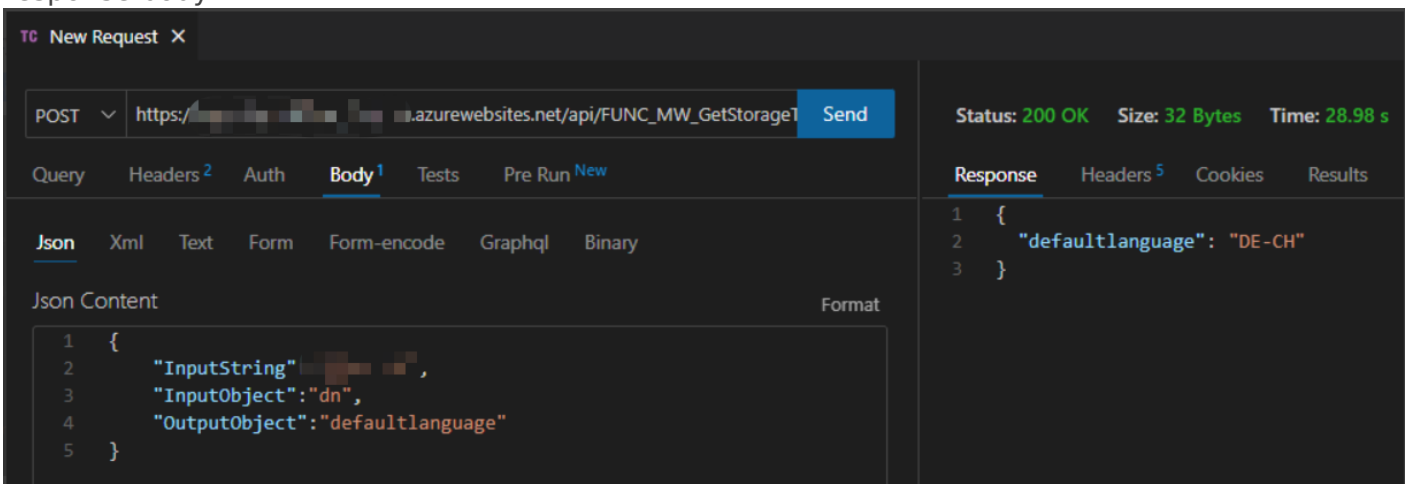


Schedules

Azure Functions can be executed regularly using Schedules. However, since Azure Runbook also offers this function, in most cases it makes more sense to create an Azure Runbook and not functions with schedules.

General REST Call

The function can be triggered via an HTTP request. The return can then also be found in the response body.



PowerShell

Via PowerShell, this can be achieved directly using the standard module "Invoke-Restmethod" with the corresponding URL and the function key.

```
$url="https://<azurefunctionappname>.azurewebsites.net/api/<azurefunction>?code=<azurefunction
key>"
$Body = @"
{
    "InputString":"$TestVariable",
}
"@

$Output = (Invoke-Restmethod -uri $url -Body $Body -Method POST -ContentType
"application/json").value
```

How to Function

This is a short guide for creating an Azure Function. This is a high-level concept in order to be able to make the preparations correctly and completely so that no unwanted surprises occur later during implementation.

Prerequisites

Defined goal

A defined goal is necessary that we can check if our end product is doing what we are expecting. This can be as simple as one phrase.

Example:

My Azure Function should receive logs via HTTP and stores them into one log analytics. The response should indicate if the process was executed successfully.

Defined input

To know how to start the code, we first need to define the input(s) we need to get with the request to reach our goal. This can be static values or multiple dynamic values.

Example:

```
{
"logtype":"<logtypetoseperate>",
  "logbody":{
    "dynamictag1":"dynamicvalue1"
      ...
    }
}
```

In this case the object member "logtype" is static and only the value can be filled in on this specific field. On the other hand, "logbody" is dynamic. It should be possible to have one or one hundred members in the sub object "logbody".

Defined output

The output defines if the Function needs to return a value or which HTTP status code is expected.

Example:

Response: 200 OK

```
{
"language":"<languagecode>"
}
```

For the Function to work as intended, it should respond with the HTTP status code 200 and return this JSON structure.

Trigger method

The method is important that we know what oder which object(s) are triggering this function.

Example:

This function should be called from every Windows device via PowerShell. The trigger should be over HTTP.

Logic model

The logic model defines the logic of the code in a very high-level draft.

Example:

1. Get input
2. Check input values
3. Parse data for Log Analytics API
4. Send data to Log Analytics API
5. Get response from Log Analytics API
6. Send response

Decisions

Coding language

Azure Function Apps can execute different code with different languages. This is defined per Azure Function App and cannot be changed. Azure Functions offers these languages for code execution:

- PowerShell
- JavaScript
- Java
- C#

- Python

Billing plan

The billing plan defines if your function runs consumed based billing or premium plan billing. The benefits of each solution can be found here: Pricing - Functions | Microsoft Azure
For most use cases the consumption plan will be more than enough.

Procedure

Create Function App

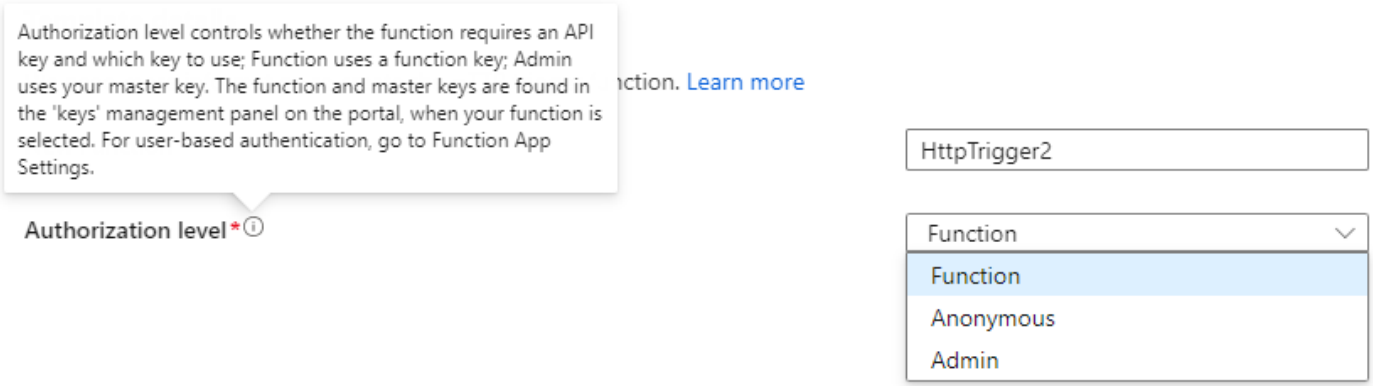
First you need an Azure Function App. On this you have to define the parts defined under "Descisions" and you assign the function into an Azure resource group.

Create Function

The Function itself holds the code which gets executed. It also is the unique identifier in the API URL to call if you use HTTP as a trigger. While creating a Function you can set the name and the trigger. For triggers you can choose between these ones:

Template	Description
HTTP trigger	A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string
Timer trigger	A function that will be run on a specified schedule
Azure Queue Storage trigger	A function that will be run whenever a message is added to a specified Azure Storage queue
Azure Service Bus Queue trigger	A function that will be run whenever a message is added to a specified Service Bus queue
Azure Service Bus Topic trigger	A function that will be run whenever a message is added to the specified Service Bus topic
Azure Blob Storage trigger	A function that will be run whenever a blob is added to a specified container
Azure Event Hub trigger	A function that will be run whenever an event hub receives a new event

After the trigger and the name of the function is set you can choose the authorization level of the function.



For testing purposes, the Anonymous option is probably the easiest. For production functions it is recommended to use the Function level of authorization.

Code

When created you can copy your code into the web editor of the Function or start to develop the code directly in the Azure Portal.

Testing

For testing you can use the built in Test/Run option. If this behaves as it should, you can get the function URL with the push of a button. For testing the built-in default key is enough.

Create Function Key

The best practice is to use Function Keys as authorization on the Function level for production ready APIs. This code is then provided in the URL Query. For each workload which uses the same function it is best practice to use a separate Function Key so the access can be revoked per consumer.

Things to consider

Maximal runtime

The maximal runtime of one Azure Function is 230 seconds. After this time the function code gets cancelled and returns a 503 error.

Centralize log collection with a custom REST API

All environments of a larger IT team write several thousand logs and status messages every day. These logs are usually only stored on the respective systems and cannot be centrally administered or managed. Alarms or monitoring on a log basis in a centralised solution is also not available.

This solution is intended to remedy these problems. On the one hand, a complex topic should be broken down and standardised in a simple solution. The installation of the solution should be as simple as possible and the evaluation of the collected data should be structured in an easily understandable dashboard. Standard components should be used whenever possible, based on standard configurations.

Use case

This API is ideal to create a middleware between the logging client and the logging storage. This brings advantages such as having full control over the secrets, access codes and to take the complexity out of the logging solution on the client. This means Azure Runbooks or Intune Remediation Scripts send the corresponding logs directly to a central log storage pot. This storage pot can then be evaluated and the logs stored and evaluated accordingly. There is also the possibility that monitoring can be built on the logs, as all logs can be stored centrally and for a longer period of time.

In an automated process, a log entry should be created for each important event. The log should be transmitted directly at this point in time, as otherwise the status of the process cannot be clearly traced in the event of a next potentially faulty step.

Function code

This code will run in the Azure Function. It receives two input types via an REST API Body. The LogType sets the table, in which the data will be saved to. The second object is dynamic and can contain as much elements as the creator needs.

Important: To get this to work you have to get the customer id and the shared key from the appropriate log analytics workspace.

```
using namespace System.Net

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

$logtype = $Request.Body.logtype
$logbody = $Request.Body.logbody

$customerId = ""
$sharedKey = ""

Function Build-Signature ($customerId, $sharedKey, $date, $contentLength, $method,
$contentType, $resource){
    $xHeaders = "x-ms-date:" + $date
    $stringToHash = $method + "`n" + $contentLength + "`n" + $contentType + "`n" + $xHeaders +
"`n" + $resource

    $bytesToHash = [Text.Encoding]::UTF8.GetBytes($stringToHash)
    $keyBytes = [Convert]::FromBase64String($sharedKey)

    $sha256 = New-Object System.Security.Cryptography.HMACSHA256
    $sha256.Key = $keyBytes
    $calculatedHash = $sha256.ComputeHash($bytesToHash)
    $encodedHash = [Convert]::ToBase64String($calculatedHash)
    $authorization = 'SharedKey {0}:{1}' -f $customerId,$encodedHash
    return $authorization
}

Function Post-LogAnalyticsData ($customerId, $sharedKey, $body, $logType){
    $method = "POST"
    $contentType = "application/json"
    $resource = "/api/logs"
    $rfc1123date = ([DateTime]::UtcNow).ToString("r")
    $contentLength = $body.Length
    $signature = Build-Signature -customerId $customerId -sharedKey $sharedKey -date
$rfc1123date -contentLength $contentLength -method $method -contentType $contentType -resource
$resource

    $uri = "https://" + $customerId + ".ods.opinsights.azure.com" + $resource + "?api-
version=2016-04-01"
```

```
    $headers = @{
        "Authorization" = $signature;
        "Log-Type" = $logType;
        "x-ms-date" = $rfc1123date;
    }

    $response = Invoke-WebRequest -Uri $uri -Method $method -ContentType $contentType -Headers
$headers -Body $body -UseBasicParsing
    return $response.StatusCode
}

$logbodyjson = ConvertTo-JSON $logbody -Depth 10

#Submit the data to the API endpoint
$params = @{
    CustomerId = $customerId
    SharedKey  = $sharedKey
    Body       = ([System.Text.Encoding]::UTF8.GetBytes($logbodyjson))
    LogType    = $LogType
}
$LogResponse = Post-LogAnalyticsData @params
$LogResponse

if($LogResponse -eq 200){
    Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
        StatusCode = [HttpStatusCode]::OK
        Body = $LogResponse
    })
}
```

This Azure Function returns the Code 200 if the POST Request and the storage in the Azure Log Analytics were successful.

Send logs via PowerShell script

On the client side you can use this PowerShell function to send the logs to the previously created Azure Function. First the Function URL must be inserted into the variable "url". After that you can initialize the function at the very beginning of your PowerShell script. Afterwards you can set the values, that need to be logged, into the Hash Table "Logs". This Hash Table can then be sent via

the Function Send-Logs() to the Azure Function API which then stores the code in the Azure Log Analytics Workspace. That it knows which table it should write the data to, you have to specify the LogType on the Send-Logs() Function.

```
Function Send-Logs(){

    param (
        [String]$LogType,
        [Hashtable]$LogBodyList
    )
    $url="<functionurl>"

    $LogBodyJSON = @"
    {
        "logtype": "$LogType",
        "logbody": {}
    }
"@

    $LogBodyObject = ConvertFrom-JSON $LogBodyJSON
    Foreach($Log in $LogBodyList.GetEnumerator()){
        $LogBodyObject.logbody | Add-Member -MemberType NoteProperty -Name $log.key -Value
$log.value
    }
    $Body = ConvertTo-JSON $LogBodyObject

    $Response = Invoke-Restmethod -uri $url -Body $Body -Method POST -ContentType
"application/json"
    return $Response
}

$Logs = @{
    "<logmember1>"="<logvalue1>"
    "<logmember2>"="<logvalue2>"
}

Send-Logs -LogType "<logtype>" -LogBodyList $Logs
```

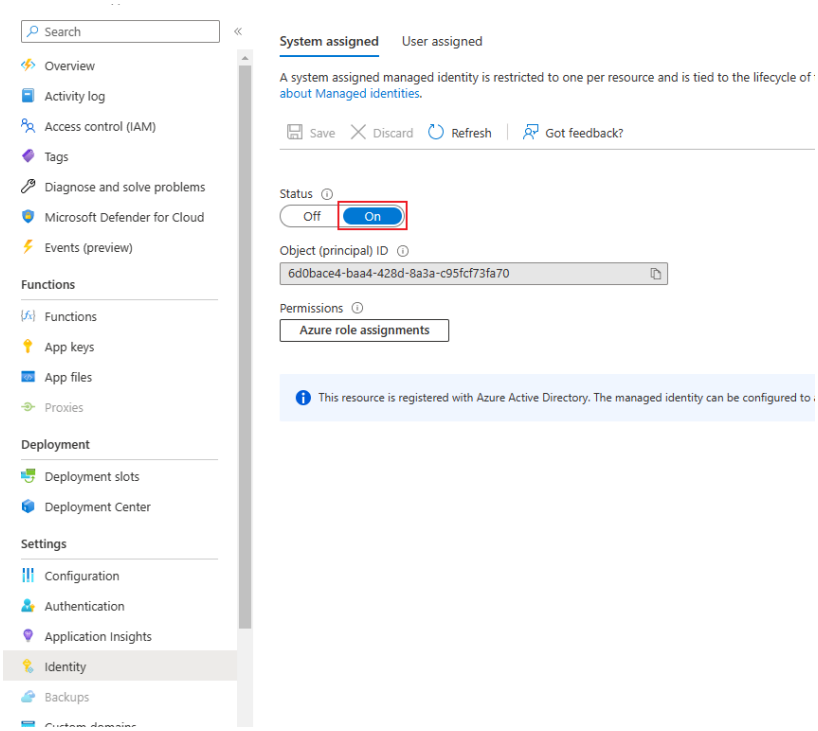
This Function then returns the HTTP status code from the LogCollection API.

Credential handling with Azure KeyVault

With this option, the secrets for Azure Functions are stored in an Azure Key Vault. An access policy is placed on the Key Vault which only allows the Managed Identity of an Azure Function to read the secret. On the Azure Function side, a Managed Identity is set up as well as an Environment Variable that is linked to the Secret in the Key Vault.

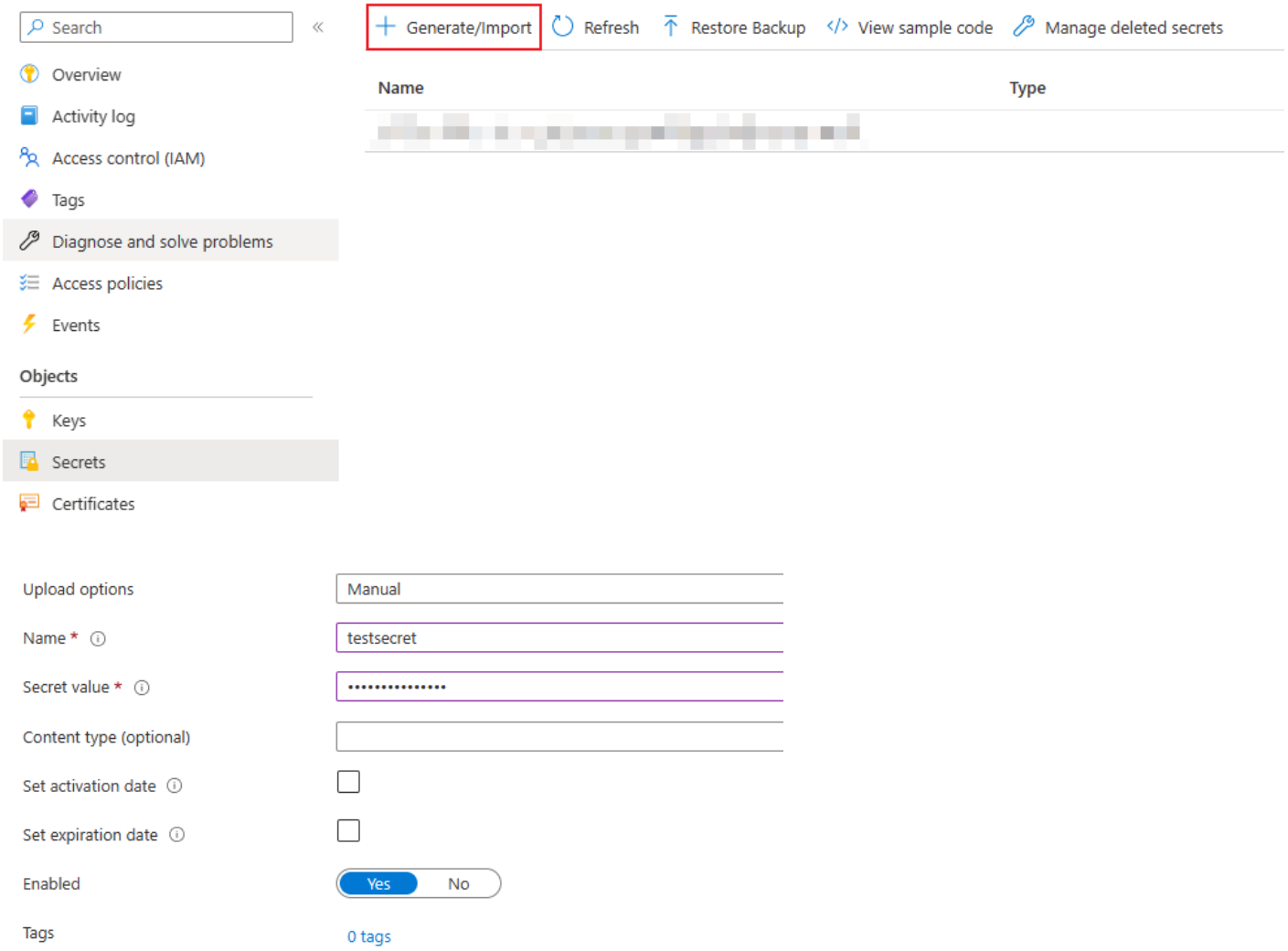
Activate managed identity

To authenticate against the Key Vault you can activate the Managed Identity of your Function App. Thus, permissions can be given to the function app and used within the function.



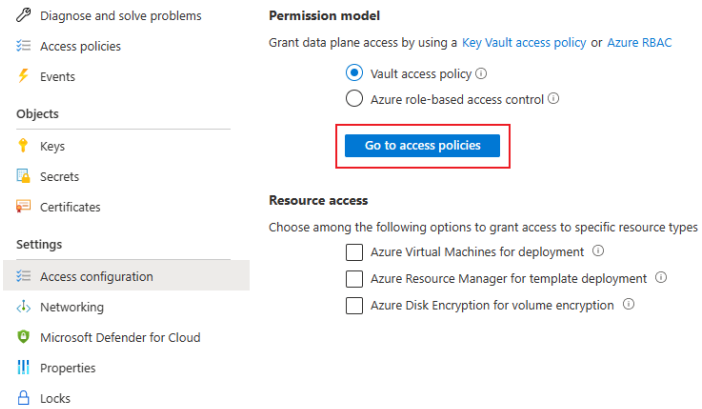
Create new Key Vault Secret

Create a new secret in the key vault and give this secret a unique name. This name will then be used to create a link from the Function App Variable to the Key Vault.



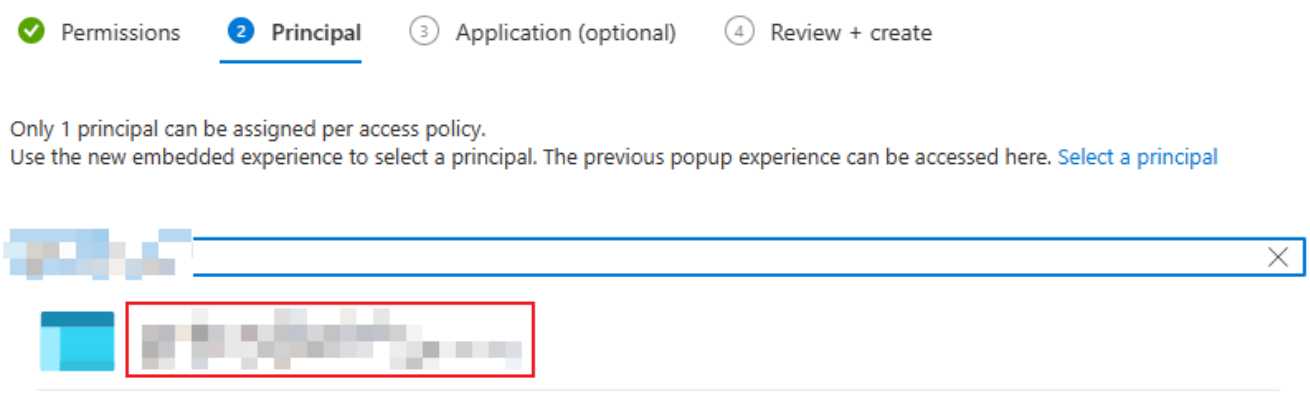
Activate access policies

Go to access policies to create a new permission container for the Azure Function.



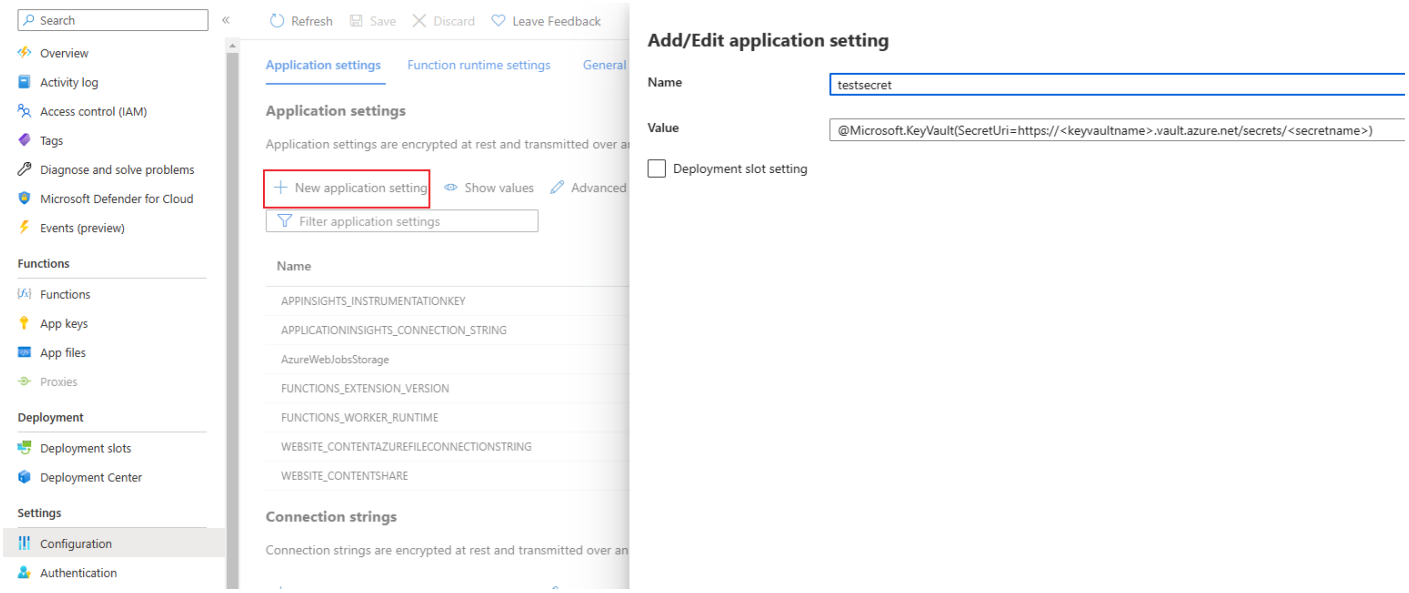
Select the necessary permissions for your function app.

Select the appropriate Managed Identity from the function, which was created earlier.



Add environment variable link

Then you can get the Key Vault name and secret name. With this information you can create an Application setting which will create an environment variable to use in the Azure Functions code. Set the name of the Environment Variable and create a link to the corresponding secret in the Key Vault.



@Microsoft.KeyVault(SecretUri=https://<keyvaultname>.vault.azure.net/secrets/<secretname>)

Usage in function code

When everything is implemented as described you can use the variable accordingly:

```
$testsecret = $env:testsecret
```

Access Azure Function App via OAuth 2.0 authentication

This guide explains how to protect an Azure Function App with Microsoft Entra ID and call it using a bearer token.

The recommended modern approach is to use **App Service / Function Authentication** with Microsoft Entra ID instead of relying only on function keys.

Goal

Protect HTTP-triggered Azure Functions so that only callers with a valid Microsoft Entra access token can execute them.

Recommended approach

Use:

- **Authentication** enabled on the Function App
- **Microsoft** as identity provider
- a dedicated app registration
- **Require authentication**
- a valid **Application ID URI / audience**
- bearer token in the `Authorization` header

Avoid older patterns that rely on exchanging a token via `/.auth/login/aad` unless you have a specific legacy reason.

Configure authentication on the Function App

In Azure Portal:

1. Open the Function App
2. Go to **Settings > Authentication**
3. Add identity provider: **Microsoft**
4. Create or link an app registration
5. Set unauthenticated requests to **Require authentication**

App registration considerations

For the protected API app registration:

- set a proper **Application ID URI**
- expose at least one API scope if user-delegated access is required
- if daemon-to-function access is required, use **application permissions / app roles** as needed

Example audience:

```
api://<function-app-client-id>
```

Example: get bearer token with client credentials

```
$TenantId     = "<tenant-id>"
$ClientId     = "<caller-app-client-id>"
$ClientSecret = "<caller-app-client-secret>"
$Scope        = "api://<function-app-client-id>/.default"
```

```
$TokenBody = @{
    client_id     = $ClientId
    client_secret = $ClientSecret
    scope         = $Scope
    grant_type    = "client_credentials"
}

$Token = Invoke-RestMethod `
    -Method POST `
    -Uri "https://login.microsoftonline.com/$TenantId/oauth2/v2.0/token" `
    -Body $TokenBody `
    -ContentType "application/x-www-form-urlencoded"
```

Best practices

- require authentication globally
- use Entra ID instead of public function keys for privileged APIs
- restrict accepted audiences
- separate caller app registration from protected API app registration
- prefer Managed Identity for Azure-to-Azure calling patterns

Summary

For secure Azure Function access in enterprise environments, protect the Function App with Microsoft Entra authentication and require valid bearer tokens.

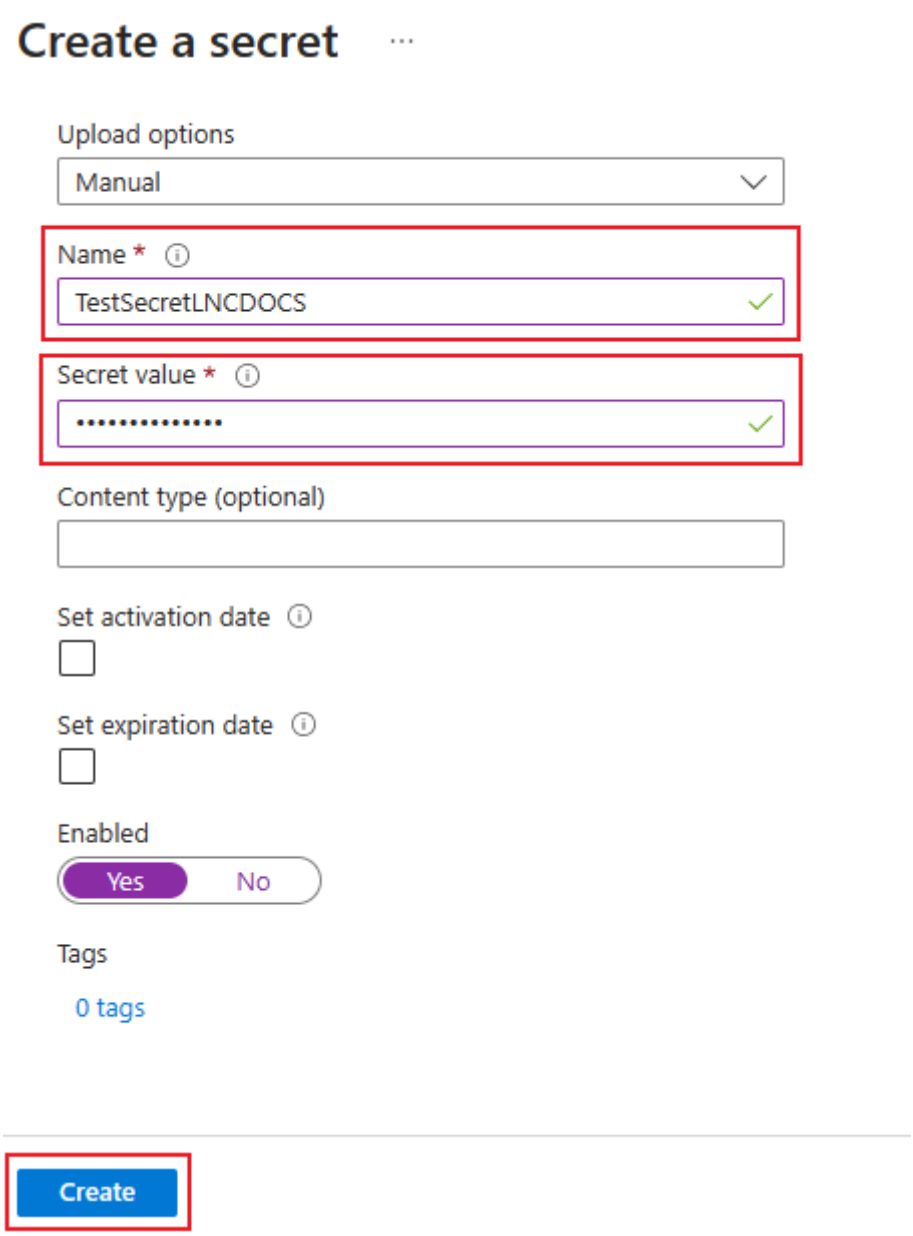
Create reference to Azure Key Vault content from function code

> Requirements: Basic Azure Function knowledge and access to an Azure Key Vault & Azure Function.

This topic shows you how to work with secrets from Azure Key Vault in your Azure Functions code without requiring any code changes. Azure Key Vault is a service that provides centralized secrets management, with full control over access policies and audit history. This article uses managed identities to access other resources.

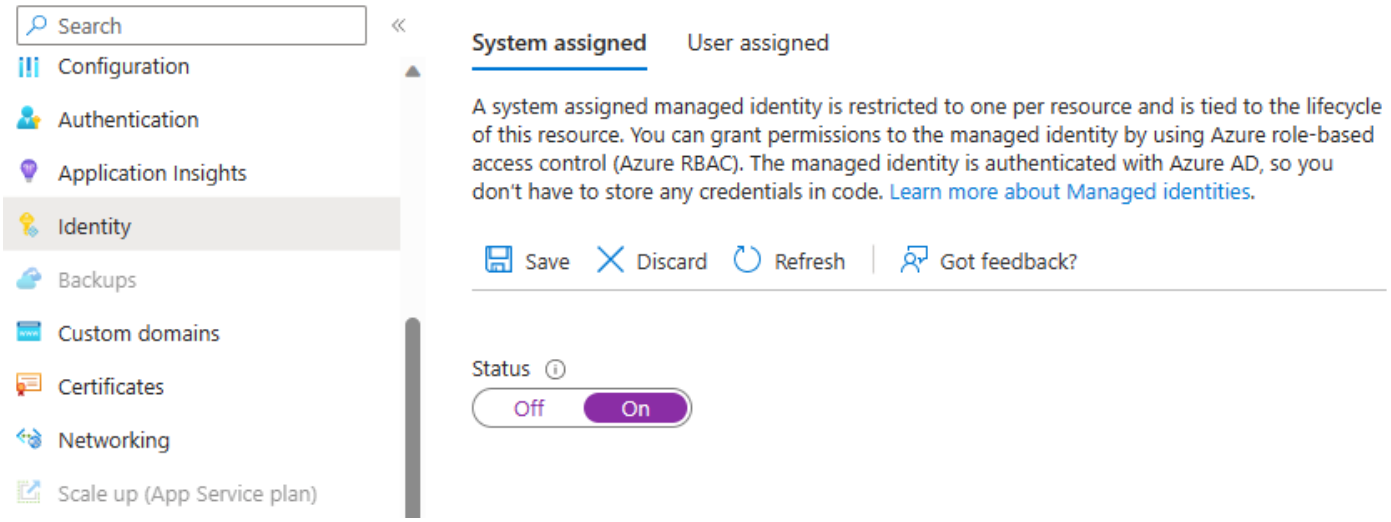
Add your secret to Azure Key Vault

First, the secret must be created in the Azure Key Vault. For this, an Azure Key Vault must exist and the permissions to create a new item must be available. There you can insert the secret that you want to use later in the code.



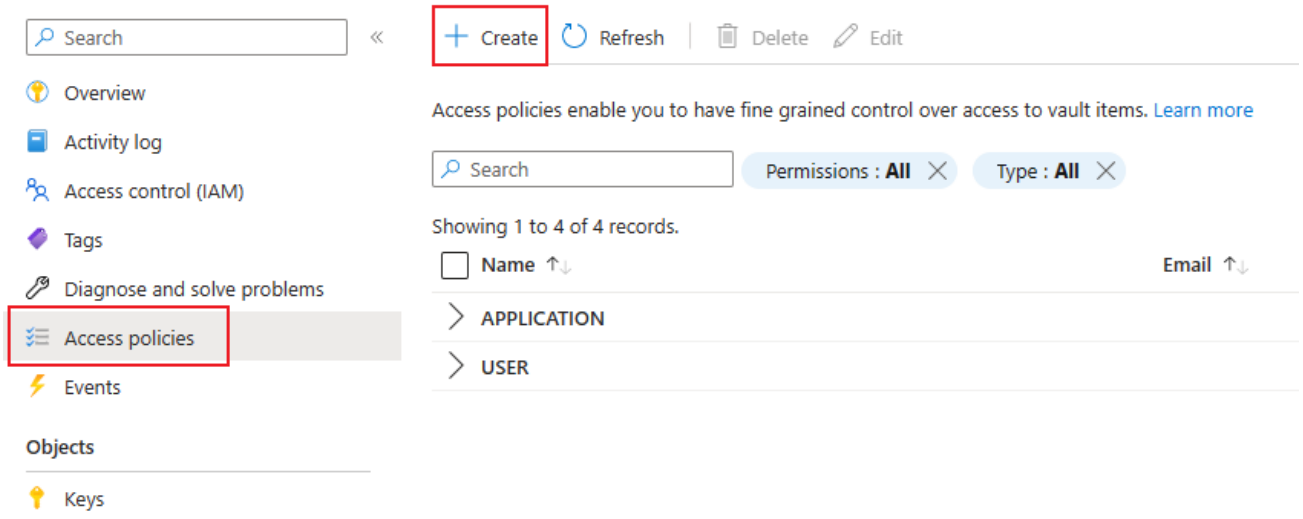
Create managed identity of function

To be able to display the secret in the Function code, you have to activate the Managed Identity in the Function App. You can do this via the menu item "Identity" and then switch the status to "On" under "System assigned". Don't forget to save your selection.

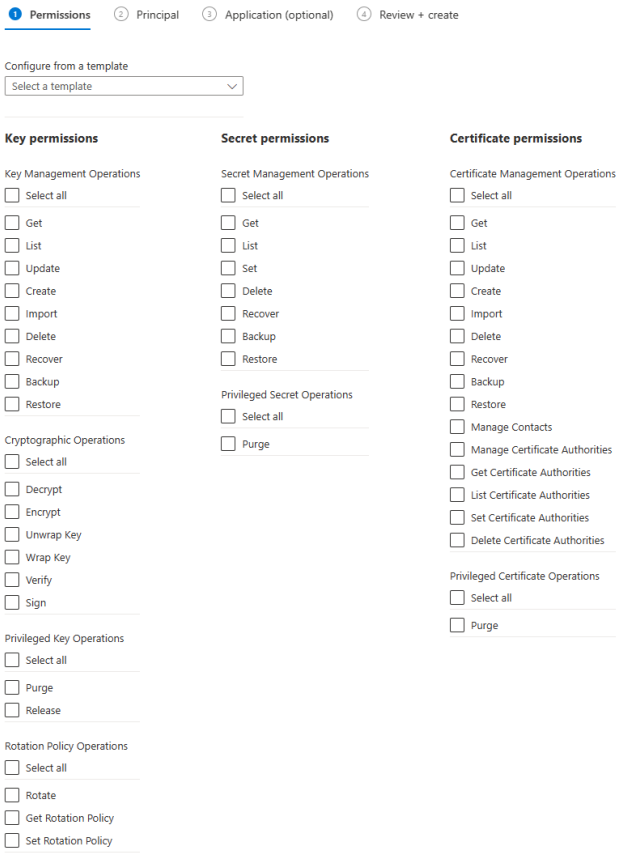


Create access policy

Then you can go back to the Key Vault and create a new access policy under "Access policies" -> "Create".



There you have to select the desired permissions. The Azure Function then connects to the Key Vault with these permissions. To read only the secret content, only the "Get" permission under "Secret permissions" is used.



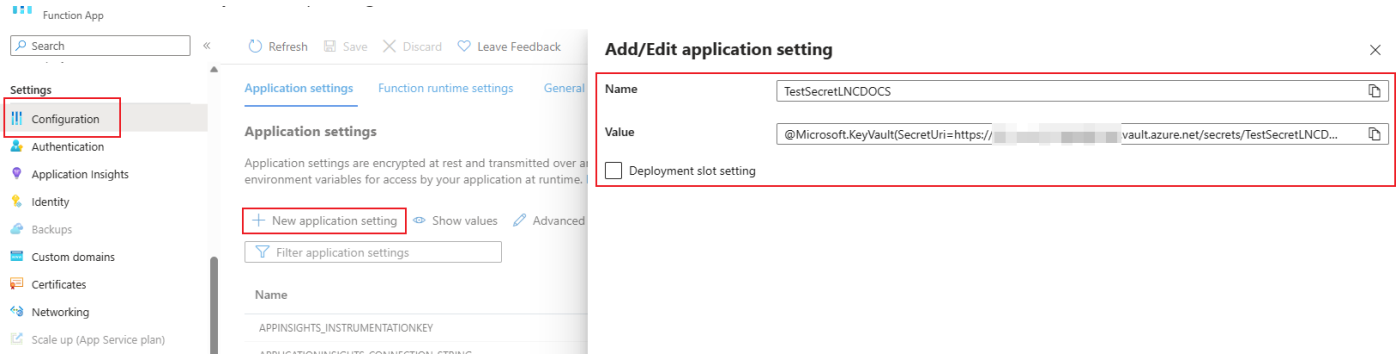
At the end you have to select the managed identity of the Azure Function and save the access policy.

Create environment variable link

Then a link to the Key Vault can be created. To do this, you must create a new application secret under "New application setting" in the Function App under "Configuration".

Here you first enter the name of the variable that you want to address in the code. Under "Value" you have to insert the following content and complete it with your values:

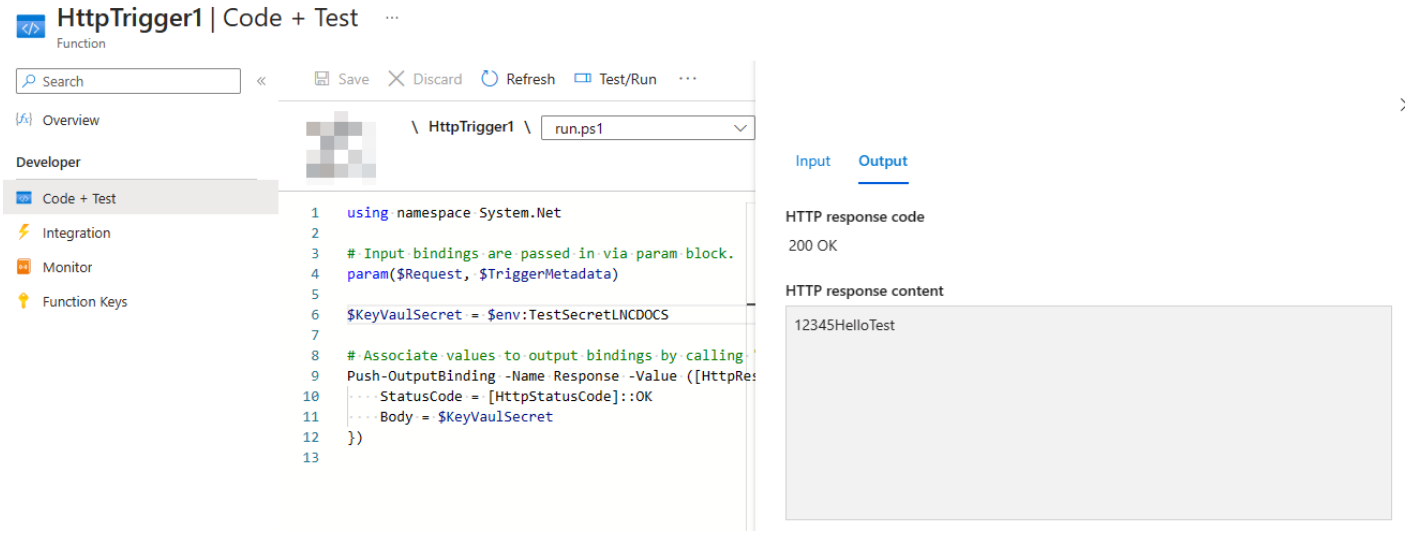
@Microsoft.KeyVault(SecretUri=https://<keyvaultname>.vault.azure.net/secrets/<secretname>



Get environment variable content

Then you can read the variable in your function code using the environment as follows:

$env:<secretname>



Thus, the values can be stored securely without all user accounts needing authorization.

Run PowerShell Code from frontend on backend using Azure Function

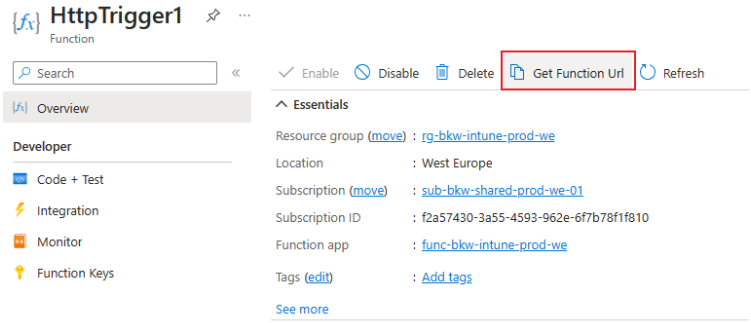
This tutorial shows how to build a REST API that executes PowerShell code using an Azure Function. This code can then return values and objects as JSON responses using "return".

Using Function

To use this function you need the following:

- Azure Function with server-side code (manual below).
- Entra ID App Registration credentials with the permissions you need: [Get app details and gr... | LNC DOCS (lucanoahcaprez.ch)]
- PowerShell code you want to run on the server (example below).

To execute the code on the server, you need to send a JSON to the function URL via POST request. It is important that the POST call goes to the correct URL. You can get this URL on the function level via this button:



The URL is structured as follows:

```
https://<yourfunctionappname>.azurewebsites.net/api/<yourfunctionname>?code=<yourfunctionkey>
```

The JSON which is needed consists of the main key "parameters", which then must have the following four keys. The keys starting with "microsoft" are needed for the authentication. The "powerShellCode"-key contains the final code which will be executed.

It is also useful to know that all other keys, which are created below "parameters", can be used as variables in the PowerShell script. As an example: If a new key named "UserLanguage" is added, the $UserLanguage variable can be used in PowerShell.

```
{
    "parameters":{
        "microsoftTenantID":"<yourtenantname>",
        "microsoftClientID":"<yourappregistrationclientid>",
        "microsoftClientSecret":"<yourappregistrationclientsecret>",
        "powerShellCode":"<yourpowershellcode>"
    }
}
```

Example Request

Here is a sample request that allows to fetch all user data from Microsoft Entra ID via Graph API.

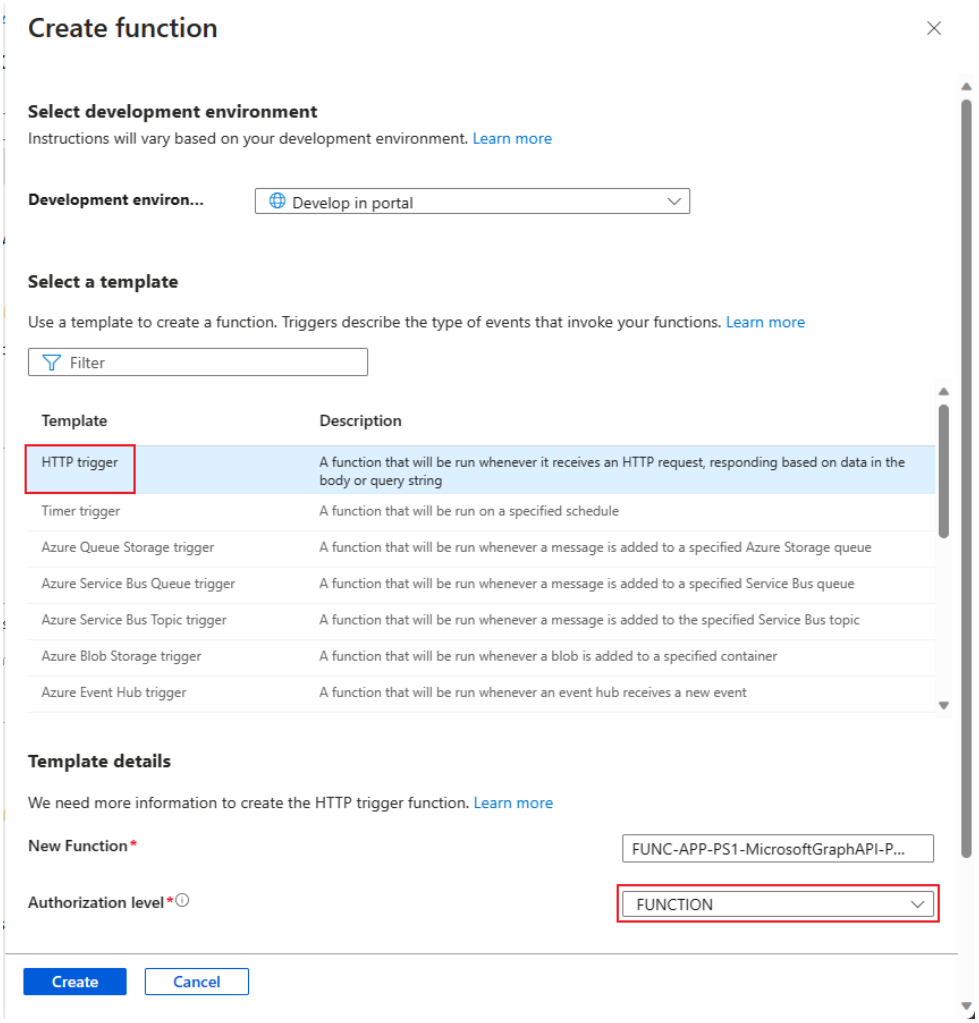
```
POST
https://<yourfunctionappname>.azurewebsites.net/api/<yourfunctionname>?code=<yourfunctionkey>

{
    "parameters":{
        "microsoftTenantID":"<yourtenantname>",
        "microsoftClientID":"<yourappregistrationclientid>",
        "microsoftClientSecret":"<yourappregistrationclientsecret>",
        "powerShellCode":"$Uri = \"https://graph.microsoft.com/v1.0/users\"\n$Result = Invoke-
RestMethod -Uri $Uri -Body $requestBody -Headers @{Authorization = $Global:AzureADAccessToken;
ConsistencyLevel = 'eventual' } -Method GET -ContentType 'application/json'\n$Members =
$Result.value\nwhile ($Result.'@odata.nextLink') {\n    $Result = Invoke-RestMethod -Uri
$Result.'@odata.nextLink' -Headers $Header\n    $Members += $Result.value\n}\nreturn $Members"
    }
}
```

Setup Function

For this Azure Function you will need an Azure Function App. It must be able to execute PowerShell code. In the Function App you have to create a Function with the HTTP triiger and FUNCTION

Authorization level.



This function then acts as the container where the server-side code runs. The Azure Function framework then accepts the REST calls and does the load balancing and handling of the HTTP requests.

Server-side PowerShell Code

This code is used on the Function to handle the requests. It is important to know that this is specifically for handling the incoming JSON and responses.

```
using namespace System.Net

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Function for returning Values to Request
function Return-FunctionValue{
    param(
        [boolean] $Error,
```

```
        [string] $StatusCodeString,
        $OutputBody
    )
    switch -exact ($StatusCodeString.ToUpper()) {
        "OK" {
            $StatusCode = [System.Net.HttpStatusCode]::OK
        }
        "NOTFOUND" {
            $StatusCode = [System.Net.HttpStatusCode]::NotFound
        }
        "NOCONTENT" {
            $StatusCode = [System.Net.HttpStatusCode]::NoContent
        }
        "BADREQUEST" {
            $StatusCode = [System.Net.HttpStatusCode]::BadRequest
        }
        "INTERNALSERVERERROR" {
            $StatusCode = [System.Net.HttpStatusCode]::InternalServerError
        }
        default {
            $StatusCode = $null
        }
    }
    if($Error){
        $OutputBody = @{
            "statusCode" = $StatusCode
            "errorMessage" = $OutputBody
        }
    }
    Write-Output $StatusCode
    Write-Output $OutputBody
    Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
        StatusCode = $StatusCode
        Body = $OutputBody
    })
    Exit
}

try{
    $BodyParameters = [PSCustomObject]$Request.Body.parameters
```

```
    Foreach($BodyParameter in $Bodyparameters.PSObject.Properties) {
        New-Variable -Name $BodyParameter.Name -Value $BodyParameter.Value
    }
}
catch{
    Return-FunctionValue -StatusCodeString "InternalServerError" -OutputBody "Parameters
provided could not be converted to variable" -Error $True
}

$powerShellCode = $powerShellCode.Replace('\n', "`n")

# Function for getting Azure AD Access Header
Function Build-MicrosoftEntraIDApplicationAccessHeader(){
    param(
        [Parameter(Mandatory=$true)]
        [string] $TenantID,
        [string] $ClientID,
        [string] $ClientSecret,
        [string] $refreshtoken
    )

    $authenticationurl = "https://login.microsoftonline.com/$TenantID/oauth2/v2.0/token"

    if($refreshtoken -and $TenantID){
        $tokenBodySource = @{
            grant_type = "refresh_token"
            scope = "https://graph.microsoft.com/.default"
            refresh_token  = $refreshtoken
        }
    }
    elseif($TenantID -and $ClientID -and $ClientSecret){
        $tokenBodySource = @{
            grant_type = "client_credentials"
            scope = "https://graph.microsoft.com/.default"
            client_id  = $ClientID
            client_secret = "$ClientSecret"
        }
    }
    else{
        Return-FunctionValue -StatusCodeString "BadRequest" -OutputBody "Authentication
```

```
failed: Not all parameters provided for authentication" -Error $True
    }

    while ([string]::IsNullOrEmpty($AuthResponse.access_token)) {
        $AuthResponse = try {
            Invoke-RestMethod -Method POST -Uri $authenticationurl -Body $tokenBodySource
        }
        catch {
            $ErrorAuthResponse = $_.ErrorDetails.Message | ConvertFrom-Json
            if ($ErrorAuthResponse.error -ne "authorization_pending") {
                Write-Output "error"
                Return-FunctionValue -StatusCodeString "BadRequest" -OutputBody
"Authentication failed: Error while posting body source: $($ErrorAuthResponse.error)" -Error
$True
            }
        }
    }

    if($AuthResponse.token_type -and $AuthResponse.access_token){
        $global:MicrosoftEntraIDAccessToken = "$($AuthResponse.token_type)
$($AuthResponse.access_token)"
        $global:Header = @{
            "Authorization" = "$global:MicrosoftEntraIDAccessToken"
        }
        Write-Output "Authorization successful! Token saved in variable."
    }
    else{
        Return-FunctionValue -StatusCodeString "BadRequest" -OutputBody "Authentication
failed: Not all parameters provided for authentication" -Error $True
    }
}


# Authorization Header with ClientId & ClientSecret
try{
    if(($microsoftTenantID) -and ($microsoftClientID) -and ($microsoftClientSecret)){
        Build-MicrosoftEntraIDApplicationAccessHeader -tenantid $microsoftTenantID -clientid
$microsoftClientID -clientSecret $microsoftClientSecret
    }
    else{
```

```
        Return-FunctionValue -StatusCodeString "BadRequest" -OutputBody "Authentication
failed: Not all parameters provided for authentication" -Error $True
    }
} catch{
    Return-FunctionValue -StatusCodeString "InternalServerError" -OutputBody "Bearer token
could not be created with provided parameters" -Error $True
}

# Run PowerShell Code
try{
    if($PowerShellCode){
        $OutputBody = Invoke-Expression $PowerShellCode
        Return-FunctionValue -StatusCodeString "OK" -OutputBody $OutputBody -Error $False
    }
    else{
        Return-FunctionValue -StatusCodeString "BadRequest" -OutputBody "PowerShell Code
failed: No PowerShell Code provided for runtime" -Error $True
    }
}
catch{
    Return-FunctionValue -StatusCodeString "InternalServerError" -OutputBody "PowerShell Code
provided did not run correctly" -Error $True
}

# Return-FunctionValue -StatusCodeString "NOCONTENT" -OutputBody ""
```

Extend KQL Data with Azure Function as Resource Exporter

In complex cloud environments, effective monitoring and observability increasingly require data from sources beyond native Azure telemetry. While Azure native resources in combination with Azure Log Analytics offer deep visibility into Azure resources, they don’t directly support querying external systems such as Microsoft Graph API, or third-party platforms like ServiceNow, CRMs, etc. To bridge this gap, you can extend the telemetry using Azure Functions that collect, transform, and retrieve external data on demand using HTTP requests and making it accessible through Kusto Query Language (KQL).

> This is a **conceptual architecture** meant to demonstrate how you can enrich your observability stack. It’s not a one-click or fully packaged solution, but rather a pattern that can be tailored to your specific API sources, data schemas, and monitoring goals.

Architecture

At its core, the approach involves using Azure Functions as external data connector — fetching data from APIs, converting it into structured records, and making it available to your existing monitoring data platform. Once the Azure Function is created, it becomes available to query, visualize, and alert on. It then works just like native or custom Azure logs.

This pattern unlocks powerful use cases, such as:

- Correlating Entra ID sign-ins with metadata from ServiceNow or Microsoft Graph API.
- Contextualizing security alerts with vulnerability data from external scanners.
- Tracking compliance-related signals alongside Azure Policy states.

In the sections that follow, it is outlined how this architecture can be implemented, the technologies involved, and important operational considerations for building a reliable, secure, and maintainable integration.

Azure Function

The Azure Function acts as a lightweight API endpoint that retrieves data from external systems, transforms it into structured JSON, and serves it to KQL using an HTTP trigger. This allows real-time enrichment of queries with external context.

Authentication

To securely access external APIs like Microsoft Graph, your Azure Function must authenticate properly. The recommended method is using **Managed Identity**, which allows the function to obtain tokens without storing secrets.

Ensure the function's identity has the necessary Graph API permissions (e.g., `Group.Read.All`) granted via Entra ID and admin consent. Learn more about System Managed Identities here: Use System Managed Ide... | LNC DOCS

Example Code

```
using namespace System.Net

param($Request, $TriggerMetadata)

if($request.Query.evaluationType -eq "entraidgroups" -and $request.Query.groupid){
    # Get Graph API Access Token
    Import-Module Microsoft.Graph.Authentication
    $requiredPermissions = @(
        "Group.Read.All"
    )
    $permissionsList = $requiredPermissions -join ', '
    Connect-MgGraph -Identity -NoWelcome
    $mgGraphRequest = Invoke-MgGraphRequest -Uri "https://graph.microsoft.com/v1.0/" -
OutputType HttpResponseMessage
    $Global:MicrosoftEntraIDAccessToken =
$mgGraphRequest.RequestMessage.Headers.Authorization.Parameter

    $uri =
"https://graph.microsoft.com/v1.0/groups/$($request.Query.groupid)/members"#?`$select=id,displ
ayName"
    $allGroups = Invoke-RestMethod -Method GET -Uri $uri -Header @{Authorization = "Bearer
$Global:MicrosoftEntraIDAccessToken" }
```

```
    $outputjson = ($allGroups.value)
}

Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = [HttpStatusCode]::OK
    Body = $outputjson
})
```

KQL Query

Once your Azure Function is returning structured JSON data, you can integrate it into your monitoring workflow using `externaldata()` in KQL:

```
externaldata(id:string, displayName:string) // Extend with fields from your API response
[

'https://<functionappname>.azurewebsites.net/api/<functionname>?code=<functionaccesskey>&evalu
ationType=entraidgroupmembers&groupid=<yourgroupid>'
]
with(format='multijson')
```

Key Considerations

- **Schema Definition**: The columns in `externaldata()` must match the structure of the JSON returned by your Function. Incorrect or missing fields will cause the query to fail or return nulls.
- **Security**: Avoid embedding access keys directly in shared queries. Use managed identity or securely store secrets if possible.
- **Performance**: This pattern pulls data at query time — ideal for on-demand enrichment, but not suitable for high-frequency or large-scale queries. Cache results where needed.
- **Error Handling**: Ensure your Function returns consistent, valid JSON. Add logging and status code checks to troubleshoot failed queries.

With the right setup, this method allows seamless integration of external data into your KQL dashboards and alerts, expanding the scope of what you can monitor and correlate in Azure.

Secure and Centralized Secret Handling with mTLS in Azure-Certificate-Secret-Proxy

This project is a community-built reference implementation for secure secret delivery to managed endpoints (Windows, macOS, Linux) using mutual TLS (mTLS).

GitHub repository: https://github.com/lucanoahcaprez/Azure-Certificate-Secret-Proxy

TL;DR

This solution enables certificate-based, centralized secret retrieval for managed endpoints without distributing static secrets to clients. Devices authenticate with mTLS, the function enforces validation policy, and secrets are fetched on demand from controlled backends.

Problem Statement

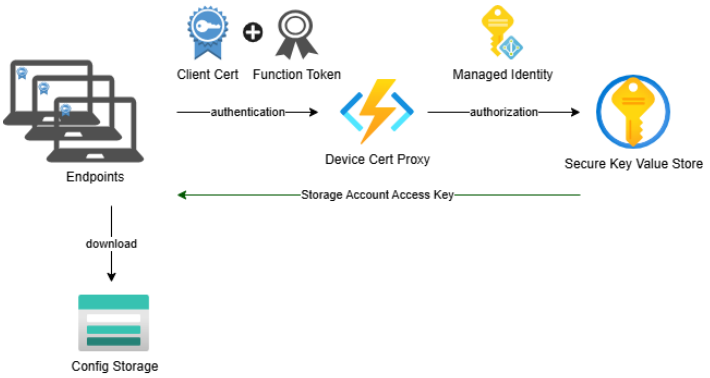
The core issue is persistent secret exposure on endpoints due to packaging and script distribution patterns.

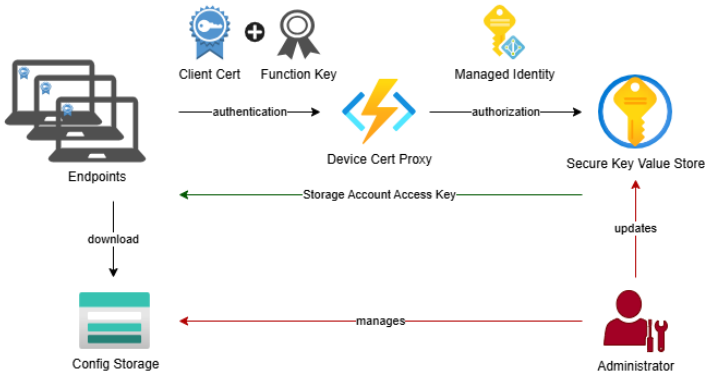
Many endpoint automation workflows still expose secrets in clear text or leave them recoverable on disk (for example in remediation scripts, package payloads, or local logs). This creates long-lived secret exposure on clients.

This solution removes static secret distribution and shifts secret access to a certificate-authenticated, server-side retrieval model.

Technical Overview

This is the general architecture:





The request path is intentionally minimal and policy-driven so trust and retrieval decisions stay server-side.

1. Client presents a device certificate during TLS handshake.
2. Azure App Service enforces client-certificate requirement.
3. Function receives and validates certificate (`X-ARR-ClientCert`).
4. Validation policy is applied (`EntraDeviceCert`, `CertChainValidation`, `TrustedThumbprints`, or combination).
5. Requested secret is retrieved from selected backend (`APPSETTINGS`, `KEYVAULT`, `TABLE`).
6. Response is returned as scoped JSON payload.

Security Properties

The following controls define the baseline security posture of the current implementation.

Control	Implementation
Strong client auth	mTLS with device certificate proof-of-possession

Control	Implementation
Policy-based trust	Entra device binding, custom root trust, thumbprint allowlist
Secret minimization	Per-request retrieval, no static local secret bundle
Credentialless backend auth	Managed Identity for Azure APIs
Operational diagnostics	Structured response diagnostics and centralized logging

Additional Notes

The following sections cover practical application scope, fit, and roadmap direction.

Example Use Cases

Typical scenarios are operational tasks that currently rely on static or embedded secrets.

- Shared device local admin password retrieval
- License/key distribution (for example ESU)
- BIOS/firmware secret retrieval
- Storage SAS URL delivery
- Log ingestion bootstrap (for example LAW/DCR scenarios)
- Central configuration values for certificate-enabled endpoints
- Server scenarios where IMDS/Arc onboarding is not available

Architecture Fit

This implementation is a reusable pattern, not a one-size-fits-all product.

This is a concept architecture and should be adapted to tenant-specific controls, compliance boundaries, and operational requirements.

Planned V2 Enhancements

Planned improvements focus on cryptographic hardening, resilience, and operational visibility.

- Application-layer payload encryption with ephemeral key exchange
- Automated key rotation primitives
- Rate limiting and abuse controls (plus DDoS hardening)
- Network isolation with private endpoints

- Improved telemetry, traceability, and anomaly detection

Documentation Entry Points

Use the links below for implementation details, deployment instructions, and architecture deep dives.

- https://github.com/lucanoahcaprez/Azure-Certificate-Secret-Proxy/tree/main/docs
- https://github.com/lucanoahcaprez/Azure-Certificate-Secret-Proxy/blob/main/docs/ARCHITECTURE.md
- https://github.com/lucanoahcaprez/Azure-Certificate-Secret-Proxy/blob/main/docs/DEPLOY.md